technische universität
dortmund

PG536

Endbericht

Optimierte Inferenzkontrolle für relationale Datenbanken

Teilnehmer:

Lorenzo Benet
Nguyen Binh Bui
Mohammed Erguig
Adalat Jabrayilov
Nils Maybaum
Peter Ndula
Torsten Schlotmann
Sarah Stahl
Marcus Teufel
Hui Zheng

Betreuer:

Prof. Dr. Joachim Biskup
Dipl.-Inf. Cornelia Tadros

10. Mai 2010

Inhaltsverzeichnis

Zusammenfassung **1**

I Seminarphase **2**

1 Grundlagen der Kontrollierten Anfragenauswertung **3**

1.1 Warum wird Inferenzkontrolle benötigt? 3

1.2 Logik-orientierte Informationssysteme 4

1.3 Wie funktioniert Inferenzkontrolle? 5

1.3.1 Arten von Informationsfluss 6

1.4 Wie funktioniert die Kontrollierte Anfrageauswertung? 7

1.4.1 Wie funktioniert Verweigern 9

1.4.2 Wie funktioniert Lügen 10

1.4.3 Wie funktioniert die Kombinierte-Methode 11

1.5 Statische Inferenzkontrolle 12

1.5.1 k-Anonymität 12

1.5.2 l-Diversität 14

2 Sichere Anfragen **20**

2.1 Einführung 20

2.1.1 Motivation 20

2.1.2 Unterschiedliche Anfragesprachen 21

2.1.3 Hinzufügung der Negation 21

2.1.4 Folge von der Negation 21

2.2 Semantische Einschrünkung 22

2.2.1 Relationaler Kalkül 22

2.2.2 Unsichere Anfragen 23

2.2.3 Relativierte Interpretation 24

2.2.4 Aktive Domäne Semantik 25

2.2.5 Domänenunabhängige Anfragen 26

2.3 Syntaktische Einschränkung 27

2.3.1 Motivation 27

2.3.2 SRNF (Safe Range Normal Form) 28

2.3.3 Algorithmus Range Restriction(rr) 29

3 Implikation **31**

3.1 Grundlagen 31

3.1.1 Prädikatenlogik 31

3.1.2 Logische Implikation 33

3.1.3 Verschiedene Implikationen 34

3.1.4 Implikationsproblem 34

3.1.5 Entscheidbarkeit 35

3.2 Bernays-Schönfinkel-Klasse 36

3.3 Annahmen über Konstanten und Domänen 37

3.3.1 Feste endliche Domäne 37

3.3.2 Variierende endliche Domäne 38

3.3.3 Feste unendliche Herbrand-Domäne 38

3.4 Relationales Submodell 39

3.4.1 Grundlagen 39

3.4.2 Entscheidbarkeit der Implikationen 42

4 Automatisches Beweisen 44

4.1 Einleitung 44

4.1.1 Was bedeutet „automatisches Beweisen“? 44

4.1.2 Modellierung 44

4.2 Logische Grundlagen 45

4.2.1 Aussagenlogik [AL] 45

4.2.2 Prädikatenlogik 1. Stufe [PL1] 51

4.3 Resolution 58

5 Optimierung des statischen Zensors 61

5.1 Motivation 61

5.2 Definitionen 62

5.3 Auf Sicherheit basierende Klassifikation der Datenbanken 65

5.4 Algorithmen 67

5.4.1 Linearzeit-Algorithmus 67

5.4.2 Logarithmische Zeit-Algorithmus 68

5.4.3 Effiziente Berechenbarkeit von Zugriffskontrolle 69

5.5 Korrektheit 70

5.6 Kritik 74

6 JDBC **76**

6.1 Einführung 76

6.2 JAVA DataBase Connectivity 76

6.3 JDBC-Treibertypen 77

6.3.1 Treibertyp 1 77

6.3.2 Treibertyp 2 78

6.3.3 Treibertyp 3 78

6.3.4 Treibertyp 4 78

6.4 Prepared Statements 79

6.5 Transaktionen 81

6.5.1 Begriffe 81

6.5.2 Eigenschaften von Transaktionen 82

6.5.3 Transaktionssteuerung 82

6.6 Anwendungsbeispiel 82

6.6.1 Java-Quellcode 83

6.6.2 Datenbanktabelle 85

7 Oracle - Virtual Private Database 87

7.1 Oracle Virtual Private Database 87

7.1.1 PL/SQL 87

7.1.2 Security Policy - Basics 98

7.1.3 Virtual Private Database 99

7.1.4 Application Context 103

8 Sicht-Änderungen 106

8.1 Sichten 106

8.1.1 Beispiel für eine Sicht 107

8.2 Sichterhaltungsproblem(View Maintenance Problem) 107

8.2.1 Lösungsansatz 108

8.3 Sichtänderungsproblem 108

8.3.1 Änderungssemantik von relationalen Sichten 109

8.3.2 Sichtänderungen unter der kontrollierten Anfragenauswertung 111

8.3.3 Lösungsansatz zum Sichtänderungsproblem 113

8.4 Multilevel Security und Polyinstantiierung 113

8.4.1 Multilevel Secure Databases 113

8.4.2 Polyinstanziierung 113

9 Inferenzfreie Sicht-Änderung 121

9.1 Anforderungen an inferenzfreie Sicht-Änderungen 121

9.1.1 Anforderungen aus Sicht eines Benutzers 121

9.1.2 Anforderungen aus Sicht des Administrators 122
9.2 Logik negierter Variablen 124
9.2.1 Formel und Variablennegation 124
9.2.2 Eigenschaften der Variablennegation auf Formeln 126
9.2.3 Anwendung der Variablennegation 126
9.3 Kontrollierte Anfragen mit Sicht-Änderungen 128
9.3.1 Ein sicherer Algorithmus zur Sicht-Änderung 129
9.3.2 Änderungs-Algorithmus für atomare Fakten 132

10 Inferenzfreie Sichterneuerung 134

10.1 Motivation 134
10.1.1 Was ist eine Sicht? 134
10.2 Materialisierte Sichten 135
10.2.1 Definition 135
10.2.2 Anwendungsbereiche 135
10.2.3 Das Sichtwartungsproblem 136
10.3 Sichterneuerung unter atomaren Änderungen 137
10.3.1 Grundidee atomarer Änderungen 137
10.3.2 Korrektheit und Vertraulichkeit 138
10.3.3 Konstante Sicht bei atomaren Änderungen 141
10.3.4 Atomare Änderungen in vollständigen Datenbanken 145

II Entwürfe 148

11 Entwurf des Mehrbenutzersystems 149

11.1 Einführung 149

11.2 Benutzer anlegen 150

11.3 Benutzer entfernen 150

11.4 Benutzer bearbeiten 151

11.5 Rollen- und Privilegienvergabe 151

11.6 ToDos 152

12 Entwurf der Sicht-Änderung 153

12.1 Motivation 153

12.1.1 Was ist das Problem? 153

12.1.2 Warum ist das Problem ein Problem? 153

12.1.3 Wie sieht die Lösung aus? 154

12.1.4 Wieso ist die Lösung eine Lösung? 154

12.2 Operationalisierung 154

12.2.1 Aktivitätsdiagramm Sichtänderung 154

12.2.2 Sequenzdiagramm Sichtänderung - Fall2 158

12.2.3 Kommentierung der Methode *ViewUpdate.update*() 158

12.2.4 Kommentierung der Methode *vucase*11() 165

12.2.5 Kommentierung der Methode *vucase*12() 166

12.2.6 Kommentierung der Methode *vucase*2() 167

12.2.7 Kommentierung der Methode *vucase*31() 168
12.2.8 Kommentierung der Methode *vucase*32() 169
12.2.9 Kommentierung der Methode *vucase*41() 171
12.2.10 Kommentierung der Methode *vucase*42() 172
12.2.11 Behandlung von Constraints 174
12.2.12 Tests der Methoden 180
12.2.13 Die Klasse SQLInteraction 186
12.3 ToDos 187

13 Entwurf der Sicht-Erneuerung 189

13.1 Idee beschreiben 189
13.1.1 Was ist das Problem? 189
13.1.2 Warum ist das Problem ein Problem? 190
13.1.3 Wie sieht die Lösung aus? 190
13.1.4 Wieso ist die Lösung eine Lösung? 190
13.2 Operationalisierung 191
13.2.1 Aktivitätsdiagramm Sicht-Erneuerung 191
13.2.2 Sequenzdiagramm Sicht-Erneuerung - Logaktualisierung . . . 191
13.2.3 Die neue Klasse ViewUpdateAdmin 191
13.2.4 Der atomare Sicht-Erneuerungsalgorithmus 199
13.2.5 Test des Sicht-Erneuerungsalgorithmus 205
13.2.6 Erstellung der Klasse SQLInteraction 207
13.2.7 Änderunggen in der Klasse SQLInteraction 207

13.2.8 Änderungen in der Klasse ChildQuery 207
13.3 Einpassung in den Prototypen 208
13.4 ToDos 208

14 Entwurf optimierter offener Anfragen 209

14.1 Problembeschreibung 209
14.2 Lösungsansätze 210
14.2.1 Nutzen der VPD-Funktion von Oracle PL/SQL 213
14.2.2 Nutzen des Zensors für geschlossene Anfragen 214
14.3 Optimierbare Sprache erkennen 217
14.4 Implementierung 220
14.4.1 Variablen 220
14.4.2 Anfrageauswertung 221
14.4.3 Klassifikationsinstanz 223
14.4.4 Zensor 223
14.5 Testfälle 226
14.5.1 Offene optimierte Anfragen 227
14.5.2 Fallunterscheidung 229
14.6 Weiterführende Aufgaben 231
14.6.1 Fehlende Funktionalität in einem Zustand 232
14.6.2 Fehlender Zensor 233
14.6.3 Persistente Klassifikationsinstanz 233

15 Entwurf freier Variablen in der Geheimnissprache 235

15.1 Dynamischer Modus 236

15.1.1 Theorembeweiser 236

15.1.2 Freie Variablen in der Geheimnissprache 238

15.1.3 Minimierung der Menge *pot_sec* mittels Prover9 239

15.1.4 Problemdarstellung 239

15.2 Approximation freier Variablen 240

15.2.1 Implementierung 241

15.3 Statischer Modus - Erzeugung von Klassifikationsinstanzen 243

15.4 Optimierung von Klassifikationsinstanzen 244

15.4.1 Fallstudie 245

15.5 Algorithmus: Clino (CLassification INstance Optimizer) 252

15.6 Korrektheitsbeweis für Clino 253

15.7 Einpassung in den Prototypen 260

15.8 Statischer Zensor für Oracle-DB 260

15.9 Implementierung des Clino Algorithmus: 261

15.10 Testen von Clino 265

15.11 Testen von Clino innerhalb des Prototyps 267

15.11.1 Testszene 1 269

15.11.2 Testszene 2 270

15.11.3 Testszene 3 271

15.11.4 Testszene 4 272

15.12ToDos: 274

16 Problemliste für die alten Prototypen 275

16.1 Überblick über Probleme der alten Prototypen 275

16.1.1 Wie wurden diese Fehler behoben? 276

16.2 Testen 277

16.2.1 Prototyp der PG495 279

16.2.2 Sonntag_JCQE 296

16.3 Behebung der Fehler in den Prototypen 314

16.3.1 Optimierter Ablehnungszensor 314

16.3.2 Testergebnisse nach der Änderung 315

17 Neues Datenbankschema 317

17.1 Entity-Relationship-Modell 317

17.2 Datenbanktabellen 319

17.3 Testfälle 322

18 ToDos 329

Literaturverzeichnis 336

Zusammenfassung

Dieser Endbericht befasst sich mit der Arbeit der Projektgruppe 536 zum Thema der optimierten Inferenzkontrolle in relationalen Datenbanken im Sommersemester 2009 und im Wintersemester 2009/2010. Die PG baut auf einem Prototypen für die kontrollierte Anfrageauswertung auf, den die Projektgruppe 495 umgesetzt hat und der anschließend durch die Diplomarbeit von Sebastian Sonntag erweitert wurde. Näheres dazu und die bereits vorhandenen Funktionen siehe [PG408] und [Son08].

Der erste Teil dieses Endberichts enthält die Ausarbeitungen der Projektgruppenteilnehmer der Seminarphase, die in den ersten beiden Vorlesungswochen des Sommersemesters 2009 stattgefunden hat und in der die theoretische Grundlage für die Arbeit dieser Projektgruppe gelegt wurde.

Der zweite Teil des Endberichts befasst sich mit der Umsetzung der Arbeiten dieser Projektgruppe. Dazu gehören die Umsetzung eines Mehrbenutzersystems (Kapitel 11), die Einführung von Sichtänderungen und Sichterneuerungen (Kapitel 12 und 13), die Umsetzung optimierter offener Anfragen (Kapitel 14), die Umsetzung von freien Variablen in der Geheimnissprache (Kapitel 15) sowie der Entwurf eines neuen erweiterten Datenbankschemas (Kapitel 17).

Am Ende der meisten Kapitel gibt es eine Auflistung von, in der Zukunft möglichen, weiteren Verbesserungen des Prototypen bzw. von Problemen, die noch behoben werden müssen. Am Ende dieses Endberichts gibt es darüber hinaus noch einmal eine zusammenfassende Auflistung dieser TODOs.

Teil I

Seminarphase

Kapitel 1

Grundlagen der Kontrollierten Anfragenauswertung

1.1 Warum wird Inferenzkontrolle benötigt?

Um die Frage zu beantworten, warum Inferenzkontrolle überhaupt benötigt wird, schauen wir uns folgendes Szenario an.
Eine Person möchte Informationen einem ausgewählten Personenkreis zugänglich machen, aber gleichzeitig kontrollieren, wer welche Information erhält. Es müssen also die Sicherheitskriterien der *Verfügbarkeit*, sowie der *Vertraulichkeit* erfüllt werden.
Nun stellt sich die Frage, ob wir die gestellten Anforderungen nicht mit den bereits bekannten Mitteln der *Zugangskontrolle* und/oder *Kryptographie* erfüllen können.
Einfach ausgedrückt, kann man Information als die Bedeutung von Daten auffassen. *Zugangskontrolle* bzw. *Kryptographie* machen also die Information zugänglich, indem ausgewählten Personen per Passwort bzw. Schlüssel der Zugriff auf solche Daten erlaubt wird, aus denen die gewünschte Information ersichtlich wird. Ein Verbot wird als Nichtvergabe des Passworts bzw. Schlüssels realisiert. Die Verfügbarkeit kann also sichergestellt werden.
Um die Vertraulichkeit sicherzustellen, muss der Zugriff auf *alle* Daten verweigert werden, aus denen die zu schützende Information ersichtlich wird. Um diese Daten zu erkennen, benötigen wir als weiteres Mittel die *Inferenzkontrolle*.

Um dies zu verdeutlichen, betrachten wir ein einfaches Beispiel. Zu schützen sei die Information „Paul hat ein Einkommen von 50.000€“. Die Informationen „Jeder Manager hat das gleiche Einkommen“, „Paul ist ein Manager“ und „Ein Manager hat ein Einkommen von 50.000€“ seien je in einer Datei gespeichert. Die zu schützende Information ist also als solche in keiner der drei Dateien enthalten und trotzdem ist sie jedem zugänglich, der diese drei Dateien einsehen darf. Die Aufgabe der *Inferenzkontrolle* ist nun zu erkennen, dass diese drei Dateien zusammen ebenfalls die zu schützende Information zugänglich machen.

1.2 Logik-orientierte Informationssysteme

In diesem Kapitel stellen wir *Logik-orientierte Informationssystem* vor, da diese im nächsten Kapitel eine wichtige Rolle einnehmen.
In einem *Logik-orientierte Informationssystem* wird zwischen Daten und Information mit Hilfe der Syntax und Semantik von Formeln unterschieden. Anfragen und Antworten werden im Rahmen der Syntax der zugrunde liegenden Logik ausgedrückt. Ihre Bedeutung und damit die jeweils enthaltene Information wird durch die Semantik festgelegt. Damit wird insbesondere ein operationalisierbarer Begriff von Implikation verfügbar, mit dessen Hilfe ein angeforderter Inhalt, d.h. das entsprechende Datum und die dadurch getragende Information, in Beziehung zu einer zu schützenden Information gesetzt werden kann.
Die reale Welt kann man in so einem System ausdrücken, indem man die *aktuellen Fakten*, aus der die reale Welt besteht, in der *Syntax* ausdrückt. Es werden Sätze in der Logik gebildet. Je nachdem ob die aktuellen Fakten in der realen Welt zutreffen oder nicht, werden diesen Sätzen durch die *Semantik* die Werte „true“ oder „false“ zugewiesen. „true“ wenn sie zutreffen, ansonsten „false“.
Ein Angreifer, der die *aktuellen Fakten* herausfinden möchte, wählt als erstes ein angemessenes System für Schlussfolgerungen aus. Anschließend drückt er sein Wissen KB in diesem System aus. Da nicht alle *aktuellen Fakten* bekannt sind, gibt es mehrere *Instanzen* der Realen Welt. Oder anders ausgedrückt, es gibt mehrere Modelle, die das System erfüllen. Nun versucht der Angreifer durch Anfragen/Beobachtungen y Inferenzen zu ziehen. Nach jeder Anfrage/Beobachtung y wird geprüft, ob es Informationen gibt, die noch nicht im Wissen KB ausgedrückt sind. Falls dies der Fall ist, wird das Wissen KB aktualisiert. Durch dieses Update hat sich dann

eventuell auch die Anzahl der möglichen Instanzen verringert.

1.3 Wie funktioniert Inferenzkontrolle?

Nachdem wir im ersten Kapitel geklärt haben, warum *Inferenzkontrolle* notwendig ist, wollen wir nun klären, wie *Inferenzkontrolle* funktioniert.
Im folgenden bezeichnen wir jemanden, der an Informationen gelangen möchte, die er eigentlich nicht erhalten soll, als „Angreifer“.
Die *Inferenzkontrolle* muss notwendigerweise auf Annahmen über die grundsätzlichen Fähigkeiten und die zu erwartenden Verhaltensweisen des Angreifer aufbauen. Nun stoßen wir auf ein Problem, denn einerseits darf Sicherheit nicht auf Unwissenheit aufbauen und deshalb müssen wir den Angreifer als *denkbar stark* ansehen. Auf der anderen Seite soll die Kontrolle algorithmisch möglichst *effizient* sein. Daraus ergibt sich, dass ein komplexitätsmäßig beschränkter „Verteidigungsalgorithmus“ einen bezüglich Schlussweisen allmächtigen Angreifer abwehren soll. Möglichkeiten dieses Problem zu beheben sind, nach geeigneten *Sonderfällen* zu suchen oder den Anwendungsbereich auf z.B. *algebra-orientierte* oder *logik-orientierte Informationssysteme* einzuschränken. Solche Systeme sind eine geeignete Einschränkung, da so ein operationalisierbarer Begriff von Implikation verfügbar ist, wie wir in Kapitel 1.2 gesehen haben.
Machen wir uns nochmal an einem einfachen Beispiel klar, was *Inferenzkontrolle* leisten muss. Legen wir einmal fest, dass die Gültigkeit der Aussage „B“ die zu schützende Information sei. Ferner sei die Gültigkeit der Aussage „aus A folgt B“ allgemein bekannt. Fragt ein Benutzer nun das Informationssystem, ob die Aussage „A“ gilt, und erhielt er eine bejahende Antwort, so könnte der Benutzer die Gültigkeit der Aussage „B“ als logische Implikation des Allgemeinwissens und der Antwort erschließen. Eine Inferenzkontrolle muss also jeweils die möglichen Implikationen aus dem bisherigen Wissen und der zutreffenden Antwort im Hinblick auf die zu schützende Information untersuchen.

Im Allgemeinen muss eine *Inferenzkontrolle* das Folgende berücksichtigen können:

- „inhaltsabhängiges“ Arbeiten (das System guckt in den Behälter für die Aus-

sage „A“ hinein);

- Darstellung des (vermutlichen) Allgemeinwissen des Benutzers;
- „zustandsabhängiges“ Arbeiten mit Hilfe der Aufzeichnung vorangehender Antworten;
- algorithmische „Lösung von Implikationsproblemen“ (was nicht immer möglich ist).

Falls nun die *Inferenzkontrolle* zu dem Ergebnis kommt, eine zu schützende Information wird gefährdet, so gibt es zwei mögliche Reaktionen. Entweder statt einer korrekten Antwort wird „mum“ ausgegeben oder es wird gelogen. Genauere Informationen folgen in Kapitel 1.4.

1.3.1 Arten von Informationsfluss

Nicht immer ist es einem Angreifer möglich die Information zu erlangen, die er sucht, oder anders formuliert, es gibt nicht nur ein Modell für $KB \cup \{y\}$. Es findet also kein *Kompletter Informationsgewinn* statt. Es gibt aber noch weitere Arten von Informationsfluss. So kann es dem Angreifer gelingen, die gesuchte Information auf einige Möglichkeiten einzuschränken. Falls sich nach einer Beobachtung die Anzahl der Möglichkeiten verringert hat, aber es existieren noch mindestens zwei unterschiedliche ununterscheidbare Interpretationen, die Modell von $KB \cup \{y\}$ sind, so spricht man von *Partiellem Informationsgewinn*. Ist die Anzahl der Möglichkeiten nach der Beobachtung nicht geringer, das heißt jedes Modell von KB ist Modell von $KB \cup \{y\}$, so fand *Kein Informationsgewinn* statt. Das *Gewählte System für Schlussfolgerungen ist ungeeignet*, falls kein Modell von $KB \cup \{y\}$ existiert bzw. $KB \cup \{y\}$ inkonsistent ist.
Da das Updaten des Wissens auf logischen Implikationen basiert, kann es vorkommen, dass diese nicht berechenbar sind. Es kann also zu Fällen kommen, in denen theoretisch ein Informationsgewinn vorliegen würde, dieser sich praktisch aber nicht wiederspiegelt. Aber auch an dieser Stelle sei angemerkt, Unwissenheit darf nicht Teil des Sicherheitskonzepts sein.

1.4 Wie funktioniert die Kontrollierte Anfrageauswertung?

In diesem Kapitel wird dargestellt, wie die *Kontrollierte Anfrageauswertung* das Prinzip der *Inferenzkontrolle* umsetzt.

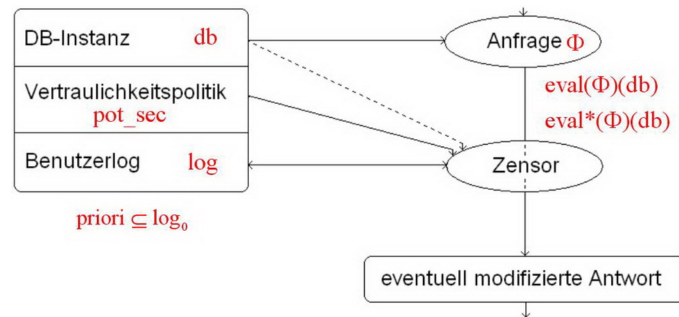


Abbildung 1.1: Die Kontrollierte Anfrageauswertung

In Abbildung 1.1 wird der Ablauf der *Kontrollierte Anfrageauswertung* grob skizziert. Die Antwort einer Anfrage Φ kann entweder die Form $eval(\Phi)(db)$ haben oder $eval^*(\Phi)(db)$. Bei Ersterem ist die Anwort „true“, wenn $eval(\Phi)(db) := db\ model\ of\ \Phi$, sonst „false“. Bei Zweiterem ist die Antwort Φ, wenn $eval^*(\Phi)(db) := db\ model\ of\ \Phi$, sonst $\neg\Phi$. Diese Antwort wird aber noch nicht ausgegeben, sondern dem Zensor übergeben. Ein *instanzabhäniger* Zensor entscheidet dann anhand des Datenbankinstanz *db*, der Vertraulichkeitspolitik *pot_sec* und dem Benutzerlog *log* ob eine Antwort gefährlich ist. Eine Antwort ist gefährlich, wenn sie eine zu schützende Information preisgeben würde. Ein *instanzunabhäniger* Zensor betrachtet die Datenbankinstanz *db* nicht. Sollte der Zensor die Antwort als gefährlich beurteilen, so muß diese modifiziert werden.

Dies kann durch den Zensor selbst geschehen oder durch einen Modifizierer. Es gilt:

```
1 ans := if  censor(db, policy, log, Φ)
2             then  modify(eval*(Φ)(db))
3             else  eval*(Φ)(db)
```

Für die *Kontrollierte Anfrageauswertung* gilt dann:

$$control_eval^{p,a,e}([\Phi_1, \ldots, \Phi_n, \ldots], log_0)(db, policy) = [(ans_1, log_1), \ldots, (ans_n, log_n), \ldots]$$

Dabei steht p für die Sicherheitspolitik. Diese kann entweder aus *möglichen Geheimnissen* (potential secrets) oder aus *Heimlichkeiten* (secrecies) bestehen. Bei *möglichen Geheimnissen* wird eine Menge von Sätzen $\{\Psi_1, \ldots\}$ geschützt, das heißt der Angreifer darf nicht folgern können, wenn einer dieser Sätze „wahr“ ist. *Heimlichkeiten* bedeutet, es wird eine Menge von Sätzen $\{\Psi_1, \neg\Psi_1, \ldots\}$ geschützt. In dem Fall darf die „wahre“ Alternative nicht gefolgert werden können. Das a steht für das Angreiferwissen (awareness) bgzl. der Sicherheitspolitik. So ist a *bekannt*, wenn der Angreifer weiß, welche Sicherheitspolitik angewandt wird, sonst ist a *unbekannt*. Die Methode, die der Zensor einsetzt, um die Vertraulichkeit sicherzustellen, wird durch e dargestellt. So kann e die Werte *verweigern*[1.4.1], *lügen*[1.4.2] oder *kombiniert*[1.4.3] annehmen. Für das Benutzerlog gilt:

$$log_n := \textbf{if } ans_n = mum \textbf{ then } log_{n-1} \textbf{ else } log_{n-1} \cup \{ans_n\}$$

Damit $control_eval^{p,a,e}([\Phi_1, \ldots, \Phi_n, \ldots], log_0)$ die Vertraulichkeit bzgl. einer $policy_1$ sicherstellt, müssen folgende Eigenschaften gelten:

Für jeden Präfix Q‘ jeder (in der Anfragesprache ausdrückbaren)
Anfragefolge $[\Phi_1, \ldots, \Phi_n, \ldots]$,
für jede „tatsächliche“ Instanz des Informationssystems db_1,
für jede Festlegung einer Vertraulichkeitspolitik durch Auflistung der zu
schützenden Aussagen $\Theta \in policy_1$,
für jedes (in der jeweiligen Logik ausdrückbare) Vorwissen, so dass die Instanz, die Vertraulichkeitspolitik und das Vorwissen eine geeignete Vorbedingung $precond^{p,a,e}$ erfüllen,
gibt es für jedes $\Theta \in policy_1$ ein alternatives Paar $(db_2, policy_2)$, das ebenfalls $precond^{p,a,e}$ erfüllt, so dass um die Vertraulichkeit sicherzustellen, folgende Bedingungen gelten:

1. Es wird die gleiche Antwort gegeben, das heißt:
   $control_eval^{p,a,e}(Q', log_0)(db_1, policy_1) = control_eval^{p,a,e}(Q', log_0)(db_2, policy_2)$

2. Obwohl die gleiche Antwort gegeben wird, gibt es eine Instanz, in der das Geheimnis „falsch“ ist, das heißt:
   falls $p = sec$, *d.h.* $\Theta = \{\Phi, \neg\Phi\}: \ \{eval^*(\Phi)(db_1), eval^*(\Phi)(db_2)\} = \{\Phi, \neg\Phi\}$
   falls $p = ps$, *d.h.* $\Theta = \Phi: \ eval^*(\Phi)(db_2) = \neg\Phi$

3. Wenn a=*bekannt*, muss $policy_1$=$policy_2$ gelten.

1.4.1 Wie funktioniert Verweigern

In diesem Kapitel betrachten wir den Verweigerungsansatz. Damit wird eine Möglichkeit dargestellt zu erkennen, ob ein Geheimnis gefährdet ist, und gegebenenfalls einzuschreiten. Da es den Rahmen sprengen würde alle Algorithmen vorzustellen, betrachten wir den Fall $p = ps$, $a = bekannt$, außerdem sei wie oben erwähnt $e = verweigern$. Der Verweigerungszensor prüft in diesem Fall, ob das Hinzufügen der *richtigen oder falschen* Antwort zum Benutzerlog ein Geheimnis implizieren würde.

Definition 1.4.1 ($Zensor^{ps,bekannt,verweigern}$)**:**
$Zensor^{ps,bekannt,verweigern}(db, pot_sec, log, \Phi) :=$
$(\exists\Psi)[\Psi \in pot_sec \ und \ [log \ \cup \ \{\Phi\} \models \Psi \ oder \ log \ \cup \{\neg\Phi\} \models \Psi]]$

Liefert der Zensor nun den Wert „true“, so wird die Antwort zu „*mum*“ modifiziert. In diesem Fall muss das Benutzerlog nicht erweitert werden. Die verneinenden Antwort muss ebenfalls überprüft und gegebenenfalls verweigert werden, denn andernfalls wäre dem Benutzer im bejahenden Fall eine Meta-Inferenz der folgenden Art möglich:
„Das System verweigert die Antwort; dies geschieht nur, falls die zutreffende Antwort eine zu schützende Information impliziert; dies trifft nur für die bejahende Antwort, aber nicht für die verneinende Antwort zu; also ist die zutreffende Antwort bejahend“.

Abschließend macht Theorem 1.4.1 eine Aussage über den Schutz der Vertraulichkeit und Abbildung 1.2 zeigt ein Beispiel für die Kontrollierte Anfrageauswertung mit dem
$Zensor^{ps,bekannt,verweigern}$.

Theorem 1.4.1 ($Biskup/Bonatti, DKE$ 01)**:**
Bei bekannten möglichen Geheimnissen und Verweigerung mit dem $Zensor^{ps,bekannt,verweigern}$ *wird bei kontrollierter Auswertung aller möglichen Anfragen die Vertraulichkeit gewahrt.*

1.4.2 Wie funktioniert Lügen

In Kapitel 1.4.1 haben wir den Verweigerungsansatz kennengelernt. Wie wir gesehen haben, hat dieser jedoch den Nachteil, auch die verneinenden Antwort prüfen zu müssen. Deswegen stellen wir nun den Lügenansatz vor. Wir beschränken uns dabei wieder auf den Fall $p = ps$, $a = bekannt$, jedoch sei diesmal $e = lügen$. Der Lügenzensor prüft, ob das Hinzufügen der richtigen Antwort zum Benutzerlog die *Disjunktion* aller möglichen Geheimnisse impliziert.

Definition 1.4.2 ($Zensor^{ps,bekannt,luegen}$)**:**
$Zensor^{ps,bekannt,luegen}(db, pot_sec, log, \Phi) := log \cup \{eval^*(\Phi)(db)\} \models pot_sec_disj$
wobei $pot_sec_disj := \Psi_1 \vee \ldots \vee \Psi_k$ *für* $pot_sec := \{\Psi_1, \ldots, \Psi_k\}$

Liefert hier nun der Zensor „true“ zurück, so wird die falsche Antwort als richtige ausgegeben und das Benutzerlog dahingehend aktualisiert. An einem kleinen Beispiel soll verdeutlicht werden, warum die Disjunktion aller möglichen Geheimnisse mitgeschützt werden muss. Wir legen fest, dass sowohl die Gültigkeit der Aussage „A“ als auch die Gültigkeit der Aussage „B“ zu schützende Information sei. Ferner sei die Gültigkeit der Aussage „A oder B“ allgemein bekannt. Fragt nun ein Benutzer das Informationssystem nacheinander, erst ob die Aussage „A“ und dann ob die Aussage „B“ gilt, so müsste das System selbst bei tatsächlich zutreffender Bejahung verneinende Antworten zurückliefern, also gegebenenfalls lügen. Dann aber sind das Vorwissen und die zurückgelieferten Antworten, also die sich ergebende Gesamtheit der Aussagen „A oder B“, „¬A“ und „¬B“ inkonsistent. Der Benutzer könnte also das Vorliegen einer Lüge erkennen; formal ergäben sich die zu schützenden Aussagen sogar als logische Implikationen aus der inkonsistenten Gesamtheit (die „alles impliziert“). Auch hier enden wir mit einem Sicherheits-Theorem [1.4.2] und einem Beispiel [1.3].

Theorem 1.4.2 ($Biskup/Bonatti, DKE$ 01)**:**
Bei bekannten möglichen Geheimnissen und Lügen mit dem $Zensor^{ps,bekannt,luegen}$ *wird bei kontrollierter Auswertung aller möglichen Anfragen die Vertraulichkeit gewahrt.*

1.4.3 Wie funktioniert die Kombinierte-Methode

In diesem Kapitel wird nun der Kombinierte-Ansatz vorgestellt. Dieser macht sich sowohl den Verweigerungs- als auch den Lügenansatz zu nutze. Die Variablen p und a seien gesetzt wie in den Kapiteln 1.4.1 und 1.4.2. Zuerst wird geprüft, ob das Hinzufügen der *korrekten und falschen* Antwort zum Benutzerlog ein Geheimnis impliziert, wenn dem so ist, wird *verweigert*. Impliziert *nur* das Hinzufügen der *korrekte* Antwort ein Geheimnis, so wird *gelogen*. Ansonsten kann *korrekt* geantwortet werden.

Definition 1.4.3 ($Zensor^{ps,bekannt,kombiniert}$)**:**

$ans_1 :=$ ***if*** $(\exists\Psi_1)[\Psi_1 \in pot_sec$ *und* $log_{i-1} \cup \{eval^*(\Phi_i)(db)\}$ *impliziert* $\Psi_1]$

  then

    if $(\exists\Psi_2)[\Psi_2 \in pot_sec$ *und* $log_{i-1} \cup \{\neg eval^*(\Phi_i)(db)\}$ *impliziert* $\Psi_2]$

      then *mum*

      else $\neg eval^*(\Phi_i)(db)$

  else $eval^*(\Phi_i)(db)$

wobei $log_i :=$ ***if*** $ans_i = mum$ ***then*** log_{i-1}

        else $log_{i-1} \cup \{ans_i\}$

Theorem 1.4.3 ($Biskup/Bonatti, FoIKS$ 02)**:**
Bei bekannten möglichen Geheimnissen und Nutzen der Kombinierten Methode mit dem $Zensor^{ps,bekannt,kombiniert}$ *wird bei kontrollierter Auswertung aller möglichen Anfragen die Vertraulichkeit gewahrt.*

Die nachfolgenden Beispiele [Abbildungen 1.4,1.5,1.6 und 1.7] sollen verdeutlichen, dass die Kombinierte Methode entweder die gleichen Antworten liefert wie der Verweigerungs- und Lügenansatz [Abbildung 1.4] oder bessere Ergebnisse liefert als einer der beiden Ansätze, ohne jedoch schlechtere als der andere zu liefern [Abbildungen 1.5,1.6 und 1.7]. Der eigentliche Beweis dieser Aussage würde den Rahmen sprengen.

1.5 Statische Inferenzkontrolle

In Kapitel 1.4 wurde die Kontrollierte Anfrageauswertung als mögliche Umsetzung der Inferenzkontrolle vorgestellt. Dies war ein *dynamischer* Ansatz. Im folgenden wollen wir auch einen *statischen* Ansatz, nämlich die *k-Anonymität* [1.5.1] vorstellen. Doch vorher wollen wir kurz die Unterschiede bzw. Vor- und Nachteile des *dynamischen* und *statischen* Ansatzes skizzieren.
Der *dynamische* Ansatz bewertet Anfragen bzgl. ihrer Gefährlichkeit während der Laufzeit des Programms. Er trifft anhand von vorherigen Anfragen, möglichem Vorwissen, der Sicherheitspolitik, der Semantik und dem eigentlichen Programm seine Entscheidung und „blockt“[1.4.1,1.4.2,1.4.3] dann die gefährlichen Antworten. Die Nachteile bei dieser Methode sind der hohe Arbeitsaufwand zur Laufzeit, sowie die Notwendigkeit die vorherigen Antworten zu speichern (Benutzerlog), außerdem können eventuell nicht berechenbare Implikationsprobleme auftreten. Der Vorteil ist, es müssen nur die Anfragen berücksichtigt werden, die tatsächlich gestellt werden. Dies hat weniger „blockings“ als bei einem *statischen* Ansatz zur Folge.
Beim *statische* Ansatz wird eine globale Analyse aller möglichen Anfragen vor dem ersten Programmstart gemacht. Bei diesem Ansatz wird anhand von möglichem Vorwissen, der Sicherheitspolitik, der Semantik und dem eigentlichen Programm, das Programm so abgeändert, dass alle möglichen Sicherheitsverletzungen von vorneherein ausgeschlossen sind. Hier sind die Nachteile die hohe Belastung in der Entwicklung, außerdem müssen alle möglichen Situationen berücksichtigt werden. Auch hier können nicht berechenbare Implikationsprobleme auftreten. Die großen Vorteile sind, es gibt keine zusätzliche Belastung zur Laufzeit und die vorherigen Antworten müssen weder gespeichert werden, noch spielen sie eine sicherheitsrelevante Rolle.

1.5.1 k-Anonymität

Die *k-Anonymität* ist ein *statischer* Ansatz zur Durchführung von Inferenzkontrolle. Sie wird benötigt, um statistische Daten zwecks Analyse zugänglich zu machen, ohne dass die Daten den zugehörenden Personen zugeordnet werden können. Dieses Problem ist nämlich keineswegs so trivial zu lösen, wie man eventuell auf den ersten Blick glauben mag. Ein erster Ansatz ist sicherlich einen Datensatz zu anonymisieren, indem man den zugehörigen Namen einschwärzt oder entfernt. Ein Attribut, das

eine Person eindeutig identifiziert wie in unserem Beispiel 1.8 der Name wollen wir in Zukunft *Identifikator* nennen. In Abbildung 1.8 haben wir den *Identifikator* entfernt. Wie man aber an der rot unterlegten Zeile [Abbildung 1.8] erkennen kann, gibt es eine Menge an Attributen, welche sich auch in anderen Tabellen befinden und damit Verknüpfungen ermöglichen, die eine Person ebenfalls bis zu einem gewissen Genauigkeitsgrad identifizieren. Diese Attribute heißen *Quasi-Identifikatoren* und sorgen dafür, dass ein statistischer Datensatz leider nicht alleine durch das Entfernen des *Identifikators* anonymisiert werden kann. In unserem Beispiel [Abbildung 1.8] sind die Attribute Geburtsdatum, Geschlecht, PLZ und Stand als *Quasi-Identifikatoren* gewählt. Das Attribut Tage fällt in die Rubrik *sonstiges* und ein Attribut, welches Informationen über eine Person darstellt, die nicht mit ihr verbunden werden sollen heißt *sensitives Attribut*. In unserem Beispiel ist dies das Attribut Diagnose.
Das Konzept der *k-Anonymität* definiert den Grad des Schutzes der sensitiven Daten einer Person durch die Größe der Gruppe, in der sie sich in den anonymisierten Daten befindet.

Definition 1.5.1:
k-Anonymität

Eine Tabelle ist k-anonymisiert, wenn es für jeden Tupel in der Tabelle noch mindestens k-1 weitere Tupel gibt, deren Quasi-Identifikatoren den gleichen Wert haben.

Nun gibt es mehrere Methoden die *k-Anonymität* zu erreichen, so kann man die *Quasi-Identifikatoren* generalisieren, indem man diese Attribute vergröbert, so geschehen in Abbildung 1.9. Eine weitere Möglichkeit ist zusätzlich einzelne Tupel zu löschen, meist Ausreißer, um weniger stark zu vergröbern. Dies ist wurde in Abbildung 1.10 gemacht. Man kann auch Werte jeweils der selben Attribute vertauschen, allerdings schränkt man so die Analysemöglichenkeiten ein und es sind nur noch Analysen wie z.B. die Mittelwertberechnung möglich. Wie man sieht, gibt es also mehrere mögliche k-anonymisierte Tabellen [1.9,1.10]. Die „beste“ k-anonymisierte Tabelle ist sicherlich die mit dem geringsten *Informationsverlust*. Um nun Tabellen bzgl. ihres Informationsverlustes vergleichen zu können, benötigt man ein Maß. Dieses Maß ist der *Abstand* vom eigentlichen Attribut, so hat z.B. die PLZ 10** den *Abstand* 2 von 1073. Die *minimale k-Anonymisierung* zu finden ist leider NP-vollständig. Auch hängt der Informationsverlust von den *Quasi-Identifikatoren* ab. Die Wahl der *Quasi-Identifikatoren* hängt wiederum davon ab, zu welchen anderen Tabellen ein möglicher Angreifer Zugang hat. So fällt z.B. das Attribut Tage

in unserem Beispiel in die Rubrik *sonstiges*, wenn nun ein Angreifer die Krankenhausakten einsehen könnte, würde sich dieses Attribut bei Extremwerten sicherlich eignen um eine Person zu identifizieren. Würde man dieses Attribut nun auch als *Quasi-Identifikatoren* deklarieren, müssten die anderen *Quasi-Identifikatoren* stärker generalisiert werden, um in unserem Fall die 2-Anonymität zu gewähren. Dementsprechend wäre der Informationsverlust größer.

1.5.2 l-Diversität

Wie wir auf Abbildung 1.11 erkennen, kann es leider auch in einer k-anonymisierten Tabelle immer noch möglich sein einzelne Tupel zu deanonymisieren. So wird man leicht drauf schließen können, dass die Daten der dritten Zeile zum Abteilungsleiter gehören müssen, und mit Abgleich der Personalliste dieser Abteilung kann nun dem *sensitiven Attribut* Einkommen ein Namen zuordnet werden. Dieser Vorgang kann durch die
l-Diversität verhindert werden.

Definition 1.5.2 (q-Block)**:**
Ein q-Block ist eine Menge von Tupeln, deren Quasi-Identifikatoren die gleichen Werte haben.

Definition 1.5.3 (l-divers)**:**
Ein q-Block ist l-divers, wenn das sensitive Attribut in mindestens l Ausprägungen in dem q-Block vorkommt. Eine Tabelle ist l-divers, wenn ihre q-Blöcke alle l-divers sind.

Atome:	**p, q, r, s1, s2 and s3**
Politik = potential secrets:	*pot_sec* := {**s1, s2, s3**}
Aktuelle Instanz:	*db* := {**¬p, q, ¬r, s1, s2, s3**}
log:	*log*:= {**p ⇒ s1∨ s2, p ∧ q ⇒ s3** }
Anfragefolge:	*Q* := [**q, ¬p**]

Anfrage	korrekt	verweigern	lügen	kombiniert	log
					p ⇒ s1∨ s2 **p ∧ q ⇒ s3**
q	q	q			q
¬p	¬p	mum			

Abbildung 1.2: Beispiel: Verweigern (bekannt,ps)

Atome:	**p, q, r, s1, s2 and s3**
Politik = potential secrets:	*pot_sec* := {**s1, s2, s3**}
Aktuelle Instanz:	*db* := {**p, q, ¬r, s1, s2, s3**}
log:	*log*:= {**p ⇒ s1∨ s2, p ∧ q ⇒ s3** }
Anfragefolge:	*Q* := [**p, q**]

Anfrage	korrekt	verweigern	lügen	kombiniert	log
					p ⇒ s1∨ s2 **p ∧ q ⇒ s3**
p	p		¬p		¬p
q	q		q		q

Abbildung 1.3: Beispiel: Lügen (bekannt,ps)

Atome:	**p**, **q**, **r**, **s1, s2** and **s3**
Politik = potential secrets:	*pot_sec* := {**s1, s2, s3**}
Aktuelle Instanz:	*db* := {**p**, **q**, **¬r**, **s1**, **s2, s3**}
log:	*log*:= {**p** ⇒ **s1**∨ **s2, p** ∧ **q** ⇒ **s3** }
Anfragefolge:	*Q* := [**q, p**]

Anfrage	korrekt	verweigern	lügen	kombiniert	log
					$p \Rightarrow s1 \vee s2$ $p \wedge q \Rightarrow s3$
q	q	q	q	q	verweigern:q lügen: q kombiniert: q
p	p	mum	¬p	¬p	verweigern: - lügen: ¬p kombiniert: ¬p

Abbildung 1.4: Beispiel 1: Kombinierte Methode (bekannt,ps)

Atome:	**p**, **q**, **r**, **s1, s2** and **s3**
Politik = potential secrets:	*pot_sec* := {**s1, s2, s3**}
Aktuelle Instanz:	*db* := {**p**, **q**, **¬r**, **s1**, **s2, s3**}
log:	*log*:= {**p** ⇒ **s1**∨ **s2, p** ∧ **q** ⇒ **s3** }
Anfragefolge:	*Q* := [**p, r**]

Anfrage	korrekt	verweigern	lügen	kombiniert	log
					$p \Rightarrow s1 \vee s2$ $p \wedge q \Rightarrow s3$
p	p	p	¬p	p	verweigern: p lügen: ¬p kombiniert: p
r	¬r	¬r	¬r	¬r	verweigern: ¬r lügen: ¬r kombiniert: ¬r

Abbildung 1.5: Beispiel 2: Kombinierte Methode (bekannt,ps)

Atome:	**p**, **q**, **r**, **s1, s2** and **s3**
Politik = potential secrets:	*pot_sec* := {**s1**, **s2, s3**}
Aktuelle Instanz:	*db* := {¬**p**, **q**, ¬**r**, **s1**, **s2, s3**}
log:	*log*:= {**p** ⇒ **s1**∨ **s2, p** ∧ **q** ⇒ **s3** }
Anfragefolge:	*Q* := [**q**, ¬**p**]

Anfrage	korrekt	verweigern	lügen	kombiniert	log
					p ⇒ s1∨ s2 **p ∧ q ⇒ s3**
q	q	q	q	q	verweigern: q lügen: q kombiniert: q
¬p	¬p	mum	¬p	¬p	verweigern: - lügen: ¬p kombiniert: ¬p

Abbildung 1.6: Beispiel 3: Kombinierte Methode (bekannt,ps)

Atome:	**p**, **q**, **r**, **s1, s2** and **s3**
Politik = potential secrets:	*pot_sec* := {**s1**, **s2, s3**}
Aktuelle Instanz:	*db* := {**p**, **q**, ¬**r**, **s1**, **s2, s3**}
log:	*log*:= {**p** ⇒ **s1**∨ **s2, p** ∧ **q** ⇒ **s3** }
Anfragefolge:	*Q* := [**p**, **q**]

Anfrage	korrekt	verweigern	lügen	kombiniert	log
					p ⇒ s1∨ s2 **p ∧ q ⇒ s3**
p	p	p	¬p	p	verweigern: p lügen: ¬p kombiniert: p
q	q	mum	q	¬q	verweigern: - lügen: q kombiniert: ¬q

Abbildung 1.7: Beispiel 4: Kombinierte Methode (bekannt,ps)

Name	G.Datum	Geschlecht	PLZ	Stand	Tage	Diagnose
-	11.3.59	Männlich	1072	Verheiratet	1	HIV
-	17.3.59	Männlich	1276	Verheiratet	7	Hepatitis
-	1.7.60	Weiblich	1073	Ledig	2	Hepatitis
-	7.9.64	Weiblich	1077	Ledig	0	Beinbruch
-	2.7.69	Männlich	1016	Geschieden	2	Tuberkulose
-	21.9.71	Weiblich	1267	Geschieden	4	Anämie
-	24.12.78	Weiblich	1268	Geschieden	4	HIV

Abbildung 1.8: Beispieltabelle

Name	G.Datum	Geschlecht	PLZ	Stand	Tage	Diagnose
-	[50-59]	Männlich	1***	Verheiratet	1	HIV
-	[50-59]	Männlich	1***	Verheiratet	7	Hepatitis
-	[60-69]	Person	10**	Single	2	Hepatitis
-	[60-69]	Person	10**	Single	0	Beinbruch
-	[60-69]	Person	10**	Single	2	Tuberkulose
-	[70-79]	Weiblich	126*	Geschieden	4	Anämie
-	[70-79]	Weiblich	126*	Geschieden	4	HIV

Abbildung 1.9: Beispiel für Generalisierung

Name	G.Datum	Geschlecht	PLZ	Stand	Tage	Diagnose
-	[55-59]	Männlich	1***	Verheiratet	1	HIV
-	[55-59]	Männlich	1***	Verheiratet	7	Hepatitis
-	[60-64]	Weiblich	107*	Ledig	2	Hepatitis
-	[60-64]	Weiblich	107*	Ledig	0	Beinbruch
-	-	-	-	-	-	-
-	[70-79]	Weiblich	126*	Geschieden	4	Anämie
-	[70-79]	Weiblich	126*	Geschieden	4	HIV

Abbildung 1.10: Beispiel für Unterdrückung

Name	G.Datum	Geschlecht	PLZ	Stand	Abteilung	Einkommen
-	[40-45]	Weiblich	23**	Verheiratet	Logistik	3.000 €
-	[40-45]	Weiblich	23**	Verheiratet	Logistik	3.000 €
-	[40-45]	Weiblich	23**	Verheiratet	Logistik	7.000 €

Abbildung 1.11: Notwendikeit der l-Diversität

Kapitel 2

Sichere Anfragen

2.1 Einführung

2.1.1 Motivation

Im Alltag benutzt man Datenbanken, um Informationen untereinander auszutauschen. Wenn man eine Anfrage an eine Datenbank gestellt hat, wünscht man sich, von der Datenbank eine Menge von endlichen Antworten zu bekommen. Aber die Datenbank kann manche Anfragen nicht behandlen, weil sie eine Menge von unendlichen Antworten besitzen.

Beispiel 2.1.1:
Wer ist kein Lehrer?

Falls die Domäne für die Leute unendlich ist, ist die Menge von Antworten auch unendlich. Eine solche Anfrage heißt unsichere Anfrage, falls sie die Datenbank in endlicher Zeit nicht behandeln kann, deshalb sollen wir möglichst solche Anfragen vermeiden und garantieren, dass alle an die Datenbank gestellten Anfragen sicher sind.

2.1.2 Unterschiedliche Anfragesprachen

In der Realität gibt es unterschiedliche Anfragesprachen. Eine grundlegende Anfragesprache ist konjunktive Anfragesprache, die starke Ausdruckskraft besitzt. Mit Hilfe von Konjunktion und Existenzquantor lassen sich viele Anfragen ausdrücken. Obwohl die konjunktive Anfragesprache starke Ausdruckskraft besitzt, kann sie noch manche Anfragen nicht ausdrücken:

Beispiel 2.1.2:
Welches sind die Hitchcock-Filme, in denen Hitchcock nicht mitspielt?

Beispiel 2.1.3:
Liste die Filme auf, für die alle Darsteller des Films schon unter der Regie Hitchcocks gespielt haben.

2.1.3 Hinzufügung der Negation

Damit solche Anfragen ausgedrückt werden können, fügen wir der konjunktiven Anfragesprache die Negation hinzu. Dadurch bekommen wir eine Anfragesprache mit stärkerer Ausdrucksfähigkeit, nämlich der relationale Kalkül, mit dem kann man die im Beispiel 2.1.2 und Beispiel 2.1.3 erwähnten Anfragen ausdrücken.

2.1.4 Folge von der Negation

Wenn die Negation unbeschränkt benutzt wird, entsteht auch ein Nachteil, nämlich die unsicheren Anfragen, die von der Domäne abhängig sind und unendliche Antworten haben. Z.B. $\neg unterrichtet(x, Englisch)$? Die Antwort dafür enthält alle Leute, die nicht in der Relation Unterrichtet vorkommen und alle Leute, die in der Relation unterrichtet vorkommen aber kein Englisch unterrichten. Weil die Domäne für Personennamen unendlich ist, und die in der Relation vorkommenden Menge von Personennamen endlich ist, gibt es unendlich viele Namen, die nicht in der Relation Unterrichtet vorkommen. Damit ist die gesamte Antwort unendlich. In der Realität terminiert eine unsichere Anfrage nicht und kann von der Datenbank nicht behandelt werden. Deshalb muss man unsichere Anfragen vermeiden. Es gibt zwei Möglichkeiten, entweder eine semantische Einschränkung oder eine syntaktische Ein-

schränkung der Anfragesprache hinzuzufügen, damit alle Anfragen sichere Anfragen sind.

2.2 Semantische Einschrünkung

2.2.1 Relationaler Kalkül

Der relationale Kalkül erweitert den konjunktivem Kalkül. Die Unterschiede sind die Negation, die Disjunktion und der All-Quantor. Disjunktion und All-Quantor können als Folge der Einführung von Negationen gesehen werden: $(\Phi \vee \Psi) \equiv \neg(\neg\Phi \wedge \neg\Psi)$ und $\forall x\, \Phi(x) \equiv \neg\exists x\, \neg\Phi(x)$.

Definition 2.2.1 (Relationaler Kalkül)**:**
Bei einem gegebenen Schema $\boldsymbol{R}$ *ist die formale Definition des relationalem Kalküls wie folgend definiert:*

- *Konstanten und Variablen in* $\boldsymbol{R}$ *sind Terme*
- $e = e^{'}$ *und* $e \neq e^{'}$ *sowie alle Atome über* $\boldsymbol{R}$ *sind grundlegende Formeln (wobei* e *und* e' *Terme sind)*
- $(\Phi \vee \Psi)$*, dabei sind* Φ *und* Ψ *Formeln des relationalen Kalküls*
- $(\Phi \wedge \Psi)$*, dabei sind* Φ *und* Ψ *Formeln des relationalen Kalküls*
- $\neg\Phi$*, dabei ist* Φ *Formel des relationalen Kalküls*
- $\exists x\, \Phi$*, dabei ist* Φ *Formel des relationalen Kalküls*
- $\forall x\, \Phi$*, dabei ist* Φ *Formel des relationalen Kalküls*

Angenommen es gibt eine Relation Movies, mit den drei Attributen Titel, Regie und Darsteller. Dann lassen sich die oben genannten Beispiele im relationalen Kalkül so formulieren:

Beispiel 2.2.1:
Welches sind die Hitchcock-Filme, in denen Hitchcock nicht mitspielt?
$\{x_t \mid \exists x_d\, Movies(x_t, Hitchcock, x_d) \wedge \neg Movies(x_t, Hitchcock, Hitchcock)\}$

Beispiel 2.2.2:
Liste die Filme auf, für die alle Darsteller des Films schon unter der Regie Hitchcocks gespielt haben.

```
1 {x_t | ∃x_r, x_d Movies(x_t, x_r, x_d) ∧ ∀y_d(∃y_r Movies(x_t, y_r, y_d)
2                                  → ∃z_t Movies(z_t, Hitchcock, y_d))}
```

In der Anfrage 2.2.1 liefert die erste Konjunktion alle Filme zurück, die von Hitchcock Regie geführt wurden. Die zweite Konjunktion liefert alle Filme zurück, die nicht unter der Regie von Hitchcock und in denen auch Hitchcock nicht mitspielt. In der Anfrage 2.2.2 liefert die erste Konjunktion alle aktuellen Filme zurück, die in der Relation Movies vorkommt. Die zweite Konjunktion liefert alle Filme zurück, deren Darsteller schon unter Regie Hitchcock gespielt haben.

2.2.2 Unsichere Anfragen

Jetzt betrachten wir einige konkrete unsichere Anfragen, um das Zusammenhang zwischen unsicheren Anfragen und Domäne, die benutzt wird, um die Antwort auf eine Anfrage zu berechnen, besser zu verstehen. Daran werden wir sehen, worin die wesentlichen Probleme liegen. Und danach können wir erst dann sichere Anfragen und domänenunabhängige Anfragen diskutieren.

Beispiel 2.2.3:
$\{(x) \mid \neg\, Movies(Cries\, and\, Whispers, Bergman, x)\}$

Die Datenbank wird für diese Anfrage alle Tupel (a) zurückliefern, für die das Tupel $(Cries\, and\, Whispers, Bergman, a)$ nicht in der Relation Movies vorkommt. Die Antwort ist von der unendlichen Domäne Dom abhängig und deshalb auch unendlich.

Beispiel 2.2.4:
$\{(x, y) \mid Movies(Cries\, and\, Whispers, Bergman, x) \vee Movies(y, Bergman, Ullman)\}$

Die Datenbank wird für diese Anfrage die folgenden Tupeln (a, b) zurückliefern: wenn

ein Tupel $(Cries\,and\,Whispers, Bergman, a)$ in der Relation Movies vorkommt, dann alle (a, b) mit $b \in Dom$ oder wenn ein Tupel (b, $Bergman$, $Ullman$) in der Relation Movies vorkommt, dann alle (a, b) mit $a \in Dom$. Deshalb ist diese Anfrage auch von der unendlichen Domäne Dom abhängig und die Antwort ist unendlich.

Beispiel 2.2.5:
$\{(x) \mid \forall y, R(x, y)\}$

Für diese Anfrage soll es eigentlich eine endliche Antwortmenge zurückgegeben werden, weil sie eine Teilmenge von $\Pi_1(R)$, der Projektion auf die erste Spalte von R, ist. Für die endliche Domäne Dom= $(1, 2, 3)$ und die Instanz $\langle(1, 1), (1, 3), (1, 2)\rangle$, ist die Antwort $\{(1)\}$. Aber in den meisten Fällen wird es eine unendliche Domäne Dom gegeben und die Aufzählung von y hängt von der Dom ab. Deshalb terminiert die Aufzählung von y nicht und man kann keine Antwort festlegen, und die Antwortmenge ist immer undefiniert. Daraus können wir schließen, dass die Anfrage auch von der Domäne Dom abhängig ist.

2.2.3 Relativierte Interpretation

Wir haben gesehen, dass bei unterschiedlicher Domänen für eine Anfrage unterschiedliche Antworten zurückgeliefert werden können. Mithilfe dieser Eigenschaft können wir jetzt die Definition der relativierten Interpretation einführen. Damit lassen sich Anfragen bzgl. unterschiedlicher Domänen, am wichtigsten bzgl. einer endlichen Domäne auswerten. Der Vorteil von relativierter Interpretation liegt darin, dass bei der Behandlung einer durch einen Quantor gebundenen Variable nicht die originale unendliche Domäne, sondern eine spezielle endliche Domäne benutzt werden darf. Dann wird die Aufzählung der Kandidaten für die durch den Quantor gebundenen Variable in einer endlichen Domäne möglich.

Definition 2.2.2 (Relativierte Instanz)**:**
Eine relativierte Instanz über dem Schema $\boldsymbol{R}$ *ist ein Paar* (d, I), *wobei* I *eine Instanz über* $\boldsymbol{R}$ *und* $adom(I) \subseteq d \subseteq Dom$ *ist.* $adom(I)$ *ist die Menge der Konstanten, die in der Instanz I vorkommen und Dom ist die übliche Domäne.*

Definition 2.2.3 (Relativierte Interpretation)**:**
Eine Kalkül Formel φ *ist interpretierbar über* (d, I), *wenn* $adom(\varphi) \subseteq d$ *und v:* $frei(\varphi) \rightarrow d$ *eine Funktion zur Belegung der freien Variablen von* φ, *deren Werte-*

bereich d ist. Man sagt, I erfüllt φ für v relativ zu d, geschrieben $I \models_d \varphi[v]$, wenn folgende Bedingungen erfüllt sind:

- $\varphi \equiv R(u)$ *ist ein Atom und* $v(u) \in I(R)$
- $\varphi \equiv (s = s')$ *ist ein Gleichheitsterm und* $v(s) = v(s')$
- $\varphi \equiv (\psi \wedge \xi)$, *ist eine Konjunktion und* $I \models_d \psi[v\ |_{frei(\psi)}]$ *und* $I \models_d \xi[v\ |_{frei(\xi)}]$
- $\varphi \equiv (\psi \vee \xi)$, *ist eine Disjunktion und* $I \models_d \psi[v\ |_{frei(\psi)}]$ *oder* $I \models_d \xi[v\ |_{frei(\xi)}]$
- $\varphi \equiv \neg\psi$, *ist eine Negation und* $I \not\models_d \psi[v]$
- $\varphi \equiv \exists x\, \psi$, *und für ein bliebiges* $c \in d$, $I \models_d \psi[v \cup \{x/c\}]$
- $\varphi \equiv \forall x\, \psi$, *und für alle* $c \in d$, $I \models_d \psi[v \cup \{x/c\}]$

Die formale Definition von relativierter Interpretation lautet:
Seien $\boldsymbol{R}$ ein Schema und $q = \{e_1, ..., e_n \mid \varphi\}$ eine Anfrage über $\boldsymbol{R}$, (d, I) ist eine relativierte Instanz über $\boldsymbol{R}$, die relativierte Interpretation von q auf I relativ zu d ist: $q_d(I) = \{v(\langle e_1, ..., e_n\rangle) \mid I \models_d \varphi[v]$ und v: $frei(\varphi) \rightarrow d$ und $frei(\varphi)$ ist die Menge von freien Variablen, die in φ vorkommen.$\}$.

2.2.4 Aktive Domäne Semantik

Definition 2.2.4 (Aktive Domäne Interpretation)**:**
Für eine Kalkül Anfrage q und eine gegebene Instanz I ist die Aktive Domäne Interpretation $q_{adom}(I)$ *von q auf I, gegeben durch* $q_{adom(q,I)}(I)$, *wobei* $adom(q, I) = adom(q) \cup adom(I)$. *Die Menge der Interpretationen von aktive Domänen Anfragen ist* $CALC_{adom} = \{q_{adom} \mid$ *q ist relationale Anfrage und adom ist die aktive Domäne* $adom(q, I)\}$.

Die aktive Domäne Semantik ist eine wichtige Semantik für Kalkül Anfragen und stammt aus der relativierten Interpretation. Wenn eine Anfrage bzgl. der aktiven Domäne verarbeitet wird, wird immer eine endliche Menge von Antworten zurückgeliefert, weil die Belegung der durch einen Quantor gebundenen Variablen oder freier Variablen auf die aktive Domäne beschrängkt wird.

Betrachten wir wiederum das Beispiel 2.2.4 unter der Aktiven Domäne: $\{(x, y) \mid Movies(Cries\,and\,Whispers, Bergman, x) \vee Movies(y,\ Bergman, Ullman)\}$. Die Antwort dafür ist eine endliche Menge: {({Darsteller, die in Cries and Whispers mitspielen}×adom(q,I))} ∪ {(adom(q,I)×{Filme von Bergman, in denen Ullman spielt})}.

2.2.5 Domänenunabhängige Anfragen

Die Aktive Domäne Semantik ist schon viel besser als die vorherige Semantik. Aber es gibt darin noch eine Schwachstelle, es behandelt Anfragen bzgl. der aktiven Domäne, die abhängig von der aktuellen Datenbank Instanz ist. Aber Tatsache ist, diese Informationen sind für Benutzer unsichtbar und ein Benutzer kann die aktive Domäne nicht festlegen. Eine bessere Definition ist die domänenunabhängige Anfrage, die bzgl. beliebiger Domänen immer die gleiche Antwort haben.

Definition 2.2.5 (Domänenunabhängige Anfragen)**:**
Eine Kalkül Anfrage q ist domänenunabhängig, falls für jede gegebene Instanz I, und je zwei Domänen $d, d' \in Dom$, $q_d(I) = q'_d(I)$.

Falls q domänenunabhängig ist, dann bezeichnet man für jedes beliebige d, die Interpretation $q_d(I)$ einfach mit $q(I)$, weil alle Domänen für die Anfrage gleichwertig sind.

Die Menge der Interpretationen von domänenunabhängigen Anfragen ist $CALC_{di}$ und $CALC_{di} = \{q_{Dom} \mid$ q ist domänenunabhängige relationale Kalkül Anfrage, Dom ist die übliche Domäne Dom$\}$.

Beispiel 2.2.6:
Domänenunabhängige Anfrage:

```
1 {x_t | ∃x_r, x_d Movies(x_t, x_r, x_d) ∧ ∀y_d(∃y_r Movies(x_t, y_r, y_d)
2                                     → ∃z_t Movies(z_t, Hitchcock, y_d))}
```

$$\{x_t \mid \exists x_r, x_d\, Movies(x_t, x_r, x_d) \wedge \forall y_d(\exists y_r\, Movies(x_t, y_r, y_d) \rightarrow \exists z_t\, Movies(z_t,\ Hitchcock,\ y_d))\}$$

Beispiel 2.2.7:
Nicht domänenunabhängige Anfrage:

```
1 {x_t | ∀y_d(∃y_r Movies(x_t, y_r, y_d) → ∃z_t Movies(z_t, Hitchcock, y_d))}
```

Der Unterschied zwischen diesen beiden Anfragen ist die erste Konjunktion, die

gewährleistet, dass die Belegung der Variable x_t unter den aktuellen Titel beschränkt wird, die in der Relation Movies vorkommen. Mit dieser Konjunktion wird für die erste Anfrage endliche Antworten zurückgeliefert. Ohne dieser Konjunktion wird für die zweite Anfrage unendliche Antworten zurückgeliefert, die besteht aus zwei Teilen: alle in der Relation Movies vorkommenden Titel, die die Anfrage erfüllt, und alle potentielle Titel, die nicht im $\Pi_{titel}(Movies)$ vorkommen.

Lemma 2.2.1 ($CALC_{di} \sqsubseteq CALC_{adom}$)**:**
$CALC_{di} \sqsubseteq CALC_{adom}$

Alle Interpretation domänenunabhängiger Anfragen sind ebenfalls aktive Domäne Interpretation. Falls eine Anfrage q domänenunabhängig ist, dann kann man die Anfrage mit einer beliebigen Domäne d ausrechnen, d.h. bei der Ausrechnung der Anfrage kann man statt der natürlichen Interpretation(q_{Dom}) die aktive Domäne Interpretation(q_{adom}) benutzen. Deshalb sind alle domänenunabhängigen Interpretation ebenfalls Aktive Domäne Interpretation.

Theorem 2.2.1 (Äquivalenztheorem)**:**
$CALC_{di} \equiv CALC_{adom} \equiv relationale\ Algebra$

Der domänenunabhängige relationale Kalkül ($CALC_{di}$), der Kalkül im Rahmen einer Aktiven Domäne Semantik ($CALC_{adom}$) und die relationale Algebra haben gleiche Ausdruckskraft.

2.3 Syntaktische Einschränkung

2.3.1 Motivation

Im letzten Abschnitt haben wir gesehen, dass domänenunabhängige Anfragen immer eine endliche Menge von Antworten besitzen. Aber im Rahmen von semantischen Einschränkungen ist es schwierig festzulegen, ob eine gegebene Anfrage domänenunabhängig ist oder nicht. Deshalb werden wir in diesem Abschnitt zur Anfragesprache eine syntaktische Einschränkung(Safe Range) hinzufügen. Diese garantiert, dass die Anfrage domänenunabhängig ist.

2.3.2 SRNF (Safe Range Normal Form)

Um die syntaktische Einschränkung (Safe Range) zu definieren, muss die Anfrage in die SRNF umgeformt werden.

Definition 2.3.1 (SRNF)**:**
Eine Anfrage ist in SRNF, falls die folgende Bedingungen erfüllt sind:

1. *Keine Variable kommt sowohl frei als auch gebunden vor.*
2. *Es kommen keine All-Quantoren vor.*
3. *Jede Implikation $\varphi \rightarrow \psi$ muss durch $\neg\varphi \vee \psi$ und jede Äquvivalenz $\varphi \leftrightarrow \psi$ durch $(\varphi \wedge \psi) \vee$ $(\neg\varphi \wedge \neg\psi)$ ersetzt werden.*
4. *Jede Negation muss direkt vor einem Atom oder vor einem Existenz-Quantor stehen.*

Um eine Formel in die SRNF umzuformulieren, benutzt man die äquivalenz erhaltende Umschreibungsregeln:

1. Variablen Substitution, damit keine Variable sowohl frei als auch gebunden vorkommt.
2. Entfernung von All-Quantoren: jede Teilformel $\forall x\, \varphi$ wird durch $\neg\exists x\, \neg\varphi$ ersetzt.
3. Entfernung von Implikationen: jede Teilformel $\varphi \rightarrow \psi$ wird durch $\neg\varphi \vee \psi$ ersetzt und jede Teilformel $\varphi \leftrightarrow \psi$ wird durch $(\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi)$ ersetzt.
4. Ersetzung der Negation:
   - $\neg\neg\varphi$ wird durch φ ersetzt.
   - $\neg(\varphi_1 \vee \ldots \vee \varphi_n)$ wird durch $(\neg\varphi_1 \wedge ... \wedge \neg\varphi_n)$ ersetzt.
   - $\neg(\varphi_1 \wedge \ldots \wedge \varphi_n)$ wird durch $(\neg\varphi_1 \vee ... \vee \neg\varphi_n)$ ersetzt.
5. Schließlich lassen sich $\wedge$, $\vee$ und $\exists$ flatten, d.h. es gibt keine 'und', deren Kind auch ein 'und' ist, ähnlich für 'oder' und 'Existenzquantor'.

Nach der Anwendung dieser äquivalenz erhaltenden Umschreibungsregeln auf φ, erhält man die SRNF(φ). Eine Formel φ ist in SRNF, falls SRNF(φ)= φ.

Beispiel 2.3.1:

```
1 {x_t | ∃x_r, x_d Movies(x_t, x_r, x_d)∧
2                  ∀y_d(∃y_r Movies(x_t, y_r, y_d)
3                              → ∃z_t Movies(z_t, Hitchcock, y_d))}
4 ⇒
5 {x_t | ∃x_r, x_d Movies(x_t, x_r, x_d)∧
6                  ¬∃y_d(∃y_r Movies(x_t, y_r, y_d)∧
7                              ¬∃z_t Movies(z_t, Hitchcock, y_d))}
```

Durch die Anwendung der äquivalenz erhaltenden Umschreibungsregeln lässt sich die Anfrage in SRNF umformen.

2.3.3 Algorithmus Range Restriction(rr)

Eingabe : eine Kalkül Formel φ in SRNF
Ausgabe: eine Teilmenge der freien Variablen von φ oder false

```
1  begin
2  case of φ:
3         R(e_1, ..., e_n): rr(φ) = die Menge der Variablen in {e_1, ..., e_n};
4         φ_1 ∧ φ_2 : rr(φ) = rr(φ_1) ∪ rr(φ_2);
5         φ_1 ∨ φ_2 : rr(φ) = rr(φ_1) ∩ rr(φ_2);
6         ¬φ_1      : if rr(φ_1) ≠ false
7                      then rr(φ) = ∅
8                      else return false
9         ∃xφ_1     : if x ∈ rr(φ_1)
10                     then rr(φ) = rr(φ_1) − x
11                     else return false
12 end case
13 end
```

Falls für die Anfrage φ eine Menge von freien Variablen zurückgeliefert wird und die zurückgelieferte Menge $rr(\varphi) = frei(\varphi)$, so ist diese Anfrage safe range.

Falls für die Anfrage φ eine Menge von freien Variablen zurückgeliefert wird und die zurückgelieferte Menge $rr(\varphi) \subset frei(\varphi)$, so ist diese Anfrage nicht safe range, weil es mindestens eine freie Variable gibt, die nicht safe range ist.

Falls false zurückgeliefert wird, das bedeutet, es gibt in der Anfrage φ mindestens eine gebundene Variable, deren Range nicht eingeschränkt ist.

Mit Hilfe dieses Algorithmus kann man überprüfen, ob eine gegebene Anfrage safe range ist. Die Menge der natürlichen Interpretationen von safe range Anfragen wird mit $CALC_{sr}$ bezeichnet.

Beispiel 2.3.2:

```
1 {x_t | ∃x_r, x_d Movies(x_t, x_r, x_d)∧
2                  ¬∃y_d(∃y_r Movies(x_t, y_r, y_d)∧
3                             ¬∃z_t Movies(z_t, Hitchcock, y_d))}
```

Diese Anfrage ist safe range, weil $rr(\varphi) = frei(\varphi) = \{x_t\}$ *ist.*

$\{x_t \mid \neg\exists y_d(\exists y_r\, Movies(x_t, y_r, y_d) \wedge \neg\exists z_t\, Movies(z_t,\, Hitchcock,\, y_d))\}$

Diese Anfrage ist nicht safe range, weil $rr(\varphi) = \emptyset$ *und* $frei(\varphi) = \{x_t\}$ *ist.*

Theorem 2.3.1 ($CALC_{sr} \equiv$ relationale Algebra)**:**
$CALC_{sr} \equiv$ *relationale Algebra, gleichzeitig ist jede safe range Anfrage domänenunabhängig.*

Mit Theorem 2.2.1 (Äquivalenztheorem) und Theorem 2.3.1 $CALC_{sr} \equiv$ *relationale Algebra können wir folgern, dass der domänenunabhängige Kalkül(*$CALC_{di}$*), der Kalkül im Rahmen einer Aktiven Domäne Semantik(*$CALC_{adom}$*), die relationale Algebra und die Safe-Range Anfrage* $CALC_{sr}$ *die gleiche Ausdruckskraft haben.*

Kapitel 3

Implikation

3.1 Grundlagen

Dieses Thema hat [BB07] als Grundlage, vor allem 3.2, 3.3 und 3.4. 3.1 fundiert auf [Bis95], [BKI08a], [Jun02] und [Sch09]. Zur Einleitung in das Thema zunächst einmal ein paar Definitionen und Grundlagen, auf denen die Lemma und Theoreme aufbauen. Hier soll aber nur ein kurzer Überblick über die verwendeten Begrifflichkeiten gegeben werden.

3.1.1 Prädikatenlogik

Die Prädikatenlogik besteht aus Funktionen, Relationen, Variablen, Konstanten und Termen. Terme sind zum einen die Variablen und Konstanten, zum anderen werden sie mit Hilfe von Funktionssymbolen und anderen Termen neugebildet:

- Jede Variable und jedes Konstantensymbol ist ein Term
- f k-stelliges Funktionssymbol, $t_1,\ldots,t_k$ Terme, dann ist f($t_1,\ldots,t_k$) auch ein Term

Ein weiterer Begriff in der Prädikatenlogik sind **Formeln**, die aus den folgenden Elementen zusammengesetzt werden:

- Prädikate sind (atomare) Formeln
- Gleichung zwischen zwei Termen ist eine Formel
- Konjunktion $F_1 \wedge F_2$, Disjunktion $F_1 \vee F_2$, Negation ¬F mit den Formeln F, F_1 und F_2
- Allquantor ∀x F („Für alle x gilt F“, x Variable)
- Existenzquantor ∃x F („Es gibt mind. ein x für das F gilt“)

Es werden noch folgende Unterscheidungen in Bezug auf die Variablen und Formeln gemacht, die im Folgenden des öfteren auftauchen:

- **gebundene Variablen**: Eine Variable x im Wirkungsbereich von ∀x oder ∃x, ansonsten heißt sie frei
- **freie Variablen**: Alle Variablen, die in einer Formel mindestens einmal frei vorkommen
- **offene Formeln**: Formeln, die freie Variablen enthalten
- **geschlossene Formeln**: kein freies Vorkommen von Variablen

Die folgenden Begriffe tauchen immer wieder im Verlauf des Textes auf und sollen deswegen auch noch einmal kurz erklärt werden:
Eine **Interpretation** I besteht aus einem Universum, Konstanten, Funktionen und Relationen, sowie einer Belegung, die allen Variablen einer Formel einen Wert des Universums zuordnet.
Unter der **Gültigkeit** einer Formel für eine Variablenbelegung versteht man, dass sie für diese Belegung wahr ist.
Mit der **Allgemeingültigkeit** einer Formel bezeichnet man wiederum den Fall, dass die Formel in jeder nicht-leeren Domäne gültig ist.
Das **Allgemeingültigkeitsproblem** beschäftigt sich mit der Frage, ob eine Formel φ in jeder Interpretation I gültig ist.
Die **Erfüllbarkeit** einer Formel bedeutet, dass es eine Interpretation gibt, für die der Wahrheitswert der Formel wahr ist.

3.1.2 Logische Implikation

Zunächst einmal die allgemeine Definition, was eine Implikation eigentlich ist. Unter dem Begriff der materialen Implikation versteht man folgende Schreibweise:

$$F_1 \Rightarrow F_2$$

Natürlichsprachlich sagt man: „Wenn F_1 gilt, dann gilt auch F_2“
In der Aussagenlogik ist dieser Ausdruck äquivalent zu der Formel: $\neg\, F_1 \vee F_2$.
Diese Beziehung wird auch im weiteren eine Rolle spielen und muss im Rahmen der kontrollierten Anfrageauswertung besonders beachtet werden, um zu vermeiden, dass durch Ausnutzung dieser Beziehung Geheimnisse verraten werden könnten.

Im folgenden wird im Zusammenhang mit der Implikation diese Schreibweise verwendet:

Definition 3.1.1:
Eine Formel(-menge) Σ impliziert logisch eine Formel(-menge) Φ,

$$\Sigma \models \Phi \textit{ gdw. } Mod(\Sigma) \subset Mod\ (\Phi)$$

mit $Mod(\Sigma) := \{I \mid \Sigma \textit{ ist gültig in der Interpretation } I\}$

Diese Relation zwischen zwei Formel(-mengen) wird klassisch-logische Folgerung genannt.
Ist der erste Teil der Folgerung eine Interpretation I, also hat man $I \models \Phi$, dann wird sie **Erfüllungsrelation** genannt. Die Folgerung ist genau dann erfüllt, wenn die Formel Φ in der Interpretation I wahr ist.
Den formalen Zusammenhang zwischen der Implikation und der klassisch-logischen Folgerung für aussagenlogische Formeln und für geschlossene prädikatenlogische Formeln kann man mit Hilfe des folgenden Theorems darstellen [BKI08a]:
Theorem 3.1.1 (Deduktionstheorem)**:**

$$\Sigma \models \Phi \textit{ gdw. } \models \Sigma \Rightarrow \Phi$$

Im folgenden Text wird allerdings nur der Begriff Implikation verwendet.

3.1.3 Verschiedene Implikationen

Wir unterscheiden im folgenden zwischen drei Arten von Implikationen, für die später gezeigt wird, dass sie unter bestimmten Voraussetzungen als äquivalent im Bezug auf die Entscheidbarkeit betrachtet werden können.

Definition 3.1.2:
Für jede Aussagenmenge Σ *und Aussage* φ*:*
Klassische Implikation
$\Sigma \models_{gen} \varphi$ *gdw. für alle Interpretationen I:*
$I \models \Sigma$ *impliziert* $I \models \varphi$
Endliche Implikation
$\Sigma \models_{fin} \varphi$ *gdw. für alle endlichen Interpretationen I:*
$I \models \Sigma$ *impliziert* $I \models \varphi$
DB-Implikation
$\Sigma \models_{DB} \varphi$ *gdw. für alle DB-Interpretationen I:*
$I \models \Sigma$ *impliziert* $I \models \varphi$

Der Unterschied zwischen diesen drei Implikationen liegt in den Interpretationen, die diese verwenden. Die klassische Implikation macht keine Einschränkungen bezüglich der Interpretation, bei der endlichen Implikation beschränkt sich die Aussage lediglich auf endliche Interpretationen. Der Begriff der DB-Interpretation wird in Kapitel3.4 Relationales Submodell noch genauer definiert.

3.1.4 Implikationsproblem

Beim Implikationsproblem geht es um die Frage, ob für eine Menge von Aussagen Σ und eine Aussage φ

$$\Sigma \models \varphi$$

gilt.
Für die kontrollierte Anfrageauswertung ergibt sich daraus die Aufgabe, dass auch

die Daten geschützt werden müssen, aus denen man Informationen über vertrauliche Daten folgern kann. Bei jeder Anfrage muss der Kontrollmechanismus somit das Wissen des Nutzers (wobei das unter gewissen Umständen auch nicht immer benötigt wird), das anfragenabhängige Wissen und die vertraulichen Daten im Bezug auf mögliche Implikationsbeziehungen betrachten und abhängig davon die Anfrage beantworten.
Nun ein einfaches Beispiel, an dem deutlich werden soll, inwiefern die Implikation ein Problem werden kann.

Beispiel 3.1.1:

Es gilt: hatGrippe(Peter) ⇒ istKrank(Peter)
Die Information istKrank(Peter) soll geschützt werden.
Dann darf das System in dem Fall keine Auskunft darüber geben, ob hatGrippe(Peter) gilt, weil daraus die geschützte Information gefolgert werden könnte.

3.1.5 Entscheidbarkeit

Die Entscheidbarkeit spielt im weiteren eine wichtige Rolle, deswegen noch mal kurz die Definition:

Definition 3.1.3:
Für beliebige Menge M gilt:

- *M ist **entscheidbar** gdw. es gibt eine Algorithmus, der für jedes x angibt, ob $x \in M$ oder $x \notin M$*
- *M ist **unentscheidbar** gdw. M ist nicht entscheidbar*
- *M ist **semi-entscheidbar** gdw. es gibt einen Algorithmus, der für jedes x aus der Menge M angibt, dass $x \in M$ gilt*

Als Entscheidungsproblem bezeichnet man dann das Problem ein effektives Verfahren für eine bestimmte Eigenschaft dieser Menge zu finden.

Entscheidbarkeit vom Implikationsproblem

Im Allgemeinen ist das Implikationsproblem unentscheidbar. Nur wenn man passend *eingeschränkte Logiken* definiert, kann man das Implikationsproblem entscheidbar machen.
Auch die Prädikatenlogik ist unentscheidbar, wobei damit gemeint ist, dass die Allgemeingültigkeit von beliebigen prädikatenlogischen Aussagen unentscheidbar ist.
In den folgenden Kapiteln wird ein relationales Submodell mit Hilfe eines Fragmentes der Prädikatenlogik konstruiert, für das das Implikationsproblem entscheidbar ist. Dazu kann die Bernays-Schönfinkel-Klasse benutzt werden, die ein solches Fragment der Prädikatenlogik liefert.

3.2 Bernays-Schönfinkel-Klasse

Die Bernays-Schönfinkel-Klasse bietet die nützliche Eigenschaft, dass sie ein entscheidbares Allgemeingültigkeitsproblem hat, und ist deswegen für uns von Bedeutung. Sie beinhaltet Aussagen, die alle einer bestimmten Form entsprechen.
Dafür muss man zunächst mal wissen, was eine Pränexform ist:

Definition 3.2.1:
*Eine Formel befindet sich in **Pränexform**, wenn alle Quantoren außerhalb bzw. vor der eigentlichen Formel stehen, d.h. von der Gestalt $Q_1\ x_1\ Q_2\ x_2\ \dots\ Q_n\ x_n\ F$ ist, mit*
- Q_i entweder $\exists$ oder $\forall$
- x_i paarweise verschieden
- F quantorenfrei
- F atomar oder in Klammern

In der Prädikatenlogik gibt es zu jeder Formel eine logisch äquivalente Formel in Pränexform.
Hier zwei Beispiele, wie eine Formel in Pränexform aussieht:

Beispiel 3.2.1:

(1) Der Ausdruck $\forall x\, P(x) \wedge \forall y\, Q(y) \wedge \forall z\, R(z)$ *sieht in Pränexform so aus:*
$\forall x\ \forall y\ \forall z\ \ (P(x) \wedge Q(y) \wedge R(z))$
(2) Ein weiterer Ausdruck in Pränexform:
$\exists x\ \forall y\ (S(x,y) \vee S(y,x))$

Mit dieser Definition kann man nun die Aussagen in der Bernays-Schönfinkel-Klasse beschreiben:

Definition 3.2.2:
Jede Aussage in der ***Bernays-Schönfinkel-Klasse*** *ist in Pränexform und hat den Präfix* $\forall^* \exists^*$

3.3 Annahmen über Konstanten und Domänen

Es gibt verschiedene Domänen, auf denen man in der kontrollierten Anfrageauswertung arbeiten kann. Im folgenden werden drei verschiedene Ansätze vorgestellt.

3.3.1 Feste endliche Domäne

Der erste Ansatz geht von einer festen endlichen Domäne aus. Dabei hat man eine endliche Menge von Relationen und eine endliche Menge von Konstanten.
Unter dieser Annahme kann jedes Entscheidungsproblem in der Prädikatenlogik, im speziellen jedes Implikationsproblem, auf ein äquivalentes Entscheidungsproblem in der Aussagenlogik reduziert werden, was effektiv gelöst werden kann.
Die Umwandlung der prädikatenlogischen Formeln in aussagenlogische Formeln kann wie folgt gemacht werden:
Die endlich vielen Grundatome können jedes einzeln als Aussagenvariable dargestellt werden. Die Allquantoren und Existenzquantoren können mit Hilfe der Konjunktion und der Disjunktion ersetzt werden.

3.3.2 Variierende endliche Domäne

In diesem Fall geht man von einer festen unendlichen Menge von Konstanten aus. Die Instanz eines Informationssystems wählt aus dieser Menge eine endliche Untermenge als ihre Domäne aus (Jede Instanz wählt eine eigene Untermenge).
Dieser Ansatz ist allerdings für die Aufgabe der kontrollierten Anfrageauswertung nicht angemessen, da man das Problem hat, bestimmen zu müssen, ob ein Grundatom falsch ist oder ob es einfach nur in der aktuellen Untermenge nicht definiert ist, weil es nicht Teil dieser speziellen Untermenge ist.

3.3.3 Feste unendliche Herbrand-Domäne

Definition 3.3.1 (Herbrand-Universum)**:**
*Ein **Herbrand-Universum** $D(F)$ einer Formel F besteht aus allen Termen, die sich aus den in F vorkommenden Konstantensymbolen und Funktionssymbolen bilden lassen. Falls F keine Konstantensymbole enthält, wird a als Konstantensymbol dazu genommen.*

Der dritte Ansatz geht auch von einer festen unendlichen Menge von Konstanten aus, die aber im Gegensatz zum zweiten Ansatz von jeder Instanz des Informationssystems einheitlich als unendliche Domäne genommen wird. Diese Domäne wird auch im folgenden als Grundlage genommen, weil sie die geringsten Einschränkungen bezüglich der zugrunde liegenden Domäne hat und somit auch einen größeren Spielraum bietet.
Eine nützliche Eigenschaft bezüglich der Entscheidbarkeit von Implikationen in dieser Domäne ist, dass, wenn die Aussagen in Pränexform mit dem Präfix $\forall^*$ oder $\exists^*$ sind, dann jede Implikation $\Psi \models \Phi$ zwischen diesen Aussagen äquivalent zur Allgemeingültigkeit von $\neg\Psi \vee \Phi$ ist, was wiederum in der Bernays-Schönfinkel-Klasse ist.
Ein paar Einschränkungen für Anfragen auf dieser Domäne müssen allerdings gemacht werden. Wenn man eine feste unendliche Herbrand-Domäne und eine offene Anfrage hat, so kann die offene Anfrage durch eine Folge von geschlossenen Anfragen beantwortet werden. Es ist klar, dass man möchte, dass diese Anfrage nach endlich vielen Substitutionen auch terminiert. Deswegen beschränkt man die Anfragen auf **sichere und domänenunabhängige Anfragen**. Sichere Anfragen garantieren,

dass man endliche Anfrageergebnisse erhält. Domänenunabhängige Anfragen liefern immer das gleiche Ergebnis zurück, unabhängig von der gerade aktiven Domäne.
Des Weiteren benötigt man einen sogenannten **Vollständigkeitstest**, um zu erkennen, wann das Anfrageergebnis vollständig aufgezählt ist und man keine weiteren Substitutionen betrachten muss. Dabei fragt man nach jeder Substitution, ob jede weitere Substitution, die noch nicht durchgeführt wurde, die Anfrageformel falsch machen würde. Diesen Test kann man als geschlossene Anfrage formulieren, die auch den Einschränkungen für die Entscheidbarkeit der Implikationen genügt.

3.4 Relationales Submodell

3.4.1 Grundlagen

Verschiedene Definitionen

Ein **Informationssystem** besteht aus einem Schema und einer Instanz.
Ein **Schema DS** besteht aus einer abzählbaren, unendlichen Menge *dom* von Konstanten und einer endlichen Menge von Prädikatennamen.
Eine **Instanz db** ist eine Herbrand-Interpretation, die die Prädikatennamen interpretiert, unter der Annahme, dass die Konstanten selbsterklärend sind.

Zunächst nochmal eine kurze Übersicht der Symbole und Schreibweisen, die in den folgenden Lemma und Theoremen verwendet werden:

L:	Prädikatenlogische Sprache (Konstanten sind Untermenge von *dom*)
dom(I):	Domäne einer Interpretation I
p^I:	Interpretation eines Prädikatensymbols p in I
c^I:	Interpretation eines Konstantensymbols c in I
σ:	Substitution
φ und ψ:	Logische Formeln
I,$\sigma \models \varphi$:	Formel φ ist wahr in I, wenn ihre freien Variablen gemäß Substitution σ interpretiert werden

Nun folgen noch eine Reihe von Definitionen von Begriffen, die für die Lemma und Theoreme wichtig sind:

Definition 3.4.1 (DB-Interpretation/DB-Instanz):
Eine Interpretation I für L ist eine DB-Interpretation oder DB-Instanz, wenn gilt:
1. dom(I)=dom,
2. p^I ist endlich für alle Prädikatensymbole p in L
3. $c_i^I=c_i$ für alle Konstanten c_i in L

Eine DB-Interpretation ist also die Klasse der Interpretationen, die relationale Datenbankinstanzen repräsentieren.

Definition 3.4.2 (Aktive Domäne):
Eine aktive Domäne $active_\varphi(I)$ einer Interpretation I ist die Menge aller $d \in dom(I)$, die in p^I auftauchen mit $p \in L$, sowie aller c^I, für die c ein Konstantensymbol in φ ist. Formal:

$$active_\varphi(I) = \{d \mid \exists\, d_1,\ldots,d_n,\ i<n,\ \langle d_1,\ldots,d_i,d,d_{i+2},\ldots,d_n \rangle \in p^I\} \cup \{c^I \mid c \in const(\varphi)\}$$

Die Menge const(φ) bezeichnet die Menge von Konstantensymbolen, die in φ auftauchen. Die aktive Domäne einer DB-Interpretation ist immer endlich (ebenso bei endlichen Interpretationen).

Definition 3.4.3 (Out_φ(x)):
$Out_\varphi(x)$ bezeichnet eine Formel, die besagt, dass x nicht zu der aktiven Domäne bzgl. φ gehört.
$(\exists x)Out_\varphi(x)$ heißt ***non-totality assumption*** *für φ.*

Die non-totality assumption ist eine Tautologie (also eine Aussage, die immer wahr ist) in der Klasse der DB-Interpretationen, aber nicht in der Klasse der endlichen und klassischen Interpretationen. Für diese muss die Gültigkeit der non-totality assumption explizit gefordert werden.

Definition 3.4.4 (Unique name assumption):
Für alle Formeln φ sei UNA_φ der folgende Satz:
$\bigwedge \{c_i \neq c_j \mid \{c_i,c_j\} \in const(\varphi) \text{ und } i<j\}$

Diese Definition formalisiert, dass unterschiedliche Namen auch unterschiedliche Objekte bezeichnen. Alle UNA sind gültig in allen DB-Interpretationen, aber nicht notwendigerweise in den anderen Interpretationen. Auch hier muss, falls gewünscht, die Gültigkeit explizit gefordert werden.

Definition 3.4.5 (Nominal Equality):

Eine prädikatenlogische Formel φ hat nominelle Gleichheit, wenn für alle Gleichungen t=u, die in φ auftauchen, t eine Variable und u eine Konstante ist.
φ wird dann auch als eine ***nominal equality formula*** *bezeichnet.*

Diese Definition garantiert eine Normalisierung von Gleichheitsformeln, indem sie Ketten von Gleichungen zwischen Variablen vermeidet.

Verschiedene Lemma

Für die Hauptaussagen dieses Themas werden noch ein paar Lemma benötigt.

Lemma 3.4.1:
Sei φ eine nominal equality formula und I eine Interpretation.
Für alle $\overline{c} \in dom(I) \setminus active_\varphi(I)$, alle Substitutionen σ und alle Variablen v, für die $\sigma(v) \notin active_\varphi(I)$:

$$I,\sigma \models \varphi \text{ gdw. } I,\sigma[v/\overline{c}] \models \varphi.$$

Lemma 3.4.2:
Sei φ eine nominal equality formula und I eine DB-Interpretation.
Für alle $\overline{c} \in dom(I) \setminus active_\varphi(I)$ und alle Substitutionen σ:

$$I,\sigma \models \varphi \text{ gdw. } J,\tau \models \varphi$$

mit J als Einschränkung von I auf $active_\varphi(I) \cup \overline{c}$, und

$$\tau(x) = \begin{cases} \sigma, & \text{falls } \sigma(x) \in active_\varphi(I) \\ \overline{c}, & \text{sonst.} \end{cases}$$

Lemma 3.4.3:
Sei φ eine nominal equality formula und sei J eine endliche Interpretation, so dass $J \models (\exists\, x)\, Out_\varphi(x)$.
Sei $f{:}dom(J) \rightarrow dom$ irgendeine totale, injektive Funktion, so dass für alle $c_i \in const(\varphi)$: $f(c_i^J)=c_i$
Sei I eine DB-Interpretation, so dass für alle Prädikate p: $p^I = \{\langle f(d_1),\ldots,f(d_n)\rangle \mid \langle d_1,\ldots,d_n\rangle \in p^J\}$

Dann gilt für alle Substitutionen σ:

$$J,\sigma \models \varphi \text{ gdw. } I,(f \circ \sigma) \models \varphi, \text{ mit } (f \circ \sigma)(x) = f(\sigma(x))$$

Das Theorem setzt die Erfüllbarkeit in der Klasse der DB-Interpretationen in Ver-

bindung mit der Erfüllbarkeit in der Klasse der endlichen Interpretationen:

Theorem 3.4.1:
Für alle nominal equality sentences φ sind folgende Aussagen äquivalent:
1. φ wird von einer DB-Interpretation I erfüllt.
2. φ wird von einem endlichen Modell J von $(\exists x)Out_\varphi(x)$ und UNA_φ erfüllt.

Wie bereits oben erwähnt, muss bei endlichen Interpretationen die Gültigkeit der *non-totality assumption* und der *unique name assumption* explizit gefordert werden.

Endliches Modell-Eigenschaft

Definition 3.4.6 (Endliches Modell-Eigenschaft)**:**
Sei $\mathcal{L}$ ein Fragment der Prädikatenlogik. Dann sagt man, dass $\mathcal{L}$ die Endliche Modell-Eigenschaft hat, wenn für alle Sätze $\chi \in L$ gilt:
Wenn χ endlich-allgemeingültig ist, dann ist χ allgemeingültig.
D.h. Wenn für alle endlichen Interpretationen I gilt $I \models \chi$, dann hat man auch $I \models \chi$ für alle (endlichen und unendlichen) Interpretationen I.

Ein Fragment mit der Endliche Modell-Eigenschaft hat ein entscheidbares Allgemeingültigkeitsproblem:
Wegen der Vollständigkeit der Prädikatenlogik ist das Allgemeingültigkeitsproblem rekursiv aufzählbar und somit semi-entscheidbar. Auch das Komplement des Problems ist wegen der Endliche Modell-Eigenschaft rekursiv aufzählbar und damit ebenfalls semi-entscheidbar. Da sowohl das Problem als auch das Komplement semi-entscheidbar sind, ist das Allgemeingültigkeitsproblem für das Fragment sogar entscheidbar.

3.4.2 Entscheidbarkeit der Implikationen

Mit diesen Ergebnissen kann man nun für ein Fragment $\mathcal{L}$ der Prädikatenlogik zeigen, dass klassische Implikationen, endliche Implikationen und DB-Implikationen äquivalent und entscheidbar sind.

Die gegenseitige Reduktion von Implikationsproblem und Allgemeingültigkeitspro-

blem sagt aus, dass für jede der drei Implikationen eine Folgerung $\Psi \models \Phi$ von zwei Aussagen Ψ und Φ äquivalent zur Allgemeingültigkeit der Aussage $\neg \Psi \vee \Phi$ ist.
Für das folgende Theorem definieren wir noch:
$\neg \mathcal{L} = \{\neg \varphi \mid \varphi \in \mathcal{L}\}$
Das alles führt nun zum entscheidenden Theorem dieses Themas, das die Bedingungen zusammenfasst, unter denen die drei Arten der Implikationen alle entscheidbar sind:

Theorem 3.4.2:
Sei ein $\mathcal{L}$ ein Fragment der Prädikatenlogik mit nomineller Gleichheit, abgeschlossen unter $\vee$, mit der Endliche Modell-Eigenschaft und fähig die negierte non-totality assumption $\neg(\exists x)Out_\varphi(x)$ und die negierte unique name assumption $\neg\ UNA\varphi$ für alle $\mathcal{L}$-Formeln φ auszudrücken.
Dann sind die Einschränkungen zu $\neg\mathcal{L}$ x $\mathcal{L}$ der Beziehungen $\models_{gen}$, $\models_{DB}$ und $\models_{fin}$ alle entscheidbar.
Außerdem sind für alle $\Psi \in \neg\mathcal{L}$ und alle $\varphi \in \mathcal{L}$ die folgenden Eigenschaften äquivalent:
1. $\Psi \models_{DB} \varphi$;
2. $\Psi \wedge (\exists x)Out_\varphi(x) \wedge UNA_\varphi \models_{gen} \varphi$;
3. $\Psi \wedge (\exists x)Out_\varphi(x) \wedge UNA_\varphi \models_{fin} \varphi$.

Kapitel 4

Automatisches Beweisen

4.1 Einleitung

4.1.1 Was bedeutet „automatisches Beweisen“?

Automatisches Beweisen – ein Teil der automatischen Deduktion – basiert auf der Verwendung von Computerprogrammen zur Erzeugung und Überprüfung von mathematischen Beweisen von logischen Theoremen. Dass es prinzipiell möglich ist, mathematische Wahrheiten formal zu modellieren und mittels einer beschränkten Anzahl von Axiomen und Regeln formal zu beweisen, wusste man seit den grundlegenden Untersuchungen zur Axiomatisierbarkeit der Logik und Mathematik.

4.1.2 Modellierung

Mathematische Probleme werden in einer geeigneten formalen Sprache formalisiert. Dabei wird insbesondere an logische Syntax gedacht. Formalen syntaktischen Objekten (auch Aussagen genannt) wird mittels einer Wahrheitsfunktion, die jeder Aussage einen Wahrheitswert zuordnet, eine Semantik gegeben. Mit Hilfe der Semantik kann die Adäquatheit der Formalisierung überprüft werden. Dieser Gesamtprozess wird Modellierung genannt.

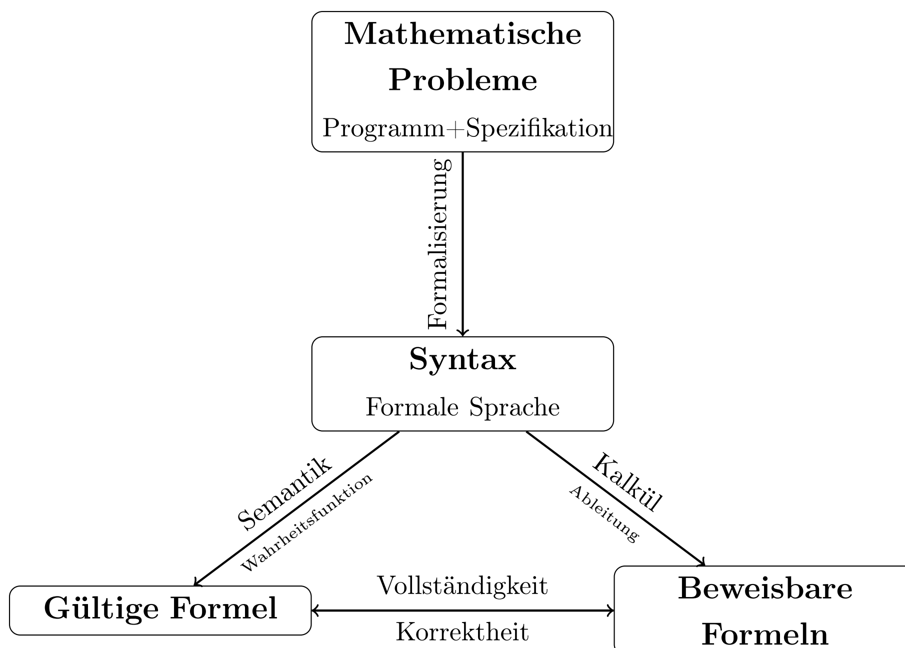


Abbildung 4.1: Zentrale Begriffe der mathematischen Logik und deren Beziehungen (nach B. Becker)

4.2 Logische Grundlagen

4.2.1 Aussagenlogik [AL]

Komplexe Aussagen werden in der Aussagenlogik aus elementaren Aussagen durch Klammern (,) und Junktoren (logische Verknüpfungsoperatoren) „¬“ „∨“ „∧“ „⇒“ „⇔“ aufgebaut. Elementare Aussagen sind „wahr“ oder „falsch“ und nicht weiter zerlegbar. Der Wahrheitswert einer komplexen Aussage ist von den Wahrheitswerten der Teilaussagen und dem gewählten Junktor abhängig. Daher ist z.B. das Ausdrücken einer Verknüpfung wie „A solange B“ in der Aussagenlogik nicht möglich.

Syntax

- Aussagenlogische Variablen der Art a, b, c sowie a_i, $i \in \mathbb{N}$ (und weitere, wenn nötig) werden verwendet.

- Signatur Σ bezeichnet die Menge aller aussagenlogischen Variablen.

Definition 4.2.1:
Die Menge Form(Σ) der aussagenlogischen Formeln ist die kleinste Menge, die die folgenden Eigenschaften hat:

1. *true und false sind in Form(Σ)*
2. *jede Variable in Σ ist in Form(Σ)*
3. *sind F und G in Form(Σ), so auch $\neg F$, $F \wedge G$, $F \vee G$*

- Formeln der Typen (1) und (2) heißen Atom. Wahrheitswerte sind $true$ und $false$ und werden 1 statt $true$, 0 statt $false$ geschrieben.
- Die logischen Operatoren $\Rightarrow$ und $\Leftrightarrow$ sind abkürzende Schreibweisen wie folgt:
  $$F \Rightarrow G \quad \text{für} \quad \neg F \vee G$$
  $$F \Leftrightarrow G \quad \text{für} \quad (F \wedge G) \vee (\neg F \wedge \neg G)$$
- Für eine Formel $F \in \text{Form}(\Sigma)$ ist $\text{Var}(F)$ die Menge aller in F auftretenden Variablen. Ein Literal ist ein Atom oder dessen Negation (z.B. true, $x, \neg y$, ...)

Semantik

- Eine Interpretation (oder Belegung) I ist eine partielle Funktion $I : \Sigma \rightarrow \{0, 1\}$ und $Int(\Sigma)$ ist die Menge aller Σ-Interpretationen.
- Eine Belegung I heißt passend für eine Formel F, wenn $I(x)$ für alle in F vorkommenden Variablen x definiert ist.
- Die Semantik von F ordnet jeder passenden Belegung I einen Wahrheitswert $[\![F]\!]_I$ zu.

Definition 4.2.2:
Seien $F, G \in$ Form(Σ) und I eine zu F passende Belegung. Dann induziert I eine

Auswertungsfunktion $[\![\ ]\!]_I : Form(\Sigma) \to \{0,1\}$ *mit der Vorschrift:*

$$[\![true]\!]_I =_{def} 1$$
$$[\![false]\!]_I =_{def} 0$$
$$[\![x]\!]_I =_{def} I(x) \text{ für Variablen } x \in \Sigma$$
$$[\![\neg F]\!]_I =_{def} \begin{cases} 1 & \text{falls } [\![F]\!]_I = 0 \\ 0 & \text{falls } [\![F]\!]_I = 1 \end{cases}$$
$$[\![(F \wedge G)]\!]_I =_{def} \begin{cases} 1 & \text{falls } [\![F]\!]_I = 1 \text{ und } [\![G]\!]_I = 1 \\ 0 & \text{andernfalls} \end{cases}$$
$$[\![(F \vee G)]\!]_I =_{def} \begin{cases} 1 & \text{falls } [\![F]\!]_I = 1 \text{ oder } [\![G]\!]_I = 1 \\ 0 & \text{andernfalls} \end{cases}$$

Beispiel 4.2.1:
Die Aussage „Jede Primzahl größer als 2 ist ungerade“ lässt sich als

$$(\chi_p \wedge \chi_{>2}) \Rightarrow \chi_u$$

modellieren mit Hilfe der folgenden atomaren Aussagen:
χ_p : *x ist Primzahl*
$\chi_{>2}$: *x ist größer als 2*
χ_u : *x ist ungerade*

Definition 4.2.3 (Erfüllbarkeit und Gültigkeit)**:**
Eine Formel $F \in Form(\Sigma)$ *heißt erfüllbar, wenn es eine Variablenbelegung* I *mit* $[\![F]\!]_I = 1$ *gibt. Ansonsten heißt* F *unerfüllbar. Gilt für alle Variablenbelegung* I, *dass* $[\![F]\!]_I = 1$, *so heißt* F *(allgemein-)gültig oder Tautologie.*

Definition 4.2.4 (semantische Folgerung)**:**
Sei $\mathcal{M} \subseteq Form(\Sigma)$ *eine Formelmenge und* $F \in Form(\Sigma)$. *Dann folgt* F *semantisch aus* $\mathcal{M}$, *falls für jede Belegung* I *mit* $[\![G]\!]_I = 1$ *für alle* $G \in \mathcal{M}$ *auch* $[\![F]\!]_I = 1$ *gilt. Schreibweise* $\mathcal{M} \models_I F$.
$\models F$ *ist äquivalent zu „F ist eine Tautologie“.*

Lemma 4.2.1:
Sei $F \in Form(\Sigma)$. *Es gilt:*

1. *F erfüllbar $\Leftrightarrow \neg F$ keine Tautologie (F Tautologie $\Leftrightarrow \neg F$ unerfüllbar).*

2. *$\models F \wedge G \Leftrightarrow \models F$ und $\models G$.*

Unerfüllbarkeitstest:
Jede Formel F enthält n Aussagenvariablen (n ist endlich). $[\![F]\!]_I$ ist nur von den Werten dieser Aussagenvariablen abhängig. Um die Erfüllbarkeit von F zu testen, ist es notwendig, 2^n Wertbelegungen zu überprüfen. Das kann beispielsweise durch eine Wahrheitstabelle ausgeführt werden.
Das Erfüllbarkeitsproblem ist entscheidbar und NP-vollständig (Satz von Cook).

Lemma 4.2.2 (Substitutionslemma)**:**
Seien F und G äquivalente Formeln. Sei S eine Formel mit mindestens einem Vorkommen der Teilformel F. Dann ist S äquivalent zu einer Formel S', wobei S' aus S hervorgeht, indem irgendein Vorkommen von F in S durch G ersetzt wird.

Äquivalenzen für die Aussagenlogik

Zwei Formeln F und G sind (semantisch) äquivalent, geschrieben $F \equiv G$,
genau dann, wenn $\Leftrightarrow$ für alle Interpretationen $I : \Sigma \to \{0, 1\}$ gilt: $I \models F \Leftrightarrow I \models G$
Man schreiben auch $F \equiv G$ für $\models F \Leftrightarrow G$. Es gelten:

1.	$F \wedge F \equiv F$	(Idempotenz)
	$F \vee F \equiv F$	
2.	$F \wedge G \equiv G \wedge F$	(Kommutativität)
	$F \vee G \equiv G \vee F$	
3.	$(F \wedge G) \wedge H \equiv F \wedge (G \wedge H)$	(Assoziativität)
	$(F \vee G) \vee H \equiv F \vee (G \vee H)$	
4.	$(F \wedge (F \vee G) \equiv F$	(Absorption)
	$(F \vee (F \wedge G) \equiv F$	
5.	$(F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$	(Distributivität)
	$(F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$	
6.	$\neg\neg F \equiv F$	(Doppelnegation)
7.	$\neg(F \wedge G) \equiv \neg F \vee \neg G$	(de Morgansche Regeln)
	$\neg(F \vee G) \equiv \neg F \wedge \neg G$	
8.	$F \Rightarrow G \equiv \neg G \Rightarrow \neg F$	(Kontraposition)
9.	$F \Rightarrow G \equiv \neg F \vee G$	(Implikation)
10.	$F \Leftrightarrow G \equiv (F \Rightarrow G) \wedge (G \Rightarrow F)$	(Koimplikation)

Normalformen

Die Menge der aussagenlogischen Formeln ist reich an Redundanzen. So liefern beispielsweise die Formeln $A \vee B$ und $\neg(\neg A \wedge \neg B)$ die gleiche Semantik. In vielen Situationen werden aber sog. Normalformen benötigt, die eine gewisse Standarddarstellung für Formeln sind.

Definition 4.2.5 (Negationsnormalform)**:**
Eine Formel $F \in Form(\Sigma)$ ist in Negationsnormalform (NNF)*, wenn Negationen nur direkt vor Aussagenvariablen auftreten.*

Beispiel 4.2.2:
$\neg(x \vee y)$,$\neg\neg x$ *nicht in NNF;* $\neg x \wedge \neg y$ *in NNF.*

Lemma 4.2.3:
Zu jeder Formel $F \in Form(\Sigma)$ gibt es eine äquivalente Formel in NNF.

Algorithmus für NNF: Ersetze

$$\begin{aligned} \neg(F \wedge G) &\rightsquigarrow \neg F \vee \neg G \\ \neg(F \vee G) &\rightsquigarrow \neg F \wedge \neg G \\ \neg\neg F &\rightsquigarrow F \end{aligned}$$

Definition 4.2.6 (Klausel und Minterm)**:**
Eine Disjunktion $F = (l_1 \vee \cdots \vee l_n)$ *von Literalen* l_i, $(1 \leq i \leq n)$ *heißt Klausel.*
Kommt höchstens ein Literal in F *positiv vor, so heißt* F Horn-Klausel.
Eine Konjunktion $F = (l_1 \wedge \cdots \wedge l_n)$ *von Literalen* $l_i(1 \leq i \leq n)$ *heißt Minterm.*

Definition 4.2.7 (Konjunktive Normalform)**:**
Eine Formel F *ist in konjunktiver Normalform* (KNF), *wenn* F *eine Konjunktion von Klauseln ist, also* $F = F_0 \wedge \cdots \wedge F_n$ *für Klauseln* F_i.

Beispiel 4.2.3:
$F = (x \vee \neg y \vee \neg z) \wedge (x \vee \neg y) \wedge (x \vee z)$ *ist in KNF.*
Als Mengenschreibweise für Formeln in KNF kann $F = \{\{x, \neg y, \neg z\}, \{x, \neg y\}, \{x, z\}\}$ *geschrieben werden.*

Definition 4.2.8 (Disjunktive Normalform)**:**
Eine Formel F *ist in disjunktiver Normalform* (DNF), *wenn* F *eine Disjunktion von Mintermen ist.*

Beispiel 4.2.4:
$F = (x \wedge \neg y \wedge z) \vee (x \wedge y) \vee (y \wedge \neg z)$ *ist in DNF.*

Lemma 4.2.4:
Jeder Formel F *ist mit einer Formel in KNF äquivalent.*
Jede Formel F *ist mit einer Formel in DNF äquivalent.*

Algorithmus für KNF (für vereinfachte Formel in NNF): Ersetze

1.	Äquivalenzeliminierung	$F \Leftrightarrow G$	$\rightsquigarrow$	$(F \Rightarrow G) \wedge (G \Rightarrow F)$
2.	Implikationseliminierung	$F \Rightarrow G$	$\rightsquigarrow$	$\neg F \vee G$
3.	Negation-nach-innen	$\neg(F \vee G)$	$\rightsquigarrow$	$\neg F \wedge \neg G$
4.	Eliminierung doppelter Negationen	$\neg\neg F$	$\rightsquigarrow$	F
5.	Disjunktionen-nach-innen	$F \vee (G \wedge H)$	$\rightsquigarrow$	$(F \vee G) \wedge (F \vee H)$
6.	$\top$- und $\bot$-Eliminierung			

Die Konversion zu DNF ist ähnlich (aber „Konjunktionen-nach-innen“ im Schritt 5)

Satz 4.2.1 (Deduktionstheorem)**:**
Sei $\mathcal{M} \subseteq Form(\Sigma)$ *und* $F, G \in Form(\Sigma)$. *Dann gilt:*

$$\mathcal{M} \cup \{F\} \models G \text{ impliziert } \mathcal{M} \models F \Rightarrow G$$

Anwendung zum Nachweis von $\mathcal{M} \models F$, wobei $\mathcal{M} = \{G_1, \ldots, G_n\}$
$\mathcal{M} \models F$?
$\{G_1, \ldots, G_n\} \models F$
$\{G_1, \ldots, G_{n-1}\} \models G_n \Rightarrow F$
$\vdots$
$\models G_1 \Rightarrow (\ldots(G_n \Rightarrow F))$ oder $\models (G_1 \wedge \ldots \wedge G_n) \Rightarrow F$

4.2.2 Prädikatenlogik 1. Stufe [PL1]

Prädikatenlogik ist eine Erweiterung der Aussagenlogik. In der Aussagenlogik werden zusammengesetzte Aussagen daraufhin untersucht, aus welchen einfacheren Aussagen sie mit Hilfe von Junktoren aufgebaut sind. Im Gegensatz zur Aussagenlogik sind bei der Prädikatenlogik die elementaren Aussagen nicht atomar, sondern können aus Relationen, Funktionen und Variablen zusammengesetzt werden.

Signaturen und Interpretation

Definition 4.2.9 (Signatur)**:**
Eine (PL1-)Signatur $\Sigma = (Func, Pred)$ besteht aus einer Menge $Func$ von Funktionssymbolen und einer Menge $Pred$ von Prädikatensymbolen. Dabei hat jedes Symbol $s \in Func \cup Pred$ eine feste Stelligkeit ≥ 0. Ein Funktionssymbol mit der Stelligkeit 0 heißt Konstante. Generell wird für jede PL1-Signatur vorausgesetzt, dass es mindestens eine Konstante gibt.

Eine Σ-Interpretation I weist den Symbolen einer Signatur Σ Bedeutungen über einer Menge von Objekten zu. Hierfür notwendig ist ein Universum U, das eine beliebige, nicht-leere Menge ist. Sie enthält alle Objekte der Interpretation.
Den Funktions- und Prädikatensymbolen in Σ weist eine Interpretation Funktionen bzw. Relationen wie folgt zu:

- Nullstellige Funktionssymbole werden Objekte aus U zugeordnet.
- Ein oder mehrstelligen Funktionssymbolen werden Funktionen zugeordnet, die aus den gegebenen Parametern in die Menge U abbilden.

- Nullstellige Prädikatensymbole werden wie Aussagevariablen in der Aussagenlogik behandelt, d.h. ihnen werden unmittelbar Wahrheitswerte zugeordnet. Einstelligen Prädikatensymbolen werden Teilmengen aus U zugeordnet.
- Mehrstellige Prädikatensymbole werden durch Relationen entsprechender Stelligkeit über U interpretiert.

Definition 4.2.10 (Interpretation)**:**
Sei $\Sigma = (Func, Pred)$ *eine Signatur. Eine* Σ*-Interpretation* $I = (U_I, Func_I, Pred_I)$ *besteht aus:*

- *einer nichtleeren Menge* U_I*, genannt Universum oder Trägermenge.*
- *einer Menge* $Func_I$ *von Funktionen*
  $Func_I = \{f_I : \underbrace{U_I \times \ldots \times U_I}_{n-mal} \rightarrow U_I \mid f \in Func \text{ mit der Stelligkeit } n\}$
- *einer Menge* $Pred_I$ *von Relationen*
  $Pred_I = \{p_I \subseteq \underbrace{U_I \times \ldots \times U_I}_{n-mal} \mid p \in Pred \text{ mit der Stelligkeit } n\}$

Die Menge der Σ*-Interpretationen wird mit* $Int(\Sigma)$ *bezeichnet.*

Terme und Termauswertung

Im Gegensatz zur Aussagenlogik können in der Prädikatenlogik Terme formuliert werden, die durch Objekte des Universums interpretiert werden.

Definition 4.2.11 (Term)**:**
Die Menge $Term_\Sigma(V)$ *der Terme über einer Signatur* $\Sigma = (Func, Pred)$ *und einer Menge* V *von Variablen ist die kleinste Menge, die die folgenden Elemente gemäß (1) - (3) enthält:*

(1) x*, falls* $x \in V$

(2) c*, falls* $c \in Func$ *und* c *hat die Stelligkeit 0*

(3) $f(t_1, \ldots, t_n)$*, falls* $f \in Func$ *mit der Stelligkeit* $n > 0$ *und* $t_1, \ldots, t_n \in Term_\Sigma(V)$

Ein Grundterm über Σ *ist ein Element aus* $Term_\Sigma(\emptyset)$*, d.h. ein Term ohne Variablen.* $Term_\Sigma =_{def} Term_\Sigma(\emptyset)$ *bezeichnet die Menge der Grundterme.*

Definition 4.2.12 (Variablenbelegung)**:**
Sei $I = (U_I, Func_I, Pred_I)$ *eine Interpretation und* V *eine Menge von Variablen. Eine Variablenbelegung ist eine Funktion*

$$\sigma : V \to U_I$$

Definition 4.2.13 (Termauswertung)**:**
Gegeben sei ein Term $t \in Term_\Sigma(V)$*, eine* Σ*-Interpretation* I *und eine Variablenbelegung* $\sigma : V \to U_I$*. Die Termauswertung von* t *in* I *unter* σ*, geschrieben* $[\![t]\!]_{I,\sigma}$*, ist gegeben durch eine Funktion*

$$[\![\ ]\!]_{I,\sigma} : Term_\Sigma(V) \to U_I$$

und ist definiert durch

$$\begin{aligned} [\![x]\!]_{I,\sigma} &= \sigma(x) \\ [\![f(t_1,\ldots,t_n)]\!]_{I,\sigma} &= f_I([\![t_1]\!]_{I,\sigma},\ldots [\![t_n]\!]_{I,\sigma}) \end{aligned}$$

Formeln und Formelauswertung

Definition 4.2.14 (atomare Formel, Atom)**:**
Eine atomare Formel (oder Atom) über einer Signatur $\Sigma = (Func, Pred)$ *und einer Menge* V *von Variablen wird wie folgt gebildet:*

1. p*, falls* $p \in Pred$ *und* p *hat die Stelligkeit 0*
2. $p(t_1,\ldots,t_n)$*, falls* $p \in Pred$ *mit der Stelligkeit* $n > 0$ *und* $t_1,\ldots,t_n \in Term_\Sigma(V)$

Um komplexere Formeln zu bilden, stehen neben den aus der Aussagenlogik bekannten Junktoren $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow$ zwei Quantoren – der Allquantor $\forall$ und der Existenzquantor $\exists$ – zur Verfügung:

$\forall x F$ „Für alle x gilt F"
$\exists x F$ „Es gibt ein x, für das F gilt"

Definition 4.2.15 (Formel)**:**
Die Menge $Formel_\Sigma(V)$ *der Formeln über einer Signatur* $\Sigma = (Func, Pred)$ *und einer Menge* V *von Variablen ist die kleinste Menge, die die folgenden Elemente gemäß (1) - (3) enthält:*

(1) P, falls P ein Atom über Σ und V ist

(2) $\neg F$, $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \Rightarrow F_2$, $F_1 \Leftrightarrow F_2$

(3) $\exists x F$, $\forall x F$

wobei $x \in V$ und $F, F_1, F_2 \in Formel_\Sigma(V)$ sind.
Ein Ausdruck über Σ (und V) ist ein Term oder eine Formel über Σ (und V). Eine Formel ohne Variablen heißt Grundformel.

Definition 4.2.16 (Formelauswertung)**:**
Gegeben sei eine Formel $F \in Formel_\Sigma(V)$, eine Σ-Interpretation I und eine Belegung $\sigma : V \to U_I$. Der Wahrheitswert von F in I unter σ, geschrieben $[\![F]\!]_{I,\sigma}$, ist definiert durch

$$
\begin{array}{lll}
[\![P(t_1,\ldots,t_n)]\!]_{I,\sigma} & = \mathit{true}\ \mathit{gdw.} & ([\![t_1]\!]_{I,\sigma},\ldots,[\![t_n]\!]_{I,\sigma}) \in P_I \\
[\![\forall x F]\!]_{I,\sigma} & = \mathit{true}\ \mathit{gdw.} & [\![F]\!]_{I,\sigma_{x/a}} = \mathit{true}\ \text{für jedes } a \in U \\
[\![\exists x F]\!]_{I,\sigma} & = \mathit{true}\ \mathit{gdw.} & [\![F]\!]_{I,\sigma_{x/a}} = \mathit{true}\ \text{für mindestens ein } a \in U
\end{array}
$$

wobei

$$\sigma_{x/a}(y) = \begin{cases} \sigma(y) & \text{falls } y \neq x \\ a & \text{sonst} \end{cases}$$

Eine Variable x ist in einer Formel F gebunden, falls x in einer Teilformel der Form $\forall x G$ oder $\exists x G$ vorkommt. Sonst heißt x freie Variable. Eine Formel ohne freie Variablen heißt geschlossen oder auch Aussage.

Äquivalenzen für PL1

Theorem 4.2.1 (Semantische Äquivalenzen)**:**
Es gelten:

1.1	$\neg\forall xF$	$\equiv$	$\exists x\neg F$	
1.2	$\neg\exists xF$	$\equiv$	$\forall x\neg F$	
2.1	$(\forall xF)\wedge(\forall xG)$	$\equiv$	$\forall x(F\wedge G)$	
2.2	$(\exists xF)\vee(\exists xG)$	$\equiv$	$\exists x(F\vee G)$	
3.1	$\forall x\forall yF$	$\equiv$	$\forall y\forall xF$	
3.2	$\exists x\exists yF$	$\equiv$	$\exists y\exists xF$	
4.1	$\forall xF$	$\equiv$	$\forall yF[x/y]$	*falls y nicht*
4.2	$\exists xF$	$\equiv$	$\exists yF[x/y]$	*in F vorkommt*

Theorem 4.2.2 (Semantische Nicht-Äquivalenzen)**:**
Es gelten:

$$\begin{aligned}(\forall xF)\vee(\forall xG) &\neq \forall x(F\vee G)\\ (\exists xF)\wedge(\exists xG) &\neq \exists x(F\wedge G)\\ \forall x\exists yF &\neq \exists y\forall xF\end{aligned}$$

Normalformen

Definition 4.2.17 (Pränexnormalform)**:**
Eine Formel $F\in Form(\Sigma)$ *ist eine Pränexnormalform* (PNF)*, falls sie die Form* $Q_1x_1\dots Q_nx_nF_0$ *besitzt, wobei* $Q_i\in\{\exists,\forall\}$ *für* $1\leq i\leq n$ *und* F_0 quantorenfrei *ist.*$Q_1x_1\dots Q_nx_n$ *heißt Präfix.*

Definition 4.2.18 (NNF, KNF, DNF)**:**
Eine Formel ist in NNF, falls Negationen nur vor Relationssymbolen auftreten; d.h. in einer NNF wird $\neg\forall xF$ *durch* $\exists x\neg F$ *bzw.* $\neg\exists xF$ *durch* $\forall x\neg F$ *ersetzt*
KNF und DNF werden analog zu den Definitionen in der AL definiert.

Für die Resolution (s. Abschnitt 4.3) müssen die Formeln noch weiter vereinfacht werden.

Definition 4.2.19 (Skolem-Normalform)**:**

1. *Formeln* F_1, F_2 *heißen erfüllbarkeitsäquivalent, wenn sie entweder beide erfüllbar oder beide unerfüllbar sind.*

2. *Eine Formel ist in Skolemform, wenn sie die Form* $\forall x_1\cdots\forall x_nG$ *hat, wobei*

   - *G quantorenfrei ist*

- *im Quantoren-Präfix nur Allquantoren vorkommen*

3. *Jede Formel kann durch Anwendung der Skolem-Regel (und der Äquivalenzumwandlungen) in eine erfüllbarkeitsäquivalente Formel in Skolemform umgewandelt werden:*

   * *Ist* $F = \forall x_1 \forall x_2 \cdots \forall x_k \exists y G$, *wobei* G *eine quantorenfreie Formel ist, in der das Funktionssymbol* f *nicht vorkommt, so ist die Formel* $\forall x_1 \forall x_2 \cdots \forall x_k G[y/f(x_1, \ldots, x_k)]$ *erfüllbarkeitsäquivalent zu* F.

Eine Formel ist in Skolem-Normalform (SNF), *wenn sie in pränexer Normalform ist und ihr Präfix nur Allquantoren* ($\forall$) *enthält.*

Idee: Falls $\exists x P(x)$, dann gibt es eine Funktion, die dieses existierende Element bezeichnet. Das Element hängt ggf. von anderen Elementen ab:
$\forall x \exists y P(y) \rightsquigarrow P(f(x))$
$\exists y P(y) \rightsquigarrow P(c)$

Algorithmus für SNF: Die Umwandlung einer Formel F in Skolemform erfolgt in zwei Phasen:

1. Wandle F in eine äquivalente Formel F' in Pränexform um
2. Wandle F' durch Anwendung der Skolem-Regel in eine erfüllbarkeitsäquivalente Formel F'' um

Beispiel 4.2.5:
$\exists x \forall y \exists z (P(x, y) \wedge R(y, z))$
$\xrightarrow{k=0} (\forall y \exists z (P(c_0, y) \wedge R(y, z))[x/c_0] = \forall y \exists z (P(c_0, y) \wedge R(y, z)$
$\xrightarrow{k=1} (\forall y (P(c_0, y) \wedge R(y, z))[z/f_1(y)] = \forall y (P(c_0, y) \wedge R(y, f_1(y))$

Definition 4.2.20:
Eine freie Formel $F(x_1, \ldots, x_m)$ *heißt erfüllbar genau dann, wenn eine Interpretation* I *und eine Belegung* σ *existieren mit* $I, \sigma \models F(x_1, \ldots, x_m)$.

Definition 4.2.21 (Matrixklauselform)**:**
Sei F *eine Formel in Pränexform. Seien* $x_1, \ldots, x_m$ *die freien Variablen von* F. *Dann ist* F *erfüllbarkeitsäquivalent zu der geschlossenen Formel* $G = \exists x_1 \cdots \exists x_m F$. *Sei* $G' = \forall y_1 \cdots \forall y_k H$ *eine Skolemform zu* G. *Sei* H' *eine KNF zu* H. *So nennen wir die Klauselmenge* $\mathcal{K}$ *zu* H' *eine Matrixklauselform zu* F.

Unifikation

Definition 4.2.22 (Substitution)**:**
Eine Substitution σ ist eine (totale) Funktion

$$\sigma : Term_\Sigma(V) \rightarrow Term_\Sigma(V)$$

die Terme auf Terme abbildet, so dass die Homomorphiebedingung

$$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$$

für jeden Term $f(t_1, \ldots, t_n), n \geq 0$, gilt und so dass σ eingeschränkt auf V fast überall[1] *die Identität ist. Die Menge $dom(\sigma) = \{x \in V | \sigma(x) \neq x\}$ ist der Definitionsbereich von σ.*

Jede Substitution σ kann eindeutig durch eine endliche Menge von (Variable, Term)-Paaren $\{x_1/t_1, \ldots, x_n/t_n\}$ repräsentiert werden, wobei $dom(\sigma) = \{x_1, \ldots, x_n\}$ und $\sigma(x_i) = t_i$ gilt.

Beispiel 4.2.6:
Für die Substitution $\sigma = \{x/g(v), y/b\}$ gilt $\sigma(f(x, y, z)) = f(g(v), b, z)$ und $\sigma(h((x, x))) = h(g(v), g(v))$.

Für die Komposition $\rho \circ \sigma$ von zwei Substitutionen σ und ρ gilt $\rho \circ \sigma(t) = \rho(\sigma(t))$

Definition 4.2.23 (Unifikator)**:**
Eine Substitution σ heißt Unifikator *der Terme s und t, wenn $\sigma(s) = \sigma(t)$ gilt; in diesem Fall sind s und t unifizierbar.*

Beispiel 4.2.7:
Die Terme $f(x, b)$ und $f(a, c)$ sind nicht unifizierbar, ebenso sind $f(x)$ und $f(g(x))$ nicht unifizierbar.
Die Substitutionen $\sigma = \{x/b, y/a, z/g(a, a)\}$ und $\mu = \{x/b, z/g(a, y)\}$ sind zwei Unifikatoren von $t_1 = f(x, g(a, y))$ und $t_2 = f(b, z)$. Allerdings ist μ allgemeiner als σ insofern, als dass in σ die Variable y unnötigerweise instantiiert wird und $\sigma(z)$ aus $\mu(z)$ durch Instanzenbildung entsteht. Es gilt offensichtlich $\sigma = \sigma' \circ \mu$ mit $\sigma' = \{y/a\}$.

Definition 4.2.24 (allgemeinster Unifikator, mgu)**:**
Ein Unifikator μ von s und t heißt allgemeinster Unifikator *(most general unifier,*

[1] „fast überall“ bedeutet „alle bis auf endlich viele“

mgu), wenn es zu jedem Unifikator σ von s und t eine Substitution σ' gibt mit $\sigma = \sigma' \circ \mu$.

4.3 Resolution

Zunächst führen wir folgende zwei Grundbegriffe ein.

Definition 4.3.1 (Herbrand-Universum)**:**
Das Herbrand-Universum $D(F)$ zu einer Formel F besteht aus allen Termen, die sich aus den in F vorkommenden Konstantensymbolen und Funktionssymbolen bilden lassen. Falls F keine Konstantensymbole enthält, wird a als Konstantensymbol hinzugenommen.

Definition 4.3.2 (Herbrand-Expansion)**:**
Sei $F = \forall x_1 \cdots \forall x_k G$ eine geschlossene Skolemformel mit Matrix-Formel G. Die Herbrand-Expansion $E(F)$ von F sei die folgende Menge von PL-Formel:

$$\{G[x_1/t_1, \ldots, x_k/t_k] \mid t_1, \ldots, t_k \in D(F)\}$$

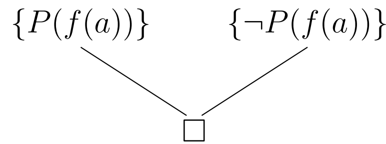
Das Ziel der Resolution ist es, einen elementaren Widerspruch (Feststellung der Unerfüllbarkeit) einer Formel abzuleiten. Dieser wird durch die sog. leere Klausel, dargestellt durch $\Box$, repräsentiert.
Ist $\Box$ aus einer Klauselmenge $\mathcal{K}$ ableitbar, so ist $\mathcal{K}$ unerfüllbar.

Satz 4.3.1 (von Gödel, Herbrand, Skolem)**:**
Eine geschlossene Skolemformel F ist genau dann erfüllbar, wenn $E(F)$ aussagenlogisch erfüllbar ist.

Wir sagen: $E(F)$ ist *aussagenlogisch erfüllbar*, wenn es eine Belegung der atomaren Formeln von $E(F)$ mit Wahrheitswerten gibt, die alle Formeln in $E(F)$ erfüllt.

Beispiel 4.3.1:

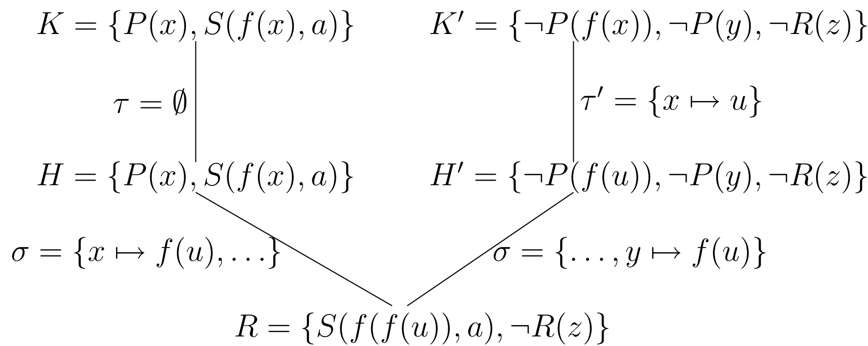
$$\begin{array}{lcl}
\textit{Sei } F & = & \forall x(P(x) \land \neg P(f(x))) \\
D(F) & = & \{a, f(a), f(f(a)), f(f(f(a))), \ldots\} \\
E(F) & = & \{P(a) \land \neg P(f(a)), \qquad \textit{wegen } x \mapsto a \\
 & & P(f(a)) \land \neg P(f(f(a))), \textit{ wegen } x \mapsto f(a) \\
 & & \ldots\}
\end{array}$$



Leider wird nicht immer so schnell eine unerfüllbare Klauselmenge erreicht. Dazu brauchen wir:

Definition 4.3.3 (PL Resolvente)**:**
Seien K, K' prädikatenlogische Klauseln, τ, τ' Variablenumbenennungen, so dass $H =_{def} \tau(K)$ und $H' =_{def} \tau'(K')$ keine Variablen gemeinsam haben.
Ferner seien $L_1, \ldots, L_k$ Literale in H und $L'_1, \ldots, L'_k$ Literale in H', so dass $\{\neg L_1, \ldots, \neg L_k, L'_1, \ldots, L'_m\}$ unifizierbar ist mit mgu σ.
Dann heißt $R =_{def} \sigma((H \setminus \{L_1, \ldots, _k\}) \cup (H' \setminus \{L'_1, \ldots, L'_m\}))$ prädikatenlogische Resolvente von K und K'

Beispiel 4.3.2:



Definition 4.3.4 (PL Resolution)**:**
Für PL-Klauselmengen $\mathcal{K}$ definieren wir:

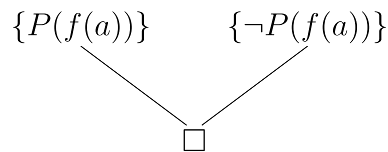
- $Res(\mathcal{K}) =_{def} \mathcal{K} \cup \{K \mid K \text{ ist PL-Resolvente zweier Klauseln aus } \mathcal{K}\}$
- $Res^0(\mathcal{K}) =_{def} \mathcal{K}$
- $Res^k(\mathcal{K}) =_{def} Res(Res^{k-1}(\mathcal{K}))$, *für alle* $k \geq 1$
- $Res^\infty(\mathcal{K}) =_{def} \bigcup_{k \geq 0} Res^k(\mathcal{K})$

Satz 4.3.2 (PL Resolutionssatz)**:**
Eine prädikatenlogische Formel F ist genau dann unerfüllbar, wenn für ihre Matrixklauselform $\mathcal{K}$ gilt:

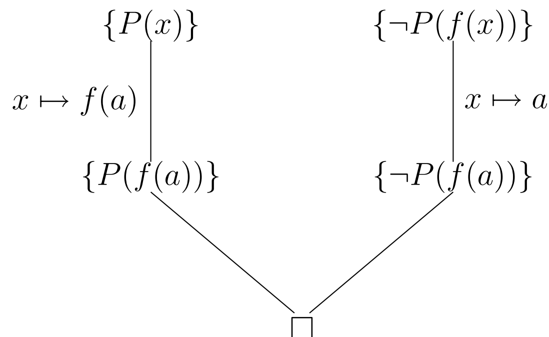
$$\Box \in Res^\infty(\mathcal{K})$$

Beispiel 4.3.3:
Wir zeigen Unerfüllbarkeit von $F = \forall x(P(x) \wedge \neg P(f(x)))$ *mithilfe der:*

- *Grundresolution:*
  - $D(F) = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$
  - $E(F) = \{P(a) \wedge \neg P(f(a)),$ *wegen* $x \mapsto a$
    $P(f(a)) \wedge \neg P(f(f(a))),$ *wegen* $x \mapsto f(a)$
    $\ldots\}$



- *PL Resolution: Matrixklauselform von* $F : \{P(x)\}, \{\neg P(f(x))\}$



Kapitel 5

Optimierung des statischen Zensors

5.1 Motivation

Stellen wir uns folgende Situation vor, wir haben eine Firmendatenbank, in der zu jedem Mitarbeiter der Name, die Abteilung und das Gehalt gespeichert sind:

Mitarbeiter	Name	Abteilung	Gehalt
	Smith	Handelsabteilung	2000
	Carsten	Handelsabteilung	2000

Abbildung 5.1: Mitarbeiter

Der Administrator will, dass das Gehalt von Smith geheim bleibt. Intuitiv denkt man, dass die Anfrage "Wieviel verdient Smith?" unbeantwortet bleiben muss. Allerdings wird leider die Information alleine dadurch nicht geschützt, da z.B. nach folgenden Anfragen:

"In welcher Abteilung arbeitet Smith?"
"In welcher Abteilung arbeitet Carsten?"
"Wieviel verdient Carsten ?"

der User die Schlussfolgerung ziehen kann, dass Smith auch 2000 Euro verdient, weil er auch in der Handelsabteilung arbeitet, wo man 2000 Euro Gehalt bekommt.

Dieses Beispiel zeigt, dass, obwohl die gegebenen Antworten “public information” waren, die Schlussfolgerung(Inference) dennoch geheim war. Das führt dazu, dass man die Anfragen beantworten muss, nachdem man ”viel“ überlegt hat und man sicher ist, dass die gegebene Information kein Geheimnis verrät. Diese ”vielen Überlegungen“ nennen wir Kontrollierte Anfrageauswertung.

5.2 Definitionen

Damit wir die Kontrollierte Anfrageauswertung überhaupt maschinell berechnen können, müssen wir das Problem mathematisch formulieren.

Definition 5.2.1 (Relationschema und Instanzen)**:**
Ein Relationschema wird durch ein Relationsymbol R, eine endliche Artributmenge $U = \{A_1, ...A_n\}$ *und eine endlichen Menge lokaler semantischer Bedingungen* Σ_{local} *bestimmt.*

$$RS = \langle R, U, \Sigma_{local} \rangle.$$

Const stellt eine unendliche Menge von Konstanten. Eine Instanz r des Relationsschema ist eine endliche Herbrand-Interpretation I des Schemas, wobei die Relationssymbole Prädikate sind und für die Konstanten $I(c) = c$ *gilt. Ein Tupel wird in der Darstellung* $\mu = R(a_1, ..., a_n)$ *angezeigt, wobei* $a_i \in Const$. *Wenn* μ *in der entsprechenden Datenbank enthalten ist, schreiben wir :*

$$r \models_M \mu$$

Anmerkung: Auf der Datenbankebene ist r die Menge von solchen μ, so dass r μ wahr macht.

$$r = \{\mu \mid r \models_M \mu\}$$

Definition 5.2.2 (Funktionale Abhängigkeit)**:**
Sei $A, B \subseteq U$ *eine Menge von Attributen und* μ_1, μ_2 *zwei beliebige Tupel. Eine Instanz r erfüllt die funktionale Abhängigkeit* $A \rightarrow B$, *wenn in r gilt:*

$$\mu_1(A) = \mu_2(A) \Longrightarrow \mu_1(B) = \mu_2(B)$$

Eine funktionale Abhängigkeit $A \rightarrow B$ heißt trivial, falls $B \subseteq A$.

Mitarbeiter	PesonalNr	Name	Abteilung	Gehalt
	16	Smith	Handelsabteilung	2000
	32	Carsten	Buchhaltung	3000
	64	Carsten	Handelsabteilung	2000

Abbildung 5.2: Tabelle 2. Mitarbeiter

Beispiel 5.2.1:
PersonalNr → Abteilung . Hier ist die Abteilung von der PersonalNr abhängig. Dagegen ist die funktionale Abhängigkeit $Name$ → Abteilung, also dass Abteilung von $Name$ abhängt, falsch.

Im Folgenden gehen wir davon aus, dass eine Datenbank vorliegt, die genau aus einem Relationsschema $RS = \langle R, U, \Sigma \rangle$ besteht, wobei Σ nur funktionale Abhängigkeit enthält. Das Datenbankschema ist durch RS und die Menge $Const$ definiert. Außerdem sei $A_i \in U$ $Const$.

Definition 5.2.3 (Anfragesprache)**:**
Die Anfragesprache L_q ist definiert durch:

$$L_q{:=}\{R(a_1, ..., a_n) \mid a_i \in Const\}.$$

Definition 5.2.4 (Gewöhnliche Anfrageauswertung)**:**
Die gewöhnliche Anfrageauswertung eval ordnet abhängig von der Datenbankinstanz r zur Anfrage $\Phi \in L_q$ den Wert true oder $false$ zu.

$$eval(\Phi) : DS \rightarrow \{true, false\} \text{ mit}$$
$$eval(\Phi)(r) := r \models_M \Phi.$$

Die alternative Version eval∗ ordnet Φ oder $\neg\Phi$ zu Φ :

$$eval^*(\Phi)(r) = \begin{cases} \Phi & \text{wenn } r \models_M \Phi \\ \neg\Phi & \text{sonst} \end{cases}$$

Definition 5.2.5 (Potential secrets)**:**
Sei Var eine unendliche Menge von Variablen. Die Sprache der potentiellen Geheimnisse ist definiert durch :

$$L_{ps} := \{\ (\exists X_1)\ (\exists X_2)\ ...\ (\exists X_l)\ R(v_1, ..., v_n) \mid 0 \leq l \leq n,$$
$$X_i \in Var,\ v_i \in Var \bigcup Const,\ \{X_i, ..., X_l\} \subseteq \{v_1, ...v_n\},$$
$$v_i \in Var \Longrightarrow v_i = X_j \text{ für ein } j \in \{1, ..., l\},$$
$$v_i, v_j \in Var \Longrightarrow v_i \neq v_j$$

Bemerkung. Offensichtlich ist L_q eine Untermenge von L_{ps}. Daher können beide Sprachen als L_{ps} bezeichnet werden. Ausserdem werden wir im Folgenden die Menge der potentiellen Geheimnisse mit *pot_sec* bezeichnen.(Vorsicht, nicht die Sprache!).

Beispiel 5.2.2:
Betrachte das Relationsschema $RS_5 = \langle R_5, \{A, B, C\}, \{A \rightarrow BC\}\rangle$ und die Instanz $r_5 = \{\mu_1, \mu_2, \mu_3\}$ mit $\mu_1 = \{R_5(a_1, b_1, c_1)\}$, $\mu_2 = \{R_5(a_2, b_1, c_2)\}$ und $\mu_3 = \{R_5(a_3, b_1, c_1)\}$. Ausserdem sei $pot_sec_5 = \{\psi_1, \psi_2\}$ mit $\psi_1 = (\exists X_1)\ R_5(X_1, b_1, c_1)$ und $\psi_2 = R_5(a_2, b_2, c_2)$. ψ_1 soll verhindern, dass der User die Kombination $B = b_1, C = c_1$ entdeckt. Die Durchsetzung der Sicherheitspolitik hält μ_1 und μ_3 und ihre kritischen Teile geheim, die zur Aufdeckung des Geheimnis beitragen könnten.

Definition 5.2.6 (Modifizierte Anfrageauswertung)**:**
Die Modifizierte Anfrageauswertung m_eval bildet eine Anfragesequenz $Q = \langle \Phi_1, \Phi_2, ..\rangle$, eine Datenbankinstanz r und eine Sicherheitspolitik pot_sec auf eine Antwortsequenz ab:

$$m_eval(Q)(r, pot_sec) = \langle ans_1, ans_2, ...\rangle.$$

Definition 5.2.7 (Sichere Anfrageauswertung)**:**
Gegeben sei eine (möglicherweise unendliche) Anfragesequenz $Q = \langle \Phi_1, \Phi_2, ..\rangle$ mit $\Phi_i \in L_q$ und eine Sicherheitspolitik $pot_sec = \{\Phi_1, ..., \Phi_m\}$ mit $\Psi_i \in L_q$. Die sichere Anfrageauswertung m_eval ist bezüglich pot_sec sicher, wenn für jede endliche Anfangsfolge Q' von Q , für jede $\Psi \in pot_sec$ und für jede Instanz r_1 von RS eine Instanz r_2 von RS existiert, die die folgenden Eigenschaften erfüllt:

1. $m_eval(Q')(r_1, pot_sec) = m_eval(Q')(r_2, pot_sec)$
2. $eval^*(\Psi)(r_2) = \neg\Psi$

m_eval heißt sicher, wenn es bezüglich jeder Menge pot_sec sicher ist.

5.3 Auf Sicherheit basierende Klassifikation der Datenbanken

Nun überlegen wir uns, wie wir anhand des bisher gesammelten Wissens eine effiziente Zugriffskontrolle durchsetzen können, ohne die Sicherheit von Datenbanken im Sinne von Definition 5.2.7 zu gefährden. Wir können ein neues Konzept Klassifikationsinstanz entwickeln.

Definition 5.3.1 (Classification schemas):
Das Klassifikationsschema vom Relationsschema $RS = \langle R, U, \Sigma \rangle$ *ist definiert durch* $RS^C = \langle R, U, \emptyset \rangle$ *und* $Const^C$ $=Const \bigcup \{\#\}$ *mit* $\# \in Const$*, wobei* S *ein neues Relationssymbol ist.*

Definition 5.3.2 (Classification instances):
Gegeben ist die Menge der potentiellen Geheimnisse pot_sec. Dann ist die Klassifikationsinstanz s bezüglich pot_sec folgenderweise definiert. Für jedes Element $\Psi = (\exists X_1)...(\exists X_l)R(v_1, ..., v_n)$ *aus pot_sec, enthält s ein Tupel* $S(v_1^*, ..., v_n^*)$ *so, dass* $v_i^* = v_i$*, wenn* $v_i \in Const$ *und* $v_i^* = \#$ *sonst. Ansonsten enthält s keine andere Formel.*

Beispiel 5.3.1:
Betrachten wir die potentiellen Geheimnisse von Beispiel 5.2.2.

$$pot_sec_5 = \{\psi_1, \psi_2\} \text{ mit } \psi_1 = (\exists X_1)\ R_5(X_1, b_1, c_1) \text{ und } \psi_2 = R_5(a_2, b_2, c_2)$$

Dann wird die entsprechende Klassifikationsinstanz s so aussehen:

$$s = \{\psi_1^*, \psi_2^*\} \text{ mit } \psi_1^* = S(\#, b_1, c_1) \text{ und } \psi_2^* = S(a_2, b_2, c_2)$$

Mit der Klassifikationsinstanz wird festgelegt, ob eine Anfrage erlaubt ist oder nicht. Intuitiv zeigt ein Element der Klassifikationsinstanz die Wertekombination von einem Geheimnis. Das Symbol # ist ein Platzhalter, ähnlich zu dem *null* Wert. Eigentlich existiert der Wert, aber er ist aus Sicherheitssicht irrelevant. Wenn die Konstanten einer Anfrage Φ identisch mit den Konstanten eines Tupel S der Klassifikationsinstanz sind, bedeutet das, dass diese Anfrage eine Geheimnis aufdecken könnte. Daher darf diese "nicht erlaubt" sein. Es können sogar partielle Matchings ausreichen, die Anfrage "nicht erlaubt" zu nennen: falls alle Konstanten von S (also bis auf #) in der Anfrage Φ enthälten sind, muss die Antwort abgelehnt werden,

ohne auf die restlichen Konstanten von Φ zu achten. z.B: $S = (a, b, \#)$ und $\Psi = (\exists X_1)\ R(a, b, c)$. Die Konstante c in Ψ spielt hier keine Rolle.
Um die erlaubten Anfragen formal zu definieren, brauchen wir noch eine Notation "Relevanz".

Definition 5.3.3 (Relevanz)**:**
Sei $\chi_1, \chi_2 \in L_{ps}$. $\chi_i(A)$ *bezeichnet den Wert der Attribute* A *in* χ. χ_1 *heißt relevant für* χ_2 *wenn für jedes* $A \in U$ *gilt:*

$$\chi_1(A) \in Const \Rightarrow \chi_1(A) = \chi_2(A)$$

Beispiel 5.3.2:
Gegeben: $\mu = (a, b, \#)$ *und* $\Psi = (\exists X_1)\ R(a, b, c)$. *Dann ist* μ *relevant für* Ψ.

Definition 5.3.4 (Erlaubte Anfragen)**:**
Gegeben ist eine Datenbankinstanz r und eine Menge von potentiellen Geheimnissen pot_sec. Eine Anfrage $\Phi = R(v_1, ..., v_n)$ *ist bezüglich r und pot_sec erlaubt, wenn es in der Klassifikationsinstanz (bezüglich r) kein Tupel* μ^* *gibt, das relevant für* Φ *ist.*

Und zum Schluss definieren wir "Access control"(Zugriffskontrolle), welche eine modifizierte Anfrageauswertung ist.

Definition 5.3.5 (Access control)**:**
Gegeben ist r als eine Instanz von RS, s als die Klassifikationsinstanz bezüglich pot_sec, und $Q = \langle \Phi_1, \Phi_2, ... \rangle$ *eine (möglicherweise unendliche) Anfragesequenz mit* $\Phi_i \in L_q$. *Die Zugriffskontrollfunktion ac ist definiert durch:*
$ac(Q)(r, pot_sec) := \langle ans_1, ans_2, ..., \rangle$ *mit*

$$ans_i := \begin{cases} eval^*(\Phi_i)(r) & \Phi_i \text{ ist erlaubt bezüglich } pot_sec \\ mum & sonst \end{cases}$$

Man muss beachten, dass die Zugriffskontrollfunktion instanzunabhängig ist, und jedes potentielle Geheimnis $\Psi \in pot_sec$ schützt, unabhängig davon, ob Ψ in der Datenbank enthalten ist oder nicht.

Beispiel 5.3.3:
Betrachten wir wieder das Beispiel 5.2.2 und die Klassifkationsinstanz s vom Beispiel 5.3.1, also gegeben sind:

$r_5 = \{\mu_1, \mu_2, \mu_3\}$ *mit* $\mu_1 = \{R_5(a_1, b_1, c_1)\}$, $\mu_2 = \{R_5(a_2, b_1, c_2)\}$ *und* $\mu_3 = \{R_5(a_3, b_1, c_1)\}$.

$s = \{\psi_1^*, \psi_2^*\}$ *mit* $\psi_1^* = S(\#, b_1, c_1)$ *und* $\psi_2^* = S(a_2, b_2, c_2)$.
Wir untersuchen die Anfragesequenz $Q_1 = \langle \Phi_1, \Phi_2, \Phi_3 \rangle$ *mit* $\Phi_1 = R_5(a_4, b_1, c_1)$, $\Phi_2 = R_5(a_2, b_1, c_2)$ *und* $\Phi_3 = R_5(a_2, b_2, c_2)$. *Laut Definition 5.3.4 sind* Φ_1 *und* Φ_3 *nicht erlaubt, da die konstanten Variablen von* ψ_1^* *und* ψ_2^* *jeweils mit* Φ_1 *und* Φ_3 *identisch sind. Eigentlich ist* Φ_1 *nicht in* r*, aber das spielt hier keine Rolle, da* Φ_1 *den kritischen Teil, nämlich die Kombination* $(B = b_1, C = c_1)$ *enthält, reicht es völlig aus, dass* $VarPhi_1$ *nicht erlaubt ist.* Φ_3 *ist in* r*, und muss wegen* μ_2^* *geheim bleiben. Und zuletzt ist* Φ_2 *in* r *enthalten und ist erlaubt bezüglich* pot_sec*. Insgesamt sieht die Antwortsequenz so aus:* $\langle mum, R_5(a_2, b_1, c_2), mum \rangle$.

5.4 Algorithmen

Nun kann mit Hilfe dieser Definitionen beschrieben werden, wie man die Zugriffskontrolle effizient einsetzen kann. Wir formulieren das Problem nochmal kurz um. Stellen wir uns vor, dass wir eine Datenbank mit n Attributen, und eine Geheimnismenge mit m Einträgen haben. Wir wollen wissen, ob unsere Geheimnismenge pot_sec "etwas ähnliches" zur gestellten Anfrage Φ enthält. Genauer gesagt, ob pot_sec irgendein Element enthält, das relevant(Definition 5.3.3) zu Φ ist. Es gibt folgende Algorithmen:

Gegeben ist eine Anfrage $\Phi \in L_q$, eine Datenbankinstanz r von RS, eine Geheimnismenge pot_sec, und die Klassifikationsinstanz s. Außerdem sei n die Anzahl von Attributen von RS, $m = |pot_sec|$ und $|r|$ die Anzahl von Tupeln, die in r wahr sind. Wir können n als konstant annehmen.

5.4.1 Linearzeit-Algorithmus

Für jedes μ^* in s teste, ob alle Attribute von μ^*, die konstante Werte haben, die gleichen Werte wie in Φ haben.

Laufzeitanalyse

Laufzeit(Linearzeit-Algorithmus) = $\mathcal{O}($ (Anzahl von geheimen Elementen) * (Rechenschritte für ein Element) $) = \mathcal{O}(m * n)$.

Bevor wir mit dem zweiten Algorithmus anfangen, lernen wir einen Datenstruktur, nämlich den B-Baum kennen, den wir gebrauchen werden.

B-Baum. Ein Baum der Ordnung $m \geq 3$ heißt B-Baum, wenn er die folgenden Eigenschaften erfüllt:

1. Jeder Knoten mit Ausnahme der Wurzel enthält mindestens $\lceil m/2 \rceil$-1 Daten.
2. Jeder Knoten hat höchstens $m - 1$ Daten.
3. Die Daten sind sortiert.
4. Knoten mit k Daten $x_1, ..., x_k$ haben k + 1 Zeiger, die auf die Bereiche $(\cdot, x_1)$, $(x_1, x_2), ..., (x_{k-1}, x_k)$, $(x_k, \cdot)$ zeigen.
5. Die Zeiger, die einen Knoten verlassen, sind entweder alle nil-Zeiger oder alle echte Zeiger.
6. Alle Blätter haben die gleiche Tiefe.

5.4.2 Logarithmische Zeit-Algorithmus

Führe folgende Operationen durch:

1. In der Preprocessingphase ordne die Tupel von s in einen B-Baum ein;
2. Erzeuge für jede Anfrage $R(a_1, ..., a_n)$ eine neue Menge: $P_{R(a_1,...,a_n)} = \{R(x_1, ..., x_n) \mid x_i \in \{\#, a_i\}$ mit $1 \leq i \leq n\}$;
3. Prüfe mit Hilfe der Baumsuche für jedes $\rho \in P_{R(a_1,...,a_n)}$, ob $\rho \in s$ ist. Wenn irgendein $\rho \in s$ ist, lehne die Antwort ab, sonst gehe zu 4.
4. Gib $eval^*(R(a_1, ..., a_n))(r)$ zurück.

Laufzeit(RT) des Logarithmische Zeit-Algorithmus

Wir bezeichnen die Laufzeit für die i-te Operation mit RT_i. Dann gilt:

$RT = RT_2 + RT_3 + RT_4$.

- $RT_1 = \mathcal{O}($ (Anzahl von geheimen Elementen) * Einfügen pro Element $) = \mathcal{O}(m * log(m))$.
- $RT_2 = \mathcal{O}($ (Anzahl der Elemente in $P_{R(a_1,\ldots,a_n)}$) $) = \mathcal{O}(2^n)$.
  Weil die Elemente in $P_{R(a_1,\ldots,a_n)}$ nur 2 Werte annehmen dürfen, entweder a_i oder #. Das kann man mit n-bit Binärzahlen vergleichen. Und da n-bit Binärzahlen 2^n verschiedene Werte haben können, hat $P_{R(a_1,\ldots,a_n)}$ auch diese Größe.
- $RT_3 = \mathcal{O}($ (Anzahl der Elemente in $P_{R(a_1,\ldots,a_n)}$) * $Tiefe_{B-Baum}$ * (Anzahl der Elemente von einem B-Baum Knoten)).
  Angenommen der B-Baum hat die Ordnung $2k$. Dann hat der Baumknoten zwischen k und $2k-1$ Daten und eine Tiefe von höchstens $log_k((m+1)/2)$. Da k eine Konstante ist, folgt:
  $RT_3 = \mathcal{O}(\ 2^n * log_k((m+1)/2) * k\ ) = \mathcal{O}(\ 2^n * log(m)\ )$
- RT_4: Wir können analog dazu auch die Tupel von r in einen B-Baum einordnen. Dann kostet auch die gewöhnliche Anfrageauswertung $log_{|r|}$ Zeit.

Insgesamt kostet der Logarithmische Zeit-Algorithmus ohne RT_1 (wegen der Preprocessingphase) :

$RT = \mathcal{O}(2^n) + \mathcal{O}(\ 2^n * log(m)\ ) + log(|r|) = \mathcal{O}(\ 2^n * log(m) + log(|r|)$ Rechenschritte.

5.4.3 Effiziente Berechenbarkeit von Zugriffskontrolle

Die Zugriffskontrollfunktion $ac(\langle\Phi\rangle)(r, pot_sec)$ kann in $\mathcal{O}(min\{2^n * log(m), m * n\}$ $+ log(|r|))$ Rechenschritten berechnet werden.

5.5 Korrektheit

Lemma 5.5.1:
Seien χ und $\chi_1, ..., \chi_n$ in L_{ps}. Dann sind die folgenden Bedingungen äquivalent:

1. $\{\chi_1, ..., \chi_n\} \models \chi$
2. $\exists \chi_i$ *mit* $\chi_i \in \{\chi_1, ..., \chi_n\}$, *so dass* χ *relevant für* χ_i *ist.*

Beweis:

- " $\Longrightarrow$ ":
  Angenommen χ ist für keine $\chi_i \in \{\chi_1, ..., \chi_n\}$ relevant. Das heißt für jede Belegung β gilt:

  $$\beta(dq(\chi_i)) \neq \beta(dq(\chi))$$

  Dann können wir ein Zeugeninstanz r erzeugen:

  $$r = \{\beta(dq(\chi_1)), ..., \beta(dq(\chi_n)).$$

  Nun heißt das, dass :

  $$r \models_M \{\chi_1, ..., \chi_n\} \text{ und } r \not\models_m \chi.$$

  Aber das widerspricht der Bedingung $\{\chi_1, ..., \chi_n\} \models \chi$.

- " $\Longleftarrow$ "
  Sei χ_i ein Element, für das χ relevant ist. Das heißt χ und χ_i sehen folgendermaßen aus:

  $$\begin{aligned} \chi: \quad & (\exists X_1)...(\exists X_l)(\exists X_{l+1})R(v_1, ..., v_{j-1}, X_{l+1}, v_{j+1}, ..., v_n) \\ \chi_i: \quad & (\exists X_1)...(\exists X_l)R(v_1, ..., v_{j-1}, v_j, v_{j+1}, ..., v_n) \end{aligned}$$

  Wobei R ein Relationssymbol, $v_i \in Const$, $X_{l+1} \in Var$, und $X_{l+1} \notin \{X_1, ..., X_l\}$ ist. Nach Definition des Existenzquantors gilt:

$$(\exists X_1)...(\exists X_l)R(v_1, ..., v_{j-1}, v_j, v_{j+1}, ..., v_n) \models$$
$$(\exists X_1)...(\exists X_l)(\exists X_{l+1})R(v_1, ..., v_{j-1}, X_{l+1}, v_{j+1}, ..., v_n)$$

Es folgt: $\chi_i \models \chi$ und infolge dessen $\{\chi_1, ..., \chi_n\} \models \chi$. ■

Lemma 5.5.2:
Sei S eine endliche, konsistente Menge von Formeln mit der Partition $\{S_1, S_2, S_3\}$, wobei:

1. $\varphi \in S_1 \implies \varphi \in L_q$
2. $\varphi \in S_2 \implies \varphi = \neg\varphi'$ *mit* $\varphi' \in L_q$
3. $\varphi \in S_3 \implies \varphi$ *ist* fd

Dann sind für jedes Ψ die folgenden Bedingungen äquivalent:

1. $S \models \Psi$
2. *Es gibt ein $\chi \in L_q$ mit $\chi \in S$ und $\chi \models \Psi$*

Beweis:

- $'' \implies ''$
  Wir beweisen diese Richtung mit Hilfe der Kontraposition :

$$\underbrace{S \models \Psi}_{1)} \implies \underbrace{\text{es existiert ein } \chi \in L_q \text{ mit } \chi \in S \text{ und } \chi \models \Psi}_{2)}$$

Kontraposition:

$$\text{nicht (es existiert ein } \chi \in L_q \text{ mit } \chi \in S \text{ und } \chi \models \Psi) \implies \text{nicht } (S \models \Psi)$$
$$\underbrace{\text{für alle } \chi \in L_q \text{ gilt } \chi \notin S \text{ oder } \chi \not\models \Psi}_{\neg 2)} \implies \underbrace{S \not\models \Psi}_{\neg 1)}$$

$\neg 2)$ heißt: $\forall \chi \in L_q$ gilt: $\chi \notin S$ oder ($\chi \in S$ und $\chi \not\models \Psi$)

$\neg 1)$ heißt: es gibt ein r, so dass:

$$r \models_M S \text{ und } r \not\models_M \Psi.$$

Um das zu zeigen, erzeugen wir eine Zeugeninstanz r, wobei r eine Herbrand-Interpretation ist:

$$r := S_1.$$

Da S eine endliche Menge ist, wäre folgendes möglich:

$$r \models_M S.$$

Da S drei Arten von Formeln enthält, müssen wir dies für alle drei Formeln beweisen.

1. (S_1): Wegen der Konstruktion gilt: $r \models_M S_1$.
2. (S_2): S_2 enthält nur die negativen Formeln. Da S konsistent ist, gibt es keine Formel in S_1, dessen negatives Komplement in S_2 ist. D.h. keine Formel in S_2 widerspricht einer Formel in S_1, es folgt: $r \models_M S_2$.
3. (S_3): S_1 besteht aus den Tupeln $\mu_1, ..., \mu_k$. Das heißt $r \models_M \mu_1, ..., r \models_M \mu_k$. Nun bedeutet $\varphi \in S_3$ mit $r \not\models_M \varphi$, dass die von φ vertretene funktionale Abhängigkeit $A \longrightarrow B$ in r verletzt ist. Das heißt zwei Tupel μ_1, μ_2 aus S_1, die in A übereinstimmen, sind in B verschieden. Daraus folgt, dass die Menge $\{\mu_1, \mu_2, \varphi\}$ inkonsistent ist. Aber da diese Menge zu S gehört, dürfte dies nicht passieren, weil S nach Annahme eine konsistente Menge ist und eine konsistente Menge keine inkonsistenten Teilmengen enthalten darf. Insgesamt folgt: $r \models_M S_3$.

Ferner gilt nach $\neg 2$), $\forall \varphi \in S_1 \quad \varphi \not\models \Psi$. Weil S_1 konsistent ist und keine fd enthält, ist keine Schlussfolgerung möglich. Das führt zu $S_1 \not\models \Psi$. Nach Lemma 5.5.1 und Definition 5.3.3 folgt, dass es für jedes $\varphi \in S_1$ ein Attribut A mit $\Psi(A) \in Const$ und $\varphi(A) \neq \Psi(A)$ gibt. Da $r := S_1$ ist, folgt $r \not\models_M \Psi$.

- $'' \Longleftarrow ''$

  Offensichtlich, da aus $\chi \models \Psi$ und $\chi \in S$ folgt $S \models \Psi$. ■

Theorem 5.5.1:
Sei r eine Instanz des RS, pot_sec $\subset L_{ps}$ eine Sicherheitspolitik, $Q = \langle \Phi_1, \Phi_2 ... \rangle$

mit $\Phi \in L_q$ eine (möglicherweise unendliche) Anfragesequenz, s eine Klassifikationsinstanz bezüglich pot_sec und ac eine Zugriffskontrollfunktion, entsprechend Definition 5.3.5. Dann ist ac sicher bezüglich pot_sec im Sinne der Definition 5.2.7.

Beweis Sei $Q' = \langle \Phi_1, ..., \Phi_n \rangle$ ein endliches Präfix von Q und Ψ ein Element der Menge *pot_sec*. Wir zeigen die Existenz einer Instanz r' eines RS. In r' gilt Σ und erlaubte Anfragen von Q' sind $true$, aber Ψ ist $false$. Wir erzeugen eine Menge log:

$$log := \Sigma \ \cup \{ans_i | i \in \{1, ..., n\}, ans_i \neq mum\}.$$

Es ist zu zeigen, dass nach der letzter Anfrage Φ_n, das Geheimnis Ψ nicht vom log impliziert wird. Wir beweisen dies durch Widerspruch und nehmen an:

$$log \models \Psi \ .$$

Nach Lemma 5.5.2 sollte es dann ein $ans_i \in log$ geben, so dass:

$$ans_i \models \Psi$$

Laut Definition 5.3.5 ist $ans_i = eval^*(\Phi_i)(r)$. Daraus folgt:

$$eval^*(\Phi_i)(r) \models \Psi$$

Laut Definition 5.2.4 ist $eval^*(\Phi_i)(r) \in \{\neg\Phi, \Phi\}$. Daher haben wir zwei Fallunterscheidungen. Wegen der Struktur von Φ und Ψ ist $\neg\Phi \models \Psi$ ausgeschlossen. Es bleibt dann nur ein Fall:

$$\Phi_i \models \Psi$$

Nach Definition 5.3.2 enthält s dann ein Tupel μ, so dass μ relevant für Φ_i ist. Das heißt Φ_i ist bezüglich *pot_sec* nicht erlaubt. Dann sollte aber $ans_i = mum$ sein. Da das log kein mum enthält, folgt:

$$log \not\models \Psi.$$

Daraus folgt, dass es eine Instanz r' mit $r' \models_M log$ und $r' \not\models_M \Psi$ gibt. Nach Definition des logs gilt: $log \models ans_i$ für alle $ans_i \neq mum.$

$\Rightarrow$ $r' \models ans_i$ für alle $ans_i \neq mum.$

Da die Zugriffskontrolle unabhängig von der Instanz ist:

$$ac(\Phi_i)(r, pot_sec) = mum \text{ iff } ac(\Phi_i)(r', pot_sec) = mum.$$

Die Zugriffskontrollfunktion gibt entweder die richtige Antwort für die Anfrage zurück oder lehnt diese ab. Insgesamt sind die Bedingungen von Definition 5.2.7 erfüllt.

$$ac(\Phi_i)(r, pot_sec) = ac(\Phi_i)(r', pot_sec)$$
$$eval^*(\Phi_i)(r') = \neg\Phi$$

Außerdem erfüllt r' auch Σ und damit RS. ■

5.6 Kritik

Wir haben Algorithmen kennengelernt, die, ohne die Sicherheit von Datenbanken zu gefährden, die Zugriffskontrolle effizient berechnen können. Mit dem oben beschriebenen Ansatz kann man auf dynamische Inferenzkontrolle verzichten und stattdessen eine statische Zugriffskontrolle einsetzen. Allerdings hat dieser Lösungsansatz auch einen großen Nachteil. Er funktioniert nur korrekt wegen der Anfragesprache L_q, weil L_q viele Restriktionen enthält, so darf z.B. der Benutzer keine komplexen Anfragen oder Anfragen mit freien Variablen eingeben. Das ist aus dem Blickwinkel des Datenbankbenutzers nicht akzeptabel, da die Datenverfügbarkeit dadurch schlechter wird. Wir werden dies nun mit ein paar Beispielen veranschaulichen.

Beispiel 5.6.1:
Stellen wir uns vor, der Benutzer stellt eine komplexe Anfrage $\Phi_3 = \Phi_1 \bigvee \Phi_2$*, wobei* $\Phi_1, \Phi_2 \in L_q$ *gilt. Außerdem darf der Benutzer den Wert von* Φ_1 *in der Datenbank nicht wissen. Aber es kann ihm manchmal gelingen das Geheimnis aufzudecken: falls* Φ_3 *den Wert* $false$ *hat, kann der Benutzer die Schlussfolgerung ziehen, dass* Φ_1 *auch* $false$ *ist.*

Wir haben die Anfragesprache L_q so definiert, dass Benutzer keine Anfragen stellen dürfen, die Variablen enthälten. Überlegen wir uns, was ohne diese Restriktion passieren würde.

Beispiel 5.6.2:
Gegeben sei eine alternative Anfragesprache $L_q^{\exists} := L_{ps}$, *Relationsschema* $RS = \langle R, \{A, B, C\}, \{A \rightarrow BC\} \rangle$ *mit einer Datenbankinstanz* $r = \{R(a_1, b_1, c_1)\}$ *und eine Sicherheitspolitik* $pot_sec = \{(\exists X)R(X, b_1, c_1)\}$. *Wir bekommen dann dementsprechend eine Klassifikationsinstanz* $s = \{S(\#, b_1, c_1)\}$.

Betrachten wir die Anfragesequenz $Q = \langle\ \Phi_1\ \Phi_2\ \rangle$, *wobei* $\Phi_1 = (\exists X)\ R(a_1, b_1, X)$ *und* $\Phi_2 = (\exists X)\ R(a_1, X, c_1)$ *ist. Laut Definition 5.3.4 sind die beiden Anfragen erlaubt. Folglich wird der Benutzer* $\langle\ (\exists X)\ R(a_1, b_1, X),\ (\exists X)\ R(a_1, X, c_1)\ \rangle$ *als Antwortsequenz bekommen. Leider kann der Benutzer mit Hilfe der funktionalen Abhängigkeit* $A \rightarrow BC$ *schlußfolgern, dass* $R(a_1, b_1, c_1)$ *in r true ist, damit wäre ein potentielles Geheimnis verraten.*

Kapitel 6

JDBC

6.1 Einführung

Falls Sie von Java heraus mit Datenbanken arbeiten möchten, so sind Sie bei **Java Database Connectivity** genannt **JDBC** an der richtigen Adresse gelandet. Damit können Sie auf die meisten Relationalen Datenbanken zugreifen, Hierfür ist es natürlich sinnvoll, sich mit Datenbankdesign auseinanderzusetzen und sich die Datenbanksprache SQL anzuschauen. Letztere vereint folgende Datenbanksprachen:

- Data Manipulation Language(DML): Daten lesen, schreiben, ändern und löschen.
- Data Definition Language(DDL): Datenbankbeschreibungssprache, um Datenstrukturen und verwandte Elemente zu beschreiben, ändern oder zu entfernen.
- Data Control Language(DCL): Datenüberwachungssprache; wird verwendet um Berechtigungen zu vergeben oder zu entziehen.

6.2 JAVA DataBase Connectivity

JDBC (Java Database Connectivity) ist ein Java-API (Application Programming Interface) zur Ausführung von SQL-Anweisungen innerhalb von Java-Applikationen

und Java-Applets. Es besteht aus einer Menge von Klassen und Schnittstellen, die in der Programmiersprache Java geschrieben sind.

Ein JDBC-Programm läuft in drei Phasen ab:

1. Treiber laden und Verbindung zur Datenbank aufbauen(Connection),
2. SQL-Anweisungen absenden,
3. Ergebnisse verarbeiten.

6.3 JDBC-Treibertypen

Es gibt insgesamt vier Treibertypen:

6.3.1 Treibertyp 1

- **JDBC-ODBC-Bridge** Es handelt sich hierbei um einen ODBC-Treiber, Java kommuniziert über eine sogennante JDBC-ODBC-Bridge mit der Datenbank. Diese wird von Sun mitgeliefert. Die verlangt, dass alle JDBC-Treiberhersteller mindestens SQL-2 Entry-Level-Standard von 1992 erfüllen. Ist auf dem Cli-

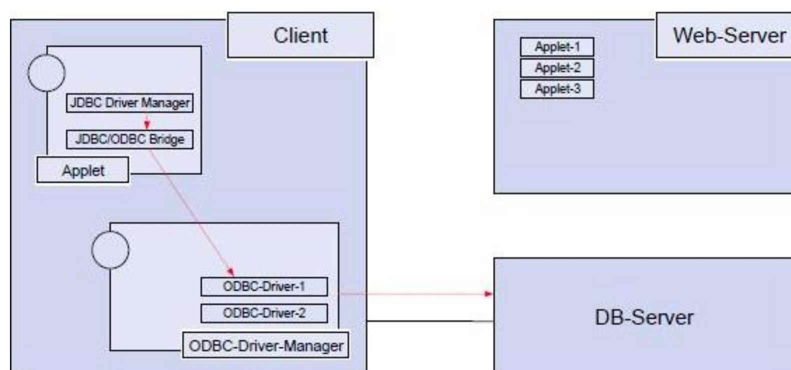


Abbildung 6.1: Treibertyp 1

ent eine ODBC-Datenquelle eingerichtet, so kann auch unter Verwendung des

von Sun mitgelieferten Brückentreibers der Kontakt zur Datenbank hergestellt werden:
String url = "jdbc:odbc:dbs"; // URL der Datenquelle
Class.forName(ßun.jdbc.odbc.JdbcOdbcDriver"); // Treiber

6.3.2 Treibertyp 2

- **Plattformeigene Treiber: native-API partly JAVA driver** Hier ist der Treiber selbst in Java geschrieben, greift jedoch auf einen Datenbankspezifischen und plattformabhängigen Treiber zurück.

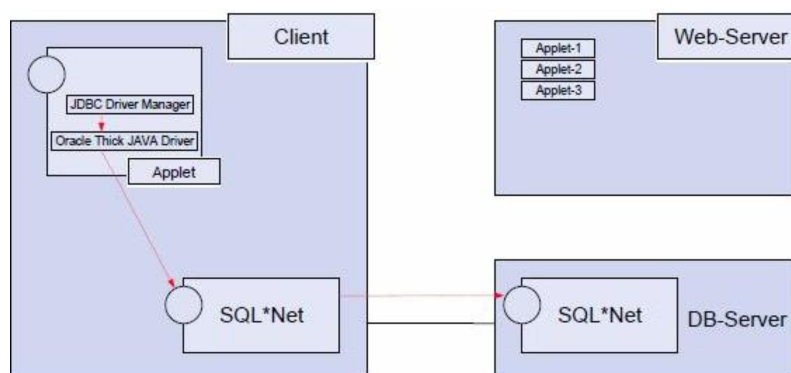


Abbildung 6.2: Treibertyp 2

6.3.3 Treibertyp 3

- **Universelle Treiber: net-protocol fully JAVA driver** Auch hier ist der Treiber komplett in Java geschrieben, es gibt aber eine Zwischenschicht, zwischen der Datenbank und dem Treiber. Dieses Verfahren wird sehr häufig eingesetzt und ist in vielem schneller als Typ1 und Typ2.

6.3.4 Treibertyp 4

- **Direkte JDBC-Treiber:native-protocol fully JAVA driver** Hat auch einen komplett in Java geschriebenen Treiber, und kommuniziert selbst mit der Datenbank über eine Kommunikationsschnittstelle. Die Zwischenschicht

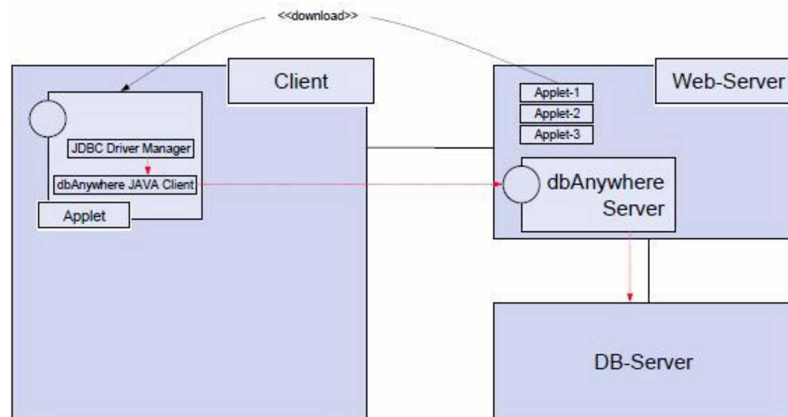


Abbildung 6.3: Treibertyp 3

existiert an dieser Stelle nicht mehr. Und damit ist dieser Treiber am performantesten.

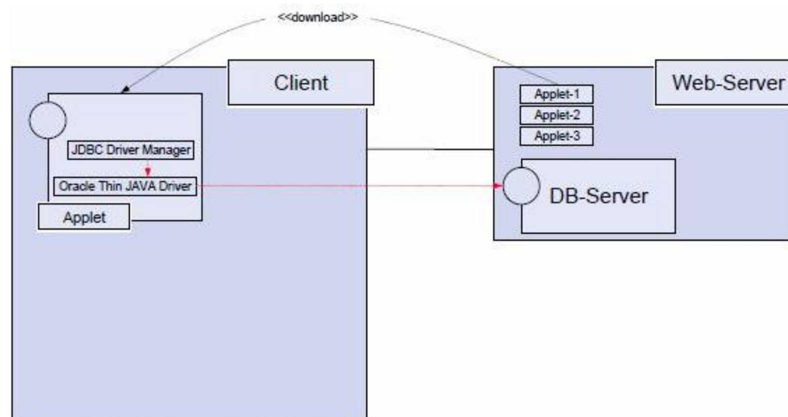


Abbildung 6.4: Treibertyp 4

6.4 Prepared Statements

Soll ein Update oder Delete durchgeführt werden, so wird statt der Methode *executeQuery* die Methode *executeUpdate* bemüht. Sie liefert die Zahl der geänderten Tupel zurück:

Statement stmt = con.createStatement(); // Statement
sqlStr = "UPDATE Person SET VNAME = 'Michael', LNAME = 'Ballack', PLZ = '22388', ORT = 'Dortmund' WHERE PID = 3309";
res = s.executeUpdate(sqlStr);

Ein Problem ist, dass Anweisungen (Statements) immer wieder neu übersetzt werden müssen bei mehrfacher Benutzung desselben SQL-Befehls. Hier ist PreparedStatements als effizientere Alternative möglich. Hierbei kann durch das Fragezeichen ('?') ein Platzhalter definiert werden.

Beispiel 6.4.1:

String insStr = "INSERT INTO Person VALUES(?,?)");
PreparedStatement updateStmt;
updateStmt = con.prepareStatement(insStr);
...
updateStmt.setInt(1, 3309); // Parameter werden übergeben
updateStmt.setString(2, Michael); // Parameter werden übergeben
...
int i = updateStmt.executeUpdate(); //Statement wird ausgeführt

JDBC/SQL Typ JAVA Typ	JDBC/SQL Typ JAVA Typ
CHAR,VARCHAR,LONGVARCHAR	java.lang.String
NUMERIC,DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT,DOUBLE	double
BINARY,VARBINARY,LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Abbildung 6.5: Abbildung von JDBC-/SQL-Datentypen in JAVA

6.5 Transaktionen

6.5.1 Begriffe

Unter einer *Transaktion* versteht man die Bündelung mehrerer Datenbankoperationen zu einer Einheit.

Zur Steuerung der Transaktionsverwaltung sind folgende Operationen notwendig:

- **begin of transaction (BOT)**: bezeichnet den Anfang einer Transaktion.
- **commit**: bezeichnet das Ende einer Transaktion. Alle Änderungen seit dem letzten BOT werden festgeschrieben.
- **abort**: bezeichnet den Abbruch einer Transaktion.
- **define savepoint**: bezeichnet einen zusätzlichen Sicherungspunkt
- **backup transaction**: setzt die Datenbasis auf den jüngsten Sicherungspunkt zurück.

im Fall einer erfolgreichen Transaktion, mit dem Befehl

```
BEGIN TRANSACTION Anweisung1;Anweisung2;Anweisung3;
```

Ob eine Transaktion Erfolg hat oder nicht wird erst während der Ausführung deutlich so erfolgt der Abschluss mit :

```
commit
```

Wird die Transaktion nicht korrekt ausgeführt, so hat man eine Sequenz der Form:

```
BEGIN TRANSACTION Anweisung1;Anweisung2;Anweisung3; abort
```

6.5.2 Eigenschaften von Transaktionen

Für Transaktionen gelten die folgenden sogenannten ACID-Eigenschaften:

- Atomicity: Eine Transaktion wird entweder vollständig ausgeführt oder überhaupt nicht ausgeführt.
- Consistency: Die Transaktion führt von einem konsistenten Zustand der Datenbank zu einem anderen konsistenen Zustand.
- Isolation: Für Transaktionen sind keine Daten sichtbar, die von anderen Transaktionen geschrieben wurden, wenn diese nicht den commit-Zustand erreichen.
- Durability: Wenn eine Transaktion den Zustand der Datenbank geändert hat und in den commit-Zustand eingetreten ist, dürfen die Änderungen nicht aufgrund eines später auftretenden Fehlers verloren gehen.

6.5.3 Transaktionssteuerung

Änderungen am Datenbestand während einer Sitzung werden normalerweise nicht direkt im Datenbestand vorgenommen, sondern gesammelt und erst am Ende der Transaktion durch die Methode **commit()** weggeschrieben oder durch **rollback()** rückgängig gemacht. Beides sind Methoden des Interfaces **java.sql.Connection**.

Sitzungen können auch im **Autocommit-Modus** sein, d.h. jede Änderung am Datenbestand wird sofort weggeschrieben. Dies entspricht einem impliziten commit nach jeder Anweisung. Manche Datenbanksysteme (z.B. MySQL) unterstützen nur den Autocommit-Modus, d.h. hier sind keine echten Transaktionen möglich.

6.6 Anwendungsbeispiel

Wir werden jetzt mit folgendem Beispiel das theoretische Wissen über JDBC praktisch werden lassen.

Ich habe bereits auf meinem Betriebssystem ein Mysql-Datenbanksystem installiert und konfiguriert. Jetzt möchte ich aus Java heraus mit dieser Datenbank kommunizieren. Hierzu brauche ich den passenden Datenbanktreiber für die Mysql-Datenbank, welche auf der Seite von Mysql als Typ4-Treiber existiert. Dafür habe ich ein Projekt in NetBeans mit dem Namen Datenbank angelegt und danach kann man in den Einstellungen das Jar-File hinzufügen. Das Vornehmen der Einstellungen kann von einer Entwicklungsumgebung zu einer anderen verschieden sein.

Nachdem es in unserem Projekt geladen worden ist, kann man die darin entaltenen Pakete sehen. Als nächstes habe ich eine Klasse mit Namen Connect erstellt. und damit sind wir bereit unsere Datenbankverbindung aufzubauen.

6.6.1 Java-Quellcode

```
import java.sql.*;

public class Connect {

public static void main(String[] args) {

// Laden des Treibers per Klassen-Loader

try {

Class.forName("com.mysql.jdbc.Driver");

} catch (ClassNotFoundException e) {

System.out.println("Kann den Treiber nicht laden!"); }

// Verbindungs-Objekt erzeugen und konfigurieren

Connection con = null;

String db = "jdbc:mysql://localhost/projektgruppe"

String user = "root";
```

```
String pass = ;
try {
con = DriverManager.getConnection(db, user, pass);
} catch (SQLException e) {
System.out.println("Verbindung fehlgeschlagen!");
}
Statement state = null;
try {
state = con.createStatement();
} catch (SQLException e) {
System.out.println("Konnte kein Statement erzeugen"); }
String sql = SSELECT * FROM studenten";
try {
ResultSet res = state.executeQuery(sql);
while (res.next()) {
String name = res.getString("Name");
String vorname = res.getString("Vorname");
System.out.println(name +  + vorname); }
} catch (SQLException e) {
System.out.println("Die Query ist fehlerhaft"); }
} }
```

6.6.2 Datenbanktabelle

Für das Anwendungsbeispiel wird folgende Datenbank verwendet:

– – Datenbank: ‘projektgruppe‘ –

– ———————————————

– – Tabellenstruktur für Tabelle ‘studenten‘ –

CREATE TABLE IF NOT EXISTS ‘studenten‘ (

‘ID‘ int(50) NOT NULL auto- increment,

‘Name‘ varchar(50) NOT NULL,

‘Vorname‘ varchar(50) NOT NULL,

PRIMARY KEY (‘ID‘)

) ENGINE=MyISAM DEFAULT CHARSET=latin1;

– – Daten für Tabelle ‘studenten‘ –

INSERT INTO ‘studenten‘ (‘ID‘, ‘Name‘, ‘Vorname‘) VALUES

(1, ’Erguig’, ’Mohammed’),

(2, ’Bui’, ’Binh’),

(3, ’Sarah’, ’Stahl’),

(4, ’Jabrailov’, ’Adalat’),

(5, ’Teufel’, ’Marcus’),

(6, ’Benet’, ’Lorenzo’),

(7, ’Ndula’, ’Peter’),

(9, 'Schlotmann', 'Thorsten'),

(10, 'Hui', 'Zheng');

Kapitel 7

Oracle - Virtual Private Database

7.1 Oracle Virtual Private Database

7.1.1 PL/SQL

Einführung

PL/SQL ist eine Oracle-spezifische prozedurale Programmiersprache. PL/SQL dient der Programmierung datenbanknaher Anwendungen direkt in der Datenbank. Dieses bewirkt eine größere Effizienz bei der Softwareentwicklung, denn mittels einer PL/SQL-Engine innerhalb der Datenbank wird ein schneller indirekter Zugriff auf Datenbankobjekte ermöglicht. PL/SQL ermöglicht im Gegensatz zu SQL eine Weiterverarbeitung von Anfrageergebnissen, welches insbesondere für Anwendungen die mit der Datenbank kommunizieren sehr effektiv ist, da der Netzwerkverkehr erheblich sinkt.

Syntax

Im folgenden wird die PL/SQL-Syntax erläutert.

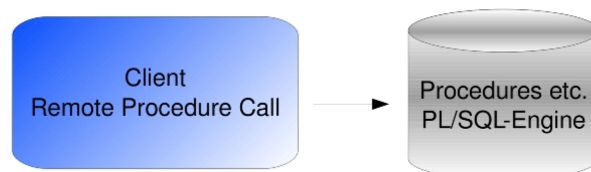


Abbildung 7.1: Compiler und Laufzeitsystem innerhalb des Oracle-DBMS.

PL/SQL Blockstruktur Die PL/SQL Programmierung erfolgt innerhalb einer Blockstruktur. Kopf, Deklarationsabschnitt, Ausführungsabschnitt, und Ausnahmeabschnitt. Der Gesamt-Code wird immer zwischen BEGIN und END eingebettet. PL/SQL unterstützt folgende verschiedene Modulare-Strukturen: Anonymer Block, Packages, Functions, Procedures und Trigger. Im Kopf werden Variablen, Cursor usw. deklariert, die dann im gesamten Block zur Verwendung bereit stehen. Die Sichtbarkeit der Variablen hängt von dem Ort der Deklaration ab, d.h. im inneren eines Blockes deklarierte Variablen sind nur in dem jeweiligen Block sichtbar. Gleiches gilt für Deklarationen zum Beispiel innerhalb von Schleifen. Um sich die Arbeit zu erleichtern bietet PL/SQL die Syntax *%TYPE* an, mit der einer Variablen der Datentyp eines Attributes einer Tabelle zugewiesen werden kann. Optional sind die Deklaration und das Fangen von Exceptions, welches nicht im Deklarationsteil geschieht sondern am Ende des Ausführungsteils. Es können vordefinierte Fehler abgefangen werden (Bsp: ZERO_DIVIDE: ORA-01476) sowie eigene (ORA-*20001*) definiert werden. Im Ausführungsteil können dann die eigentlichen Operationen implementiert werden. Die einzelnen Module werden später genauer erläutert.

Datentypen Oracle PL/SQL bietet viele Datentypen an, das liegt aber auch daran, dass Abwärtskompatibilität gegenüber älteren Oracle-Versionen gewahrt werden soll. Die gängigen Datentypen sind in Abbildung .3 rot markiert. VARCHAR2(L) verhält sich wie ein String, wobei L die mögliche Länge entweder in Zeichen (CHAR) oder in Byte (BYTE) angibt. Default ist CHAR als Zeichenlänge. VARCHAR2 darf in PL/SQL 32267 Bytes belegen wohingegen der gleichnamige Datenbanktyp nur 4000 Bytes belegen darf. In NUMBER(P,S) steht P für die Gesamtlänge (Genauigkeit) und S für die Zeichenanzahl (von P) der Dezimalstellen. Für Inhalte mit

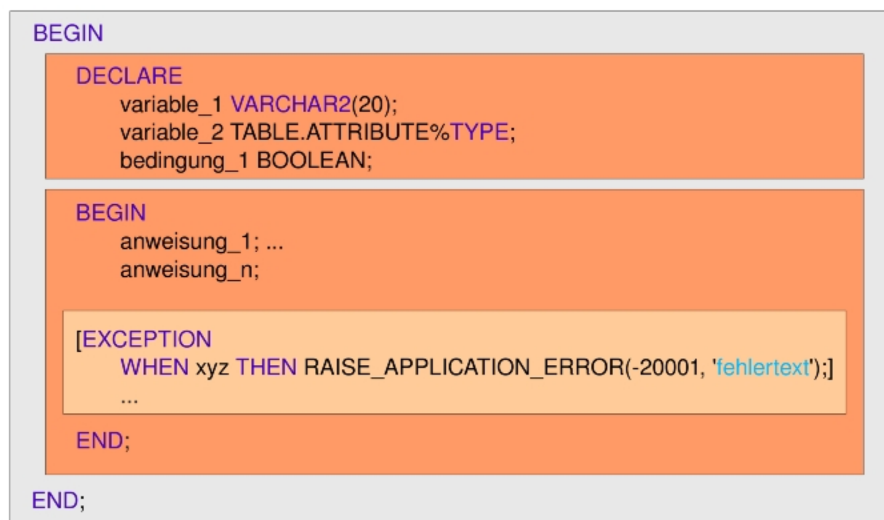


Abbildung 7.2: PL/SQL Blockstruktur.

mehr Speicher verwendet man LONG (2GB) oder CLOB (4GB). Datentypen lassen sich auch in andere Datentypen casten (TO_DATE). *%RECORD* ermöglicht zeilenweise (Datensatz) Datenverarbeitung. Dabei muss im Kopf ein *TYPE <name> IS RECORD* definiert werden, der zur Weiterverarbeitung verwendet werden kann. Kürzer ist der *%ROWTYPE* Befehl, der alle Typen einer Zeile in einer Tabelle übernimmt, dabei muss aber die exakte Verwendung der Attributnamen der Tabelle im Ausführungsteil beachtet werden. Das Cursor-Konzept stellt eine weitere wichtige Datenverarbeitung dar. Alle Datenverarbeitungsverfahren haben gemein, dass eine indirekte Verarbeitung von Werten der Tabelle vollzogen wird um die Sicherheit der Datenbank-Objekte zu gewährleisten (ACID). Wichtige Kontrollfunktionen sind *COMMIT, ROLLBACK und LOCK*. Letzteres sperrt den Schreibzugriff auf die Tabelle die z.B. in einer Java-Anwendung bearbeitet wird. Somit kann man eine Art Scheduling schalten. Für weitere Informationen bitte in die Online-Dokumentation schauen.

Cursor Der Cursor löst das sogenannte „Impedence Mismatch“ Problem, das soviel heißt wie „Fehlerhafte Variablenzuweisung“, denn bei einer SELECT-Anfrage wird meist eine Ergebnismenge (ResultSet) zurückgegeben, die natürlich nicht in einer Zeilenvariablen gespeichert werden kann. Die ResultSet muss ebenfalls zeilenweise

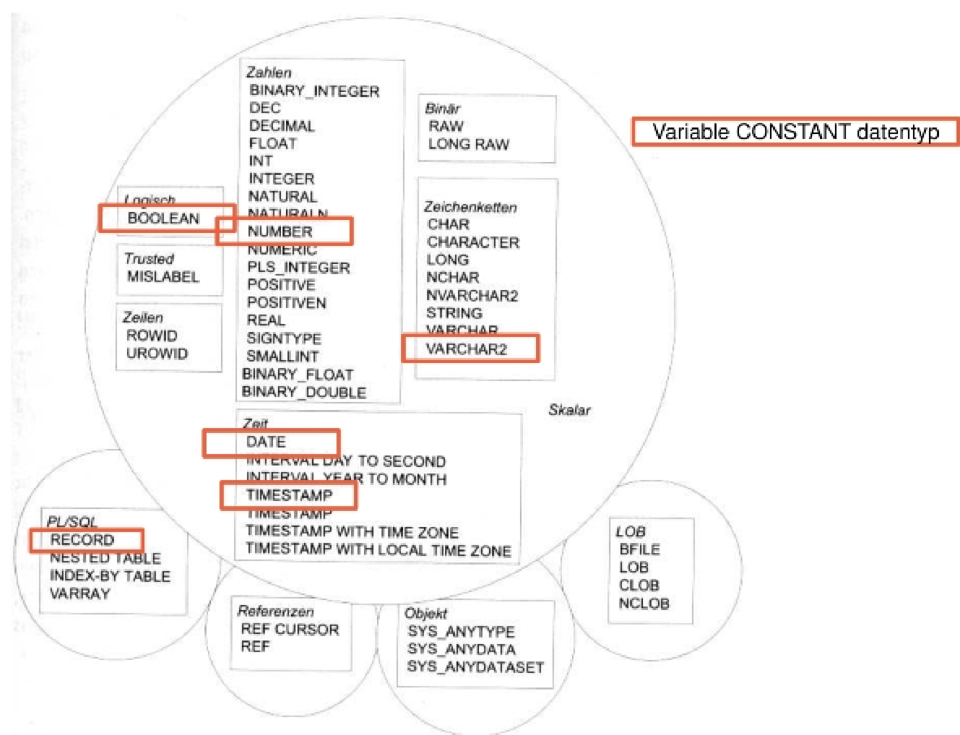
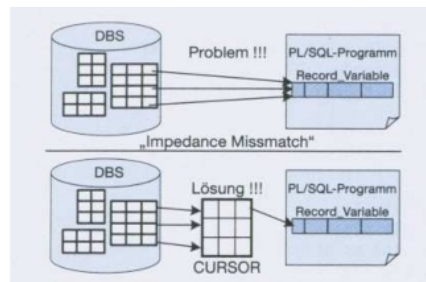


Abbildung 7.3: PL/SQL Datentypen.

abgespeichert werden und genau dafür sorgt der Cursor, der einen Zeiger auf das aktuelle Element in dieser ResultSet bereitstellt. Mittels einer Schleife kann man dann jedes Element also jede Zeile hintereinander verarbeiten. Dazu müssen jeweils die Attribute der Zeile in richtiger Reihenfolge vordeklarierten Variablen zugewiesen werden. Wichtig ist die Verwendung von *OPEN <cursor>* und *CLOSE <cursor>* welches sich genauso verhält wie der *LOCK*-Befehl.

Schleifen Die Schleifen sind wie bekannt zu verwenden.

Ablaufsteuerung und Sichtbarkeit Wie unter Datentypen schon beschrieben erkennt man in der Abbildung .6 die Sichtbarkeit von Variablen. PL/SQL unterstützt neben der bekannten IF-THEN-ELSE Anweisung auch die CASE Anweisung. Dazu mehr in der Online-Dokumentation.




```
CURSOR <c_name> IS SELECT <spalte1> FROM <tabelle> [ORDER BY <spalte>];
OPEN <c_name>;
LOOP
    FETCH <c_name> INTO { <variable1> | <recordname> };
        <verarbeitung der variable1>;
    EXIT WHEN <c_name>%NOTFOUND;
END LOOP;
CLOSE <c_name>;
```

Abbildung 7.4: CURSOR-Konzept gegen Impedence Mismatch.

```
FOR i IN 1..n LOOP
    <anweisung>;
END LOOP;
```

```
WHILE <bedingung> LOOP
    <anweisung>;
END LOOP;
```

Abbildung 7.5: PL/SQL Syntax der FOR- und WHILE-Schleife.

```
IF <bedingung1> THEN
    <anweisung1>; <variable1> NUMBER(3,1);
ELSIF <bedingung2> THEN
    <anweisung2>;
ELSE
    <anweisung3>;
END IF;
```

Abbildung 7.6: PL/SQL Ablaufsteuerung und Sichtbarkeit.

```
SELECT <spalte1>, <spalte2> FROM <tabelle> INTO <variable1>, <variable2>
WHERE <bedingung1>;
<verarbeitung der variablen>;
```

```
v_strg := q'# GRANT Lehrender TO tadros #';
EXECUTE IMMEDIATE v_strg;
```

Abbildung 7.7: PL/SQL Datenbankzugriff.

Datenbankzugriff und Native Dynamic SQL (NDS) PL/SQL hat Grenzen bezüglich der Kommunikation mit den Datenbankobjekten. Hierzu zählen Befehle wie *GRANT*. Um diese Befehle trotzdem verwenden zu können, muss ein String mittels *EXECUTE IMMEDIATE* übergeben werden, der direkt auf der Datenbank ausgeführt wird. Diese Art des Datenbankzugriffs nennt man Native Dynamic SQL (NDS).

Grundlagen

Vorweg: Oracle bietet neben den selbstdefinierten Funktionen usw. eine umfangreiche Bibliothek der man sich bedienen sollte. Die meisten Pakete beginnen mit

```
CREATE OR REPLACE FUNCTION GET_CURYEAR RETURN VARCHAR2
AS
 v_year INTEGER;
 v_edityear VARCHAR2(4);

 BEGIN
   SELECT extract(year FROM sysdate) INTO v_year FROM dual;
   v_edityear := to_char(v_year);

 RETURN v_edityear;
END GET_CURYEAR;
```

Abbildung 7.8: Funktion um das aktuelle Jahr zu berechnen.

*DBMS_* und fügen dem dann den Typ des Paketes an. DBMS_OUTPUT.LINE() ist eine Funktion um eine Ausgabe auf die Kommandozeile zu projezieren.

Function Eine Funktion wird meist für die Berechnung von Werten verwendet, um diese Werte als Rückgabewerte in die Prozedur zurückzuliefern. Der Unterschied zwischen einer Funktion und einer Prozedur ist lediglich die Tatsache, dass in der Funktion immer mindestens ein Rückgabewert (RETURN) vorhanden sein muss. In erster Linie dient die Funktion dem guten Programmierstil. Als Einstieg zunächst der Sprung ins kalte Wasser.

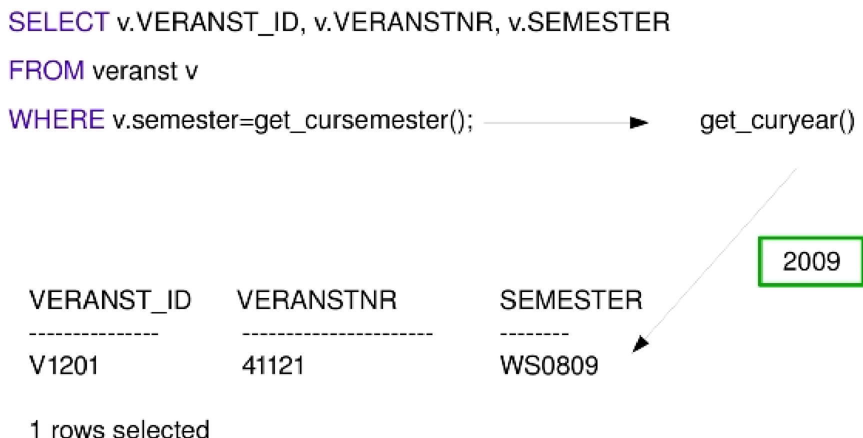


Abbildung 7.9: SELECT-Anfrage um Veranstaltungen des aktuellen Semesters auszugeben.

Was geschieht in dieser Funktion? Zusammengefasst wird die Ausgabe des aktuellen Jahres *2009* zurückgeliefert. Es fällt die allgemeine Blockstruktur wie oben erläutert auf. Lediglich Funktion-spezifische Syntax ändert das Bild etwas ab. Hier müssen Schlüsselwörter wie *AS* verwendet werden um den Deklarationsteil einzuleiten. Zu Beginn muss der Name *GET_ CURYEAR* der Funktion und der Rückgabewert *RETURN <datentyp>* definiert werden. Die SELECT Anweisung im Ausführungsteil ist neu. Hier wird über eine Dummy-Tabelle *DUAL* eine Anfrage gestellt. DUAL dient der Einhaltung der allgemeinen Syntax und gewährleistet, dass nur eine Zeile zurückgeliefert werden kann, da die Struktur von DUAL nur ein (variables) Tupel aufweist. *FROM DUAL* kann immer dann angewendet werden wenn man systemspezifische Informationen wie Zeit o.ä. erhalten möchte. Desweiteren fällt hier die oben angesprochene Verarbeitung von Attributen in Variablen auf. Jede *SELECT* Anweisung muss *INTO* einer Variablen bzw. Datensatzvariablen stattfinden. In Abbildung .9 erkennt man einen möglichen Gebrauch der Funktion GET_CURYEAR(). Mittels einer SELECT-Anfrage wird nach Veranstaltungen gesucht, die im aktuellen Semester gehalten werden. Die Prozedur *GET_ CURSEMESTER()* erstellt aus dem Rückgabewert *2009* der Funktion *GET_ CURYEAR()* den String *WS0809* der mit den Tabelleneinträgen verglichen wird. Die Ausgabe sind dementsprechend Veranstaltungen (hier nur eine), die im Wintersemester 2008/2009 gehalten werden.

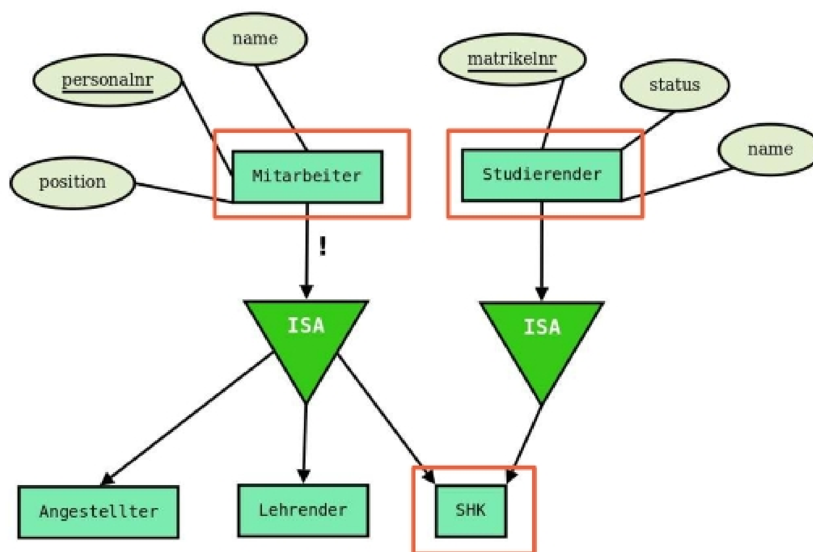


Abbildung 7.10: Beispiel eines Entity-Relationship-Model.

Procedure Für ein Beispiel einer Prozedur schauen wir zuerst auf eine Tabellenstruktur. Abbildung .10 zeigt ein ER-Diagramm indem Beziehungen zwischen Personen modelliert worden sind. Für unser Beispiel ist die Relation zwischen Mitarbeiter und Studierender interessant. Ein Studi kann nämlich als Studentische Hilfskraft (SHK) gleichzeitig ein Mitarbeiter sein. Nun existiert eine Mitarbeiter Tabelle (mi) mit PrimaryKey *PERSONALNUMMER* und eine Studierenden Tabelle (s) mit PrimaryKey MATRIKELNR. Die Prozedur in Abbildung .11 & .12 & .13 soll nun automatisch einen Datenabgleich über beide Tabellen durchführen und die Relation *PERSONALNUMMER/MATRIKELNR* in die Tabelle *ARBEITET_ALS_SHK* aufnehmen. Die Prozedur ruft man mit dem Befehl *EXECUTE* in der Datenbank auf. Ausgabe ist lediglich *anonymous block completed.* Ob die Prozedur also semantisch korrekt arbeitet bleibt zu überprüfen.

Trigger Trigger bieten die Möglichkeit Anfragen und Operationen auf der Datenbank abzufangen und zu verarbeiten. Trigger stellen somit ein nützliches Hilfsmittel aus Sicht der Sicherheit dar. In Abbildung .14 wird das Verfahren exemplarisch durchgeführt. Mit ihnen lassen sich z.B. INSERT Befehle abfangen, ein Attribut editieren (z.B. eine SEQUENCE einfügen) und erst dann weiter zur Ausführung

```
CREATE OR REPLACE PROCEDURE UPDATE_ARBEITET_ALS_SHK AS
BEGIN

 DECLARE
  v_pos mi.POSITION%TYPE;
  v_persnr mi.PERSONALNR%TYPE;
  v_matrnr s.MATRIKELNR%TYPE;

  counter NUMBER(4);
```

Abbildung 7.11: Deklarationsteil der Prozedur UPDATE_ARBEITET _ALS_SHK

```
CURSOR c_shk (b_mitarb mi.POSITION%TYPE:='SHK') IS
SELECT mi.PERSONALNR, s.MATRIKELNR FROM mi, s
WHERE POSITION=b_mitarb
AND mi.NAME=s.NAME
AND mi.VORNAME=s.VORNAME                    --GEFAHR: Leerzeichen
AND mi.GEBDATUM=s.GEBDATUM
AND mi.PERSONALNR NOT IN (SELECT r_shk.PERSONALNR FROM r_shk);
```

Abbildung 7.12: CURSOR Deklaration der Prozedur UPDATE_ARBEITET _ALS_SHK

```
BEGIN

 SELECT COUNT(*) INTO counter FROM mi WHERE mi.POSITION='SHK';
 OPEN c_shk;
 --raise_application_error(-20000, counter);

    LOOP
     FETCH c_shk INTO v_persnr, v_matrnr;
     EXIT WHEN c_shk%NOTFOUND;
     INSERT INTO ARBEITET_ALS_SHK VALUES(v_persnr, v_matrnr);
    END LOOP;

 CLOSE c_shk;
END;
END UPDATE_ARBEITET_ALS_SHK;
```

Abbildung 7.13: Ausführungsteil der Prozedur UPDATE_ARBEITET _ALS_SHK

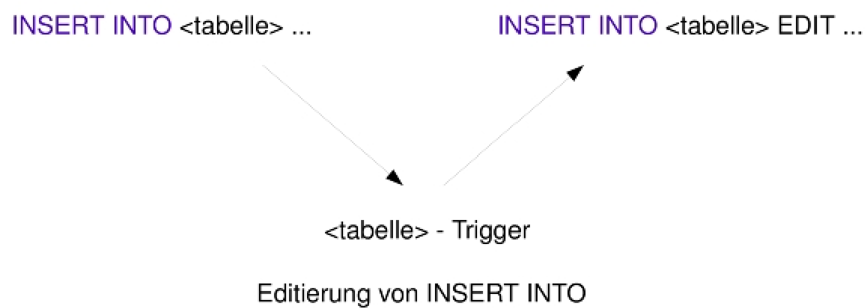


Abbildung 7.14: Konzept der Trigger.

schicken (s.Abbildung .15).

```
CREATE OR REPLACE TRIGGER STUDI_AUTOSETID_TRG  BEFORE INSERT ON
"STUDIERENDER"   FOR EACH ROW
 BEGIN
   IF inserting THEN
     IF :NEW."MATRIKELNR" IS NULL THEN
      SELECT MATRIKELNR_SEQUENCE.NEXTVAL INTO :NEW."MATRIKELNR"
      FROM dual;
      END IF;
   END IF;
 END;
```

Abbildung 7.15: Abfangen eines INSERT auf die Tabelle STUDIERENDER. Der Trigger wird ausgelöst bzw. „gefeuert“.

7.1.2 Security Policy - Basics

Einige Sammlungen von Richtlinien (Policies) werden von Aufsichtsbehörden aufbauend auf Gesetzgebungen vorgeschrieben. Bekannte Sammlungen sind die Sarbanes-Oxley Act (USA) und die EuroSox (Europa), die auf der Sarbanes-Oxley Act basiert und der Europäischen Gesetzgebung angepasst ist. Neben diesen vorgeschriebenen Richtlinien entwerfen Firmen meist in einem Gremium aus Datenschutzbeauftragten, IT-Fachleuten, Fachabteilungen und eventuell auch Juristen, ihre eigenen Globalen und Lokalen Richtlinien. Die Richtlinie ist eine umfassende und obligatorische Sammlung von Vorgaben und Bedingungen, die die gesamte (globale) Unternehmenspolitik darstellt. D.h. alle Firmensitze weltweit haben sich an die Global Policy zu halten. Diese Richtlinie deklariert allgemeine Standards, welcher Sicherheitslevel nach Norm (TCSEC, ITSEC, ITS(K), CC) verwendet werden soll, generelle Rechte, Privilegien und Verpflichtungen. Dahingegen werden keine technischen Details aufgeführt.

Definition 7.1.1:
Eine ***Sicherheitsstrategie*** *(security policy) teilt Systemzustände in disjunkte Zustandsmengen ein. Einmal* ***befugt*** *(authorized or secure) und* ***unbefugt*** *(unauthorized or nonsecure).*

Definition 7.1.2:
Eine ***Sicherheitsstrategie*** *(security policy) eines Systems legt die Menge von Re-*

geln und Maßnahmen fest, die zum Schutz der sensiblen Informationen bzw. Daten in dem betroffenen System einzusetzen sind.

7.1.3 Virtual Private Database

VPD ist ein Oracle-Konzept zur Umsetzung von Sicherheitsstrategien und bietet eine feingranulierte, zeilenbasierte Zugriffskontrolle an; Optional ist die spaltenbasierte Zugriffskontrolle. Mittels PL/SQL wird ein Mechanismus für VPD bereitgestellt, der zur vollständigen Realisierung notwendig ist. Abbildung .16 zeigt das Verfahren von VPD. Zwei „Sales Representatives“ (Verkäufer) möchten auf die Tabelle Orders (Aufträge) zugreifen und stellen eine *SELECT ** Anfrage. Normalerweise liefert das Ergebnis alle Einträge in der Tabelle zurück. Die Sicherheitsstrategie des Unternehmens sagt jetzt aber, das nur bestimmte Aufträge angesehen und bearbeitet werden dürfen. VPD setzt genau hier an, mittels einer Policy-Funktion wird die *SELECT ** Anfrage subtil editiert, d.h. der Benutzer bekommt nichts davon mit. Editiert heißt, das ein sogenanntes Prädikat angehängt wird. Das Prädikat findet sich in einer *WHERE-Klausel* wieder. Hier wird das Prädikat *sales_rep_id > 159* angehängt und bewirkt das nur Zeilen ausgegeben werden, dessen Attribut sales_rep_id einen Wert größer 159 besitzt. Der Effekt ist die subtile Ausblendung der anderen Zeilen. Der Verkäufer soll denken es gibt keine anderen Einträge in der Tabelle. Der Mechanismus findet direkt in der Datenbank statt, d.h. die Policy-Funktionen müssen in PL/SQL in der Datenbank implementiert werden. Um die Policy zu definieren und implementieren wird am besten der „Oracle Enterprise Manager“ oder der „Policy Manager“ verwendet, da der Programmieraufwand und die Fehleranfälligkeit bzgl. der Syntax nicht zu unterschätzen sind. Die Funktionen lassen sich sehr einfach über den Oracle-spezifischen „SQLDeveloper“ entwickeln. Abbildung .17 stellt eine *SELECT ** Anfrage aus Sicht des Mitarbeiters mit der *PERSONALNR=2IE18126199HK226* dar. In diesem Fall wird nur der Eintrag mit seinen eigenen persönlichen Daten angegeben, obwohl weitere 27 Zeilen (die der anderen Mitarbeiter) existieren. Die Policy-Funktion welche genau dieses umsetzt ist in Abbildung .19 angegeben. Richtig, woher soll die Datenbank wissen, dass es sich um die besagte Person handelt. Hierfür gibt es den sogenannten „Application Context“, der weiter unten behandelt wird und anhand von Verbindungsinformationen den Benutzer filtert. Die Reihenfolge für einen Entwurf eines VPD-basierten

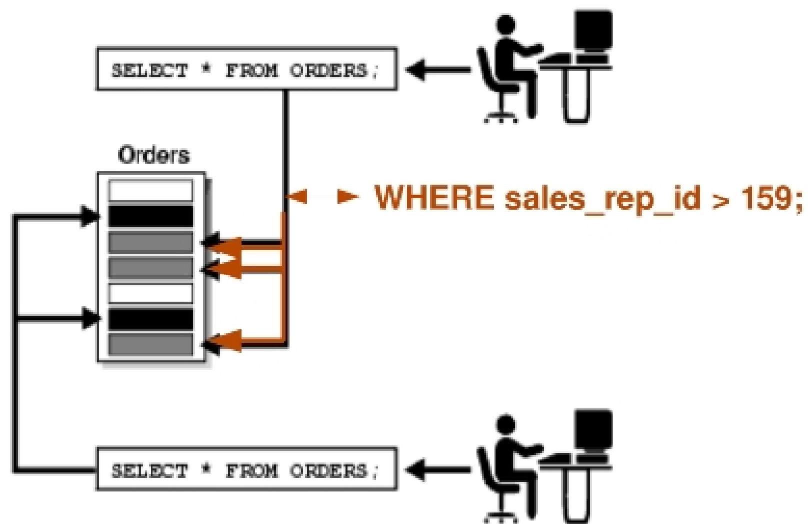


Abbildung 7.16: Oracle VPD-Konzept.

- SELECT * FROM mitarbeiter; WHERE Personalnr = '2IE18126199HK226'

PERSONALNR	TITEL	NAME	VORNAME	MAIL
2IE18126199HK226	Prof. Dr.	Biskup	Joachim	biskup@ls6.cs.uni-dortmund.de

1 rows selected

Abbildung 7.17: Interner Aufruf der Policy-Funktion.

```
BEGIN dbms_rls.add_policy(
object_schema => 'MAYBAUM',
object_name => 'MITARBEITER',
policy_name => 'pgtest', function_schema => 'Maybaum',
policy_function => 'PGPOLICY',
statement_types => 'select,',
update_check => FALSE , enable => TRUE , static_policy => FALSE ,
policy_type => dbms_rls.CONTEXT_SENSITIVE ,
long_predicate => FALSE , sec_relevant_cols => '' , sec_relevant_cols_opt => NULL );
END;
```

Abbildung 7.18: Definition einer Security Policy.

Mechanismus ist folgende:

- Policy exakt, informell ausformulieren
- Policy bei Bedarf formell ausformulieren
- Grundeinstellungen der Policy in Oracle übertragen (Namen, Typ, Schema, für welche Befehle, usw.)[1]
- Policy-Funktionen inkl. Verwendung des SYS_CONTEXT um User-Informationen zu erhalten in PL/SQL implementieren[2]

Folgende Policy-Typen können gewählt werden:

- STATIC - Bei diesem Policy-Typ wird die Policy-Funktion einmal ausgeführt. Danach wird das Prädikat in der SGA (System Global Area) im Hinblick auf schnelle Performance gecacht. Wird nur für ein Objekt angewendet.
- SHARED_STATIC - Wie STATIC, nur sucht der Server zuerst nach einem gecachten Prädikat, das von derselben Policy-Funktion desselben Policy-Typs generiert wurde. Wird für mehrere Objekte gemeinsam benutzt.

[1]vgl. Abbildung .18
[2]vgl. Abbildung .19

```
CREATE OR REPLACE FUNCTION PGPOLICY(
 schema_var IN VARCHAR2,
 table_var  IN VARCHAR2
)
RETURN VARCHAR2 IS
 return_val VARCHAR2 (400);
BEGIN
 --return_val := NULL;
 return_val := q'#Personalnr = '2IE18126199HK226'#';   } WHERE-Klausel
 RETURN return_val;
END PGPOLICY;
```

Abbildung 7.19: Policy-Funktion zu Abbildung .17.

- CONTEXT_SENSITIVE - Bei diesem Policy-Typ wertet der Server die Policy-Funktion bei der Ausführung der Anweisung erneut aus, wenn Kontextänderungen ermittelt werden. Der Server führt die Policy-Funktion stets beim Parsen der Anweisung aus und cacht den von der Funktion zurückgegebenen Wert nicht. Wird nur für ein Objekt angewendet.

Virtual Private Database (VPD) auf Spaltenebene setzt Security Policys auf Zeilenebene nur durch, wenn in der Benutzerabfrage auf eine bestimmte Spalte oder bestimmte Spalten zugegriffen wird. Diese Spalten, die sensible Informationen enthalten, werden als security-relevante Spalten markiert. VPD auf Spaltenebene wird für Tabellen und Views, jedoch nicht für Synonyme angewendet. VDP kann so konfiguriert werden, dass wie folgt zwei verschiedene Verhaltensweisen erzeugt werden:

- Standardverhalten, das die Anzahl von Zeilen beschränkt, die für eine Abfrage zurückgegeben werden, die Spalten mit sensiblen Informationen referenziert.
- Spalten-Masking-Verhalten von VPD auf Spaltenebene zeigt alle Zeilen an, einschließlich Zeilen, die sensible Spalten referenzieren, im Gegensatz zu dem

```
SELECT   substr(lower(sys_context('userenv','session_user')),1,10) AS Benutzer,
         substr(sys_context('userenv','ip_address'),1,10) AS IP,
         substr(sys_context('userenv','sessionid'),1,20) AS  ID
FROM dual;

BENUTZER     IP           ID
----------   ----------   --------------------
maybaum      129.217.19   1010015

1 rows selected
```

Abbildung 7.20: Beispiel zur Abfrage der USER_SESSION Informationen. Diese Abfrage könnte in der Policy-Funktion oder im AC implementiert werden.

Standardverhalten von VPD auf Spaltenebene, das die Anzahl von zurückgegebenen Zeilen beschränkt. Die sensiblen Spalten werden jedoch als NULL-Werte angezeigt.

7.1.4 Application Context

Ein Anwendungskontext dient der Erstellung einer virtuellen Umgebung mit bestimmten Bedingungen in der sich Benutzer innerhalb der Datenbank bewegen dürfen, d.h. es können Privilegien für bestimmte Gruppen (auch Programme), die sich mit der Datenbank verbinden implementiert werden. Wenn der *SYS_CONTEXT* nicht ausreicht können also eigene AC erstellt werden. Im folgenden werden nur kurz Varianten aufgelistet auf die aber nicht weiter eingegangen werden wird. Wichtig ist nur die Tatsache, das die Verwendung des richtigen Kontextes von der Rechnerarchitektur abhängt und dementsprechend gewählt werden muss.

Wie angesprochen existieren drei Typen von „application context". Alle drei basieren auf verschiedenen Vorbedingungen. Weitere Informationen in der Online-Dokumentation.

- db(server) session-based ac
- global ac

```
CREATE OR REPLACE CONTEXT orders_ctx USING orders_ctx_pkg;

-> Kunde(custom) soll nur eigene Aufträge(Orders) sehen dürfen
```

Abbildung 7.21: AC *orders_ ctx* wird erstellt.

```
CREATE OR REPLACE PACKAGE orders_ctx_pkg IS
  PROCEDURE set_custnum;
 END;
/
CREATE OR REPLACE PACKAGE BODY orders_ctx_pkg IS
  PROCEDURE set_custnum
  AS
    custnum NUMBER;
  BEGIN
    SELECT cust_no INTO custnum FROM scott.customers
       WHERE cust_email = SYS_CONTEXT('USERENV', 'SESSION_USER');
    DBMS_SESSION.SET_CONTEXT('orders_ctx', 'cust_no', custnum);
  EXCEPTION
   WHEN NO_DATA_FOUND THEN NULL;
  END set_custnum;
END;
/
```

Abbildung 7.22: Paket welches den AC definiert. Anhand der Email Adresse des aktuellen Benutzers wird die Kundennummer im Kontext geschaltet.

- client session-based ac

Die Abbildungen .21 & .22 & .23 beschreiben ein Beispiel für das Vorgehen einen eigenen AC zu definieren.

```
CREATE TRIGGER set_custno_ctx_trig AFTER LOGON ON DATABASE

 BEGIN
  sysadmin_vpd.orders_ctx_pkg.set_custnum;
 END;
/
```

Abbildung 7.23: Trigger um bei Anmeldung an der Datenbank den AC zu schalten.

Kapitel 8

Sicht-Änderungen

8.1 Sichten

Ein Datenbanksystem(DBS) besteht aus einer Vielzahl von Basisrelationen oder Grundfakten. Die wesentliche Aufgabe eines DBS ist es, große Datenmengen effizient, widerspruchsfrei und dauerhaft zu speichern und benötigte Teilmengen in unterschiedlichen, bedarfsgerechten Darstellungsformen für Benutzer und Anwendungsprogramme bereitzustellen. Diese Vielzahl kann für einen Benutzer einerseits unübersichtlich sein, andererseits ist es aber auch möglich, dass gewisse Informationen aus Vertraulichkeitsgründen vor dem Benutzer verborgen bleiben sollen.

Ein DBS besteht aus einem Datenbankmanagementsystem (DBMS), das Verwaltungssoftware genannt wird, und der Menge der zu verwaltenden Daten, der tatsächlichen Datenbank (DB). Die Verwaltungssoftware organisiert intern die strukturierte Speicherung der Daten und kontrolliert alle lesenden und schreibenden Zugriffe auf die Datenbank. Zur Abfrage und Verwaltung der Daten bietet ein Datenbanksystem eine Datenbanksprache an. Die bekannteste Form eines Datenbanksystems ist das Relationale Datenbanksystem, aus diesem Grund wird nur dieses in dieser Ausarbeitung betrachtet.

Die Arbeit des Datenbank-Benutzers kann erheblich erleichtert werden, indem Sichten(*Views*) benutzt werden können. Mehrere Relationen können so z.B. per Verbundoperation zu einer neuen Relation(Sicht) zusammengefasst und überflüssige

Informationen mittels Selektion oder Projektion ausgeblendet werden. Die neue Relation ist immer für eine bestimmte Anwendung gedacht. Es gibt zwei Arten von Sichten: die *virtuelle Sicht* und die *materialisierte Sicht*. In dieser Arbeit befasse ich mich mit der Letzteren.

8.1.1 Beispiel für eine Sicht

Betrachten wir zwei Relationen, welche einem Mitarbeiter eine Gehaltsklasse und einer Gehaltsklasse einen Betrag zuordnen:

MITARBEITER

Name	Gehaltsklasse
Ndula	A
Karsch	A
Müller	B
Abdul	C
Smith	C

GEHALT

Gehaltsklasse	Betrag
A	40,000
B	50,000
C	60,000
D	70,000

Abbildung 8.1: Zwei Relationen

Man kann eine neue Relation(Sicht) aus den beiden Relationen machen, indem die Gehaltsklasse mittels Projektion ausgeblendet wird. Die neue Relation hat dann als Tupel *Name* und *Betrag*. Der *SQL*-Befehl dafür lautet:

CREATE VIEW Mitarbeitergehalt AS
SELECT Mitarbeiter.Name, Gehalt.Betrag
FROM Mitarbeiter, Gehalt
WHERE Mitarbeiter.Gehaltsklasse = Gehalt.Gehaltsklasse;

Die resultierende Sicht *“Mitarbeitergehalt”* erhält folgende Werte:

8.2 Sichterhaltungsproblem(View Maintenance Problem)

Gegeben sind ein Basisschema B, ein Sichtschema V, und eine Sichtabbildung $f{:}Inst(B) \mapsto Inst(V)$.

Angenommen eine Sicht soll erhalten werden. Anders gesagt, wann immer die Basisdatenbankinstanz sich ändert, dann soll die Sichtinstanz entsprechend aktualisiert werden. Gesetzt den Fall, die Basisdatenbank B beinhaltet I_B, und μ (eine Menge von Einfügungen und Löschungen) bildet I_B auf $I_B^{'}$ ab. Eine naive Weise die Sicht zu aktualisieren wäre $f(I_B^{'})$ zu finden. Das wäre zu aufwendig, da $I_B^{'}$ wesentlich zu groß ist im Vergleich zu der Differenz zwischen I_V und $I_V^{'}$. Aus diesem Grund muss man effizientere Wege benutzen, um das Update v in Abbildung 3 zu finden. Das nennt man das *Sichterhaltungsproblem*.

8.2.1 Lösungsansatz

Algorithmen können benutzt werden, um zu testen, ob eine Änderung an der Basisdatenbank eine Wirkung auf die Sicht hat. Wenn eine solche Aktualisierung die Sicht betrifft, sagt man die Aktualisierung ist „relevant", sonst ist sie „irrelevant".
Beispiel 2.1: Gegeben eine Basisdatenbankschema B = (R[AB],S[BC]) und vier Sichten V_1 bis V_4.
$V_1 = (R \bowtie \sigma_{c>50} S)$
$V_2 = \pi_A R$
$V_3 = R \bowtie S$
$V_4 = \pi_{AC}(R \bowtie S)$
Wenn wir jetzt $\langle b, 20 \rangle$ in S einfügen, führt dies zu keiner Änderung in V_1 und V_2. Änderungen in V_3 und V_4 hängen von den schon vorhandenen Werten in der Datenbank ab.

8.3 Sichtänderungsproblem

Als Erinnerung, das Sichterhaltungsproblem tritt auf, wenn die Änderungen, die in der Basisdatenbank vorgenommen werden, auf die Benutzersicht übertragen werden sollen. Bei dem Sichtänderungsproblem ist der umgekehrte Fall zu behandeln; nämlich, dass Änderungen, die ein Benutzer auf seine Sicht vorgeführt hat, auf die Datenbank übertragen werden müssen. Also, das Gegenteil von Sichterhaltungsproblem.
Hier wird ein Basisschema I_B, ein Sichtschema I_V und ein Update V auf I_V gegeben.

Zu finden ist ein Update μ, so dass Abbildung 4 vollständig wird.

Jetzt, betrachten wir das Beispiel aus 8.1.1 und einen Geschäftsführer, der das Gehalt des Mitarbeiters *Karsch* auf 50.000€ erhöhen möchte. Das Problem jetzt ist, dass die auf der Sicht durchzuführende Änderung auf die Basisrelationen abgebildet werden muss. Eine Möglichkeit wäre es, den Betrag der Gehaltsklasse *A* auf 50.000€ zu ändern, was bedeutet, dass alle Mitarbeiter der Gehaltsklasse *A* eine Gehaltserhöhung bekommen. Andererseits wäre es auch möglich, die Gehaltsklasse von Mitarbeiter *Karsch* auf B zu ändern, was bezüglich der Auswirkungen auf die Sicht genau den gleichen Effekt hat und vermutlich auch eher der Intention des Geschäftsführers entspricht. Dass zwei Möglichkeiten zur Verfügung stehen, zeigt, dass die auf den Basisrelationen durchzuführende Änderung nicht eindeutig bestimmt ist.
Gesetzt den Fall, der Geschäftsführer möchte den Eintrag von Mitarbeiter *Abdul* löschen. Er hat wiederum zwei Möglichkeiten: Entweder kann er *Abdul* aus der Tabelle *Mitarbeiter* entfernen, was zu dem richtigen Ergebnis, führt oder er entfernt die Gehaltsklasse *C* aus der Relation *Gehalt*.

Definition 8.3.1:
Diese Nicht-Eindeutigkeit der Umsetzung einer Sichtänderung auf einer Basisrelation nennt man das „Sichtänderungsproblem “.

In der Regel sind Sichten mit mehreren Relationen, wie im obigen Beispiel, nicht eindeutig änderbar. Für Sichten, die sich auf nur eine Relation beziehen, gibt es einfache Bedingungen unter denen das Sichtänderungsproblem nicht auftritt, und Sichten, in der Gruppierungs- und Aggregationsfunktionen vorkommen, sind gänzlich nicht änderbar. Es macht zum Beispiel keinen Sinn, einen in einer Sicht dargestellten Durchschnittswert zu ändern, der sich ja aus einer Vielzahl von Grunddaten zusammensetzen kann.

8.3.1 Änderungssemantik von relationalen Sichten

In diesem Abschnitt wollen wir die Anforderungen und Eigenschaften der behandelten Sichtänderungsproblematik betrachten. Wir steigen jetzt in einige der getroffenen Definitionen ein und deuten diese dann bezüglich der Sichtänderungsproblematik. Im Folgenden bezeichne

- DS das Schema einer gegebenen Datenbank,
- f eine Sicht(-Abbildung),
- db einen Datenbankzustand (eine Instanz),
- U_{DS} die Menge aller Datenbank-Änderungen,
- U_f die Menge aller Sicht-Änderungen und
- T eine Abbildung $U_f \mapsto U_{DS}$.

Übersetzung

Gegeben sei eine Sichtänderung $u \in U_f$. Eine Datenbankänderung $\mathrm{T}(u) \in U_{Df}$ wird als Übersetzung von u bezeichnet, wenn folgende zwei Eigenschaften gelten:

1. **Konsistenz**: $f \circ \mathrm{T}(u) = u \circ f$.
   D.h. jede von einem Benutzer durchgeführte Änderung (u) auf einer Sicht muss so übersetzt werden , dass f auf die Übersetzung ($\mathrm{T}(u)$) das gleiche Ergebnis ($f \circ \mathrm{T}(u)$) liefert wie die Änderung auf der Sicht selbst ($u \circ f$). Schließlich soll die vom Benutzer durchgeführte Änderung auf seiner Sicht auch nach ihrer Ausführung zu dem vom ihm erwarteten Bild führen.
2. **Akzeptierbarkeit**: $\forall db \in DS$ gilt: $u \circ f(db) = f(db) \Rightarrow \mathrm{T}(u)(db) = db$.
   Diese Eigenschaft stellt sicher, dass Änderungen auf einer Sicht, die diese unverändert lassen, auch den Datenbankzustand nicht verändern.

Vollständige Menge von Sichtänderungen

Eine Menge $U \subset U_f$ von Sichtänderungen ist vollständig, wenn gilt:

1. **Komposition**: $\forall u \in U, \forall v \in U$ gilt : $u \circ v \in U$
2. **Umkehrbarkeit**: $\forall db \in DS, \forall u \in U, \exists v \in U : v \circ u \circ f(db)$. Diese Eigenschaft ist sehr sinnvoll, da der Benutzer in der Lage sein sollte, eine von ihm versehentlich durchgeführte, inkorrekte Einfügung korrigieren zu können. Der

Benutzer erwartet auch, dass eine Datenbankinstanz in einen Zustand zurückgesetzt werden könnte, der bereits einmal existierte.

8.3.2 Sichtänderungen unter der kontrollierten Anfragenauswertung

Die Sicht eines Benutzers auf eine Datenbankinstanz entsteht durch die Antworten auf eine Folge von ihm gestellten Anfragen. Ein wesentlicher Unterschied zu Sichten in „klassischen“ Datenbanken besteht darin, dass die bei der kontrollierten Anfrageauswertung gegebenen Antworten nicht mit dem tatsächlichen Datenbankinhalt übereinstimmen müssen. Während man den Verweigerungs-Zensor noch am ehesten mit einer Selektion vergleichen kann, stellen die Lügen-Zensoren ein in klassischen Datenbanken unbekanntes Konzept dar, indem sie Fakten „erfinden”, die so gar nicht existieren. Darüber hinaus hängen die gegebenen (und nicht gegebenen!) Antworten auch von der Wahl der Reihenfolge der Anfragen ab, was eine Folge der dynamischen Zugriffskontrolle ist. Bei den klassischen Sicht-Änderungen bestand die Problematik darin, dass es unter Umständen mehr als nur eine Übersetzung der Änderung gibt und so keine Eindeutigkeit bezüglich der Durchführung bestand. In dieser Ausarbeitung wollen wir diese Form der Mehrdeutigkeit nicht betrachten. Da wir beispielsweise Verbund-Operationen nicht betrachten, ist es aus der Sicht des Benutzers immer eindeutig, wie eine von ihm vorgenommene Sicht-Änderung vorgenommen werden soll, nämlich genau durch die Einfügung bzw. Entfernung oder Änderung der von ihm genannten Fakten in die Grundfaktenmenge. Aus der Sicht des Administrators ergibt sich auch nicht das Problem, dass es Mehrdeutigkeiten bezüglich der Abbildung einer Änderung geben könnte, sondern das Problem, dass die Sicht des Benutzers nicht der tatsächlichen Instanz entsprechen muss und dass daher eine „primitiv“ durchgeführte, direkte Änderung des Datenbankzustandes nicht den Erwartungen eines Benutzers entspricht und schlimmstenfalls sogar neue (verbotene) Inferenzen ermöglicht. In gewisser Weise wünscht sich der Administrator sogar Mehrdeutigkeiten, nämlich in Form der aus der kontrollierten Anfrageauswertung bekannten Vertraulichkeitseigenschaft: dem Benutzer muss es immer glaubhaft erscheinen können, dass in der ihm vorliegenden Instanz keine Geheimnisse vorliegen. Natürlich erwartet ein Benutzer, dass er nach einer Sicht-Änderung passende Antworten zu seinen Anfragen auf der neuen Instanz erhält. D.h., dass der Übersetzer

(gestrichelter Pfeil), also korrekt arbeitet 8.3.2.

Die Sicht eines Benutzers auf eine Datenbankinstanz entsteht durch die Antworten auf eine Folge von ihm gestellten Anfragen. Ein wesentlicher Unterschied zu Sichten in „klassischen" Datenbanken besteht darin, dass die bei der kontrollierten Anfrageauswertung gegebenen Antworten nicht mit dem tatsächlichen Datenbankinhalt übereinstimmen müssen. Während man den Verweigerungs-Zensor noch am ehesten mit einer Selektion vergleichen kann, stellen die Lügen-Zensoren ein in klassischen Datenbanken unbekanntes Konzept dar, indem sie Fakten „erfinden", die so gar nicht existieren. Darüber hinaus hängen die gegebenen (und nicht gegebenen!) Antworten auch von der Wahl der Reihenfolge der Anfragen ab, was eine Folge der dynamischen Zugriffskontrolle ist. Bei den klassischen Sicht-Änderungen bestand die Problematik darin, dass es unter Umständen mehr als nur eine Übersetzung der Änderung gibt und so keine Eindeutigkeit bezüglich der Durchführung bestand. In dieser Ausarbeitung wollen wir diese Form der Mehrdeutigkeit nicht betrachten. Da wir beispielsweise Verbund-Operationen nicht betrachten, ist es aus der Sicht des Benutzers immer eindeutig, wie eine von ihm vorgenommene Sicht-Änderung vorgenommen werden soll, nämlich genau durch die Einfügung bzw. Entfernung oder Änderung der von ihm genannten Fakten in die Grundfaktenmenge. Aus der Sicht des Administrators ergibt sich auch nicht das Problem, dass es Mehrdeutigkeiten bezüglich der Abbildung einer Änderung geben könnte, sondern das Problem, dass die Sicht des Benutzers nicht der tatsächlichen Instanz entsprechen muss und dass daher eine „primitiv" durchgeführte, direkte Änderung des Datenbankzustandes nicht den Erwartungen eines Benutzers entspricht und schlimmstenfalls sogar neue (verbotene) Inferenzen ermöglicht. In gewisser Weise wünscht sich der Administrator sogar Mehrdeutigkeiten, nämlich in Form der aus der kontrollierten Anfrageauswertung bekannten Vertraulichkeitseigenschaft: dem Benutzer muss es immer glaubhaft erscheinen können, dass in der ihm vorliegenden Instanz keine Geheimnisse vorliegen. Natürlich erwartet ein Benutzer, dass er nach einer Sicht-Änderung passende Antworten zu seinen Anfragen auf der neuen Instanz erhält. D.h., dass der Übersetzer (blauer Pfeil), also korrekt arbeitet(Abbildung 6).

Aus Sicht des Administrators sollen diese erwarteten Eigenschaften ebenfalls erfüllt sein, aber in einer Art und Weise, welche die Vertraulichkeit gewährleistet.

8.3.3 Lösungsansatz zum Sichtänderungsproblem

Die Nicht-Eindeutigkeit von Sichtänderungen können schon bei der Definition eines Datenbankschemas vermieden werden, indem der Administrator eine sinnvolle Definition angibt.

8.4 Multilevel Security und Polyinstantiierung

8.4.1 Multilevel Secure Databases

Diese Datenbanken benutzen das mandatorische Zugriffskontrollsystem, wobei alle Benutzer jeweils einer Sicherheitsebene zugeordnet werden und die Daten mit Sicherheitsmarken versehen sind.
Bei relationalen Datenbanken hat man dabei folgende Ebenen, auf denen man eine Sicherheitsmarke anwenden kann:
Datenbank-Ebene: hier wird eine komplette Datenbank mit einer Sicherheitsmarke versehen.
Relations-Ebene: ermöglicht das Abgrenzen von Relationen untereinander.
Tupel-Ebene: unterscheidet Tupel verschiedener Ebenen innerhalb einer Relation.
Element-Ebene: ermöglicht es, die einzelnen Attribute innerhalb eines Tupels bezüglich ihrer Sicherheitsebene zu spezifizieren.
Unten ist ein Beispiel eines Multilevel Secure Database mit Sicherheitsmarken auf Element und Tupel-Ebene(TC).

8.4.2 Polyinstanziierung

Betrachten wir nun einen Benutzer auf der U-Ebene, welcher die für ihn neue Information, dass sich das Raumschiff Enterprise auf einer diplomatischen Mission nach Vulcan befindet, in die Relation SOD(**S**tarship **O**rder **D**estination) einfügen will. Ein „herkömmliches" Datenbanksystem wird die Einfügung mit Hinweis auf einer Primärschlüsselverletzung abweisen. Für den Benutzer wäre diese überraschend, da sich aus seiner Sicht kein Eintrag bzgl. des Raumschiffes Enterprise in der Datenbank befindet. Der Benutzer kann nun schließen, dass er offensichtlich keine vollständige

Sicht auf den Datenbankzustand besitzt und sich daher das Raumschiff Enterprise tatsächlich auf einer geheimen Mission befinden muss. Um eine Abweisung der Einfügung zu unterdrücken, bleibt nur noch der Ausweg übrig, sowohl den geheimen S-Eintrag (Enterprise, Spying, Romulus) als auch den U-Eintrag (Enterprise, Diplomacy, Vulcan) in die Datenbank aufzunehmen, wie in 8.4.2 dargestellt. Diesen Sachverhalt, wobei in einer Relation zwei oder mehr Tupel mit dem gleichen Primärschlüssel vorkommen, nennt man *Polyinstanziierung*.

Es gibt zwei Arten von Polyinstanziierung: *Entität-Polyinstanziierung* und *Element-Polyinstanziierung*.
Definition 4.1 (*Entität-Polyinstanziierung*). Entität-Polyinstanziierung liegt vor, wenn in einer Relation mehrere Tupel mit dem gleichen Primärschlüssel vorkommen. Diese Primärschlüssel sind mit unterschiedlichen Sicherheitsmarken versehen.
Die Interpretation dieses Vorkommens ist, dass die Primärschlüssel auch auf unterschiedliche Objekte der realen Welt verweisen.
Im Unterschied dazu steht die Element-Polyinstanziierung:
Definition 4.2 (*Element-Polyinstanziierung*).
Element-Polyinstanziierung liegt vor, wenn in einer Relation mehrere Tupel mit dem gleichen Primärschlüssel vorliegen, diese Schlüssel mit den gleichen Sicherheitsmarken versehen sind und sich die Marken von mindestens einem weiteren Attribut unterscheiden.
Eine Interpretation der Element-Polyinstanziierung ist, dass beide Primärschlüssel auf das gleiche „reale" Objekt verweisen und die auf der niedrigeren Ebene vorliegenden weiteren Attribute eine Cover-Story für die geheimen Informationen in der höheren Ebene sind.

Polyinstanziierung und kontrollierte Anfrageauswertung

In der kontrollierten Anfrageauswertung werden Anfragen bzgl. ihres Zutreffens oder Nicht-Zutreffens ausgewertet. Für einen anfragenden Benutzer, der so nach und nach Informationen über das (Nicht-)Zutreffen von Fakten erhält, ergibt sich also eine Sicht auf die in der betrachteten Instanz enthaltenen Fakten, welche auch mit Hilfe von Relationen dargestellt werden können. Wir betrachten dazu ein einfaches Beispiel:
Beispiel 4.1:

- *db*:= $\{a, b, c, s\}$
- *pot_sec*:=$\{s\}$
- *constraints*:= $\{b \Rightarrow s\}$
- *Q*:= $\langle a, b, c, s\rangle$

Auf die Anfragen aus Q enthält man als Antwort *ans*:= $\langle a, \neg b, c, \neg s\rangle$. Sowohl die Instanz *db* als auch die Antwort *ans* lassen sich relational darstellen:

Das obige Beispiel zeigt, dass die kontrollierte Anfrageauswertung mit Lügenzensor eng verwandt mit MLS-Datenbanken mit Polyinstanziierung ist, denn man kann die Instanz- und Antwort-Relation auch zusammen in einer polyinstanziierten Relation abbilden.

Semantik in MLS-Datenbanken

Man muss sich klarmachen, welche Ebene eigentlich welche Fakten für wahr hält. Für den U-Benutzer ist die Sicht relativ klar, er kann nur das glauben, was er sieht. Ein S-Benutzer, der die oben genannte Relation vor sich hat, sieht sowohl Tupel auf seiner Ebene, als auch die Tupel aller von ihm dominierten Ebenen. Welche Tupel soll er jetzt für glaubhaft halten?
Wir werden hier kurz zwei verschiedene Ansätze von „Glaubens-Semantik" vorstellen:

1. **Suspicious Approach** („verdächtiger" Ansatz):
   Der Benutzer glaubt nur an diejenigen Daten, die er auf seiner eigenen Ebene sieht, daher werden die Anfragen nur auf der Ebene des anfragenden Benutzers ausgewertet.
2. **Trusted Approach** („vertraunsvoller" Ansatz):
   Der Benutzer vertraut gründsätzlich auch Informationen auf niedrigeren Ebenen. Dabei „überschreiben" natürlich polyinstanziierte Informationen auf höheren Ebenen die entsprechenden Cover Stories auf niedrigeren Ebenen. Der Vorteil bei diesem Ansatz ist, dass weniger Redundanzen entstehen. So müssen nicht alle Informationen, die auf mehreren Ebenen gültig und glaubhaft sind,

auf die jeweils höheren Ebenen kopiert werden. In dem Beispiel in Abbildung 10 könnte man die Tupel für die Einträge *a* und *c* auf der S-Ebene weglassen.

Einer der Nachteile ist, dass Informationen in der niedrigeren Ebene immer als „glaubhaft" erscheinen, auch wenn das nicht der Fall ist. Auch gibt es keine Möglichkeit den Nicht-Glauben an Informationen auszudrücken, sondern nur den Weg, durch Polyinstanziierung die niedrigen Ebenen zu überschreiben.

Glaubenszusicherung durch reichhaltige Sicherheitsmarken

Die Vorteile vom Trusted Approach und Suspicious Approach können mit reichhaltigen Sicherheitsmarken vereinigt werden. Die einfachen Marken **U**, **C** und **S** werden durch reichhaltigere „Glaubenszusicherungsstrings" wie z.B, **US**, **C-S**, **U-S** ersetzt, welche die folgenden Bedeutungen haben:
US : dieses Tupel wird von den Ebenen *U* und *S* als glaubhaft angesehen, währenddessen es für *C* „irrelevant" ist.
C-S : Ein Tupel, an welches Ebene *C* glaubt und das von *S* explizit als Cover Story identifiziert, und unter dem gleichen Primärschlüssel gibt es einen Eintrag, an den S glaubt. Dieses Tupel ist für Ebene *U* nicht sichtbar.
U-S : Ebene *U* hält die Information für glaubhaft, *C* dagegen sieht die Information für sich als „irrelevant“ an. *S* hält das Tupel für falsch.

Vermeidung von Polyinstanziierung

Die Änderungen von Benutzern auf der niedrigen Ebene eröffnen einen Inferenzkanal zu Daten auf der höheren Ebene. Da Primärschlüsseleigenschaft verletzt ist, stellt sich jetzt die Frage, ob Polyinstanziierung und Integrität überhaupt miteinander kompatibel sind. Dazu gibt es zwei verschiedene Meinungen:
Einerseits sind Polyinstanziierung und Integrität fundamental inkompatibel und daher muss Polyinstanziierung unter allen Umständen verhindert werden. Andererseits ist Polyinstanziierung ein inhärentes Phänomen, dass in der MLS-Welt unvermeidlich ist.

Lösungsansätze

Die hier vorgeschlagenen Lösungsansätze zur Vermeideung von Polyinstanziierung sind aber entweder nicht praktikabel oder mit einer hohen Verfügbarkeitseinschränkung verbunden:

- Die Relationen können in je eine pro Ebene partitioniert werden.
- Partitioniere die Domäne des Primärschlüssels: z.B. U-Benutzer nur Schlüssel mit Anfangsbuchstaben A-E benutzen und S-Benutzer nur solche Schlüssel, die mit F-Z beginnen.
- Ermögliche Einfügungen nur durch vertrauensvolle Benutzer.

MITARBEITERGEHALT

Name	Betrag
Ndula	40,000
Karsch	40,000
Müller	50,000
Abdul	60,000
Smith	60,000

Abbildung 8.2: Sicht aus beiden Relationen

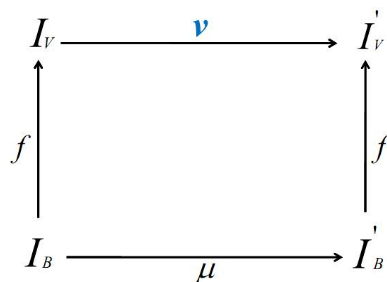


Abbildung 8.3: Sichterhaltung

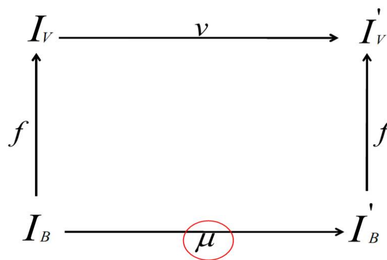


Abbildung 8.4: Sichtänderung

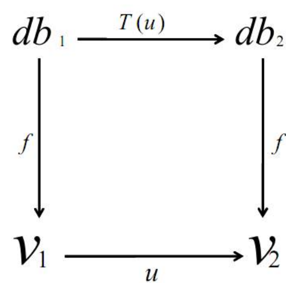


Abbildung 8.5: Sichtänderungen nach [BS81]

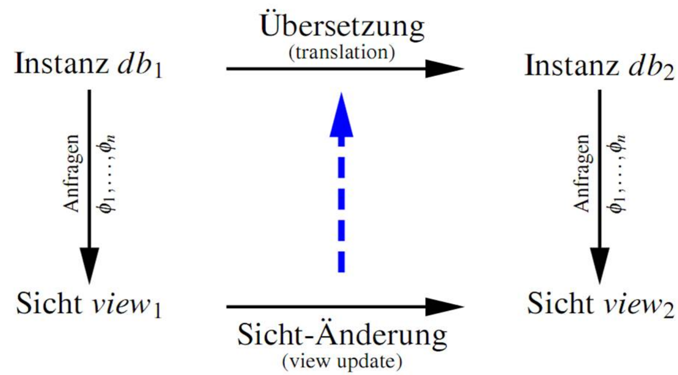


Abbildung 8.6: Sichtänderungen unter der kontrollierten Anfrageauswertung

SOD	Starship		Order		Destination		TC
	Enterprise	U	Spying	S	Romulus	S	S
	Falcon	U	Exploration	U	Talos	U	U
	Valiant	U	Exploration	U	Eden	U	U
	Voyager	S	Spying	S	Kronos	S	S

Abbildung 8.7: Eine Beispielinstanz für eine MLS-Relation SOD

SOD	Starship		Order		Destination		TC
	Enterprise	U	Spying	S	Romulus	S	S
	Falcon	U	Exploration	U	Talos	U	U
	Valiant	U	Exploration	U	Eden	U	U
	Voyager	S	Spying	S	Kronos	S	S
	Enterprise	U	Diplomacy	U	Vulcan	U	U

Abbildung 8.8: Eine polyinstatiierte MLS-Relation SOD

db:

VAR	Wert
a	TRUE
b	TRUE
c	TRUE
s	TRUE

ans:

VAR	Wert
a	TRUE
b	FALSE
c	TRUE
s	FALSE

Abbildung 8.9: Relationale Darstellung von Instanz und Antwort

VAR	Wert	TC
a	TRUE	S
a	TRUE	U
b	TRUE	S
b	FALSE	U
c	TRUE	S
c	TRUE	U
s	TRUE	S
s	FALSE	U

Abbildung 8.10: Kontrollierte Anfrageauswertung interpretiert als polyinstantiierte MLS-Relation

VAR	Wert	TC
a	TRUE	S
~~a~~	~~TRUE~~	~~U~~
b	TRUE	S
b	FALSE	U
c	TRUE	S
~~c~~	~~TRUE~~	~~U~~
s	TRUE	S
s	FALSE	U

Abbildung 8.11: „Überschreiben” von polyinstanziierte Informationen durch entsprechenden Cover Stories

Kapitel 9

Inferenzfreie Sicht-Änderung

9.1 Anforderungen an inferenzfreie Sicht-Änderungen

Bevor Sicht-Änderungen in diesem Kapitel formal definiert werden und ein entsprechender Algorithmus vorgestellt wird, sollen zunächst Eigenschaften, die einerseits aus Sicht des Benutzers bezüglich seiner Änderung und andererseits aus Sicht des Administrators gelten sollen, vorgestellt werden.

9.1.1 Anforderungen aus Sicht eines Benutzers

Da wir im Folgenden nur vollständige Instanzen betrachten, sind die möglichen Benutzer-Operationen

- Einfügen
- Löschen
- Ändern

identisch: das Entfernen eines Faktums entspricht seiner Negation, gleiches gilt für das Einfügen. Wir sprechen daher nur von Änderungs-Operationen.

Wenn ein Benutzer seine Sicht modifiziert, so erwartet er, dass Antworten auf weitere Anfragen konsistent mit den von ihm zuvor gemachten Änderungen sind. Die Antworten des Systems sollen also korrekt sein, so dass diese mit der vom Benutzer vermuteten Welt übereinstimmen. (Korrektheit)

Eine vom Benutzer angeforderte Änderung soll nur in dem Fall, dass diese auch tatsächlich Auswirkungen auf die Sicht des Benutzers hat, eine echte Modifikation der Datenbankinstanz zur Folge haben. Dementsprechend erwartet ein Benutzer eine Abweisung seiner Änderungs-Operation, wenn er versucht einen Zustand zu erzeugen, an den er entweder schon glaubt, oder aber der bereits existiert, aber durch seine vorangegangenen Anfragen noch nicht aufgedeckt ist. (Akzeptierbarkeit)

Weiterhin ist sich ein Benutzer eventuell vorhandener Konsistenzbedingungen bewusst, so dass nicht alle seine Änderungs-Wünsche erfolgreich sein können bzw. dürfen. Bei einer solchen Konsistenzverletzung soll das System eine entsprechende Meldung an den Benutzer zurückliefern und seine gewünschte Änderung nicht durchführen. (Konsistenz)

Schließlich möchte ein Benutzer noch in der Lage sein durch ihn selbst begangene Fehler korrigieren zu können. Dazu erwartet er, dass eine von ihm angeforderte und vom System akzeptierte und durchgeführte Änderung auch wieder rückgängig gemacht werden kann. (Umkehrbarkeit)

9.1.2 Anforderungen aus Sicht des Administrators

Der Administrator hat zunächst einmal den Wunsch, alle im vorherigen Abschnitt definierten Anforderungen, die ein Benutzer hat, erfüllt zu sehen. Zusätzlich muss aus Sicht des Administrators noch ein sogenanntes Vertraulichkeitsinteresse eingehalten werden. Ein Benutzer darf diesem Interesse nach unter keinen Umständen, durch eigene Anfragen oder Änderungs-Operationen und die dadurch resultierenden Systemreaktionen bzw. Systemantworten auf Geheimnisse schließen können.

Um diese zusätzliche Anforderung gewährleisten zu können, darf das System gegebenenfalls lügen, wie es bereits aus der kontrollierten Anfrageauswertung bekannt ist. Dabei ist allerdings aus Gründen der Integrität die Verwendung von Lügen möglichst einzuschränken. Insbesondere bedeutet dies, dass das System nach Möglichkeit

nicht lügen soll, wenn die Wahrheit keine Geheimnisse verletzt. Des Weiteren muss jede Lüge gerechtfertigt sein, was genau dann der Fall ist, wenn die Wahrheit an Stelle der Lüge zur Aufdeckung von Geheimnissen führen würde.

Im Folgenden wird anhand von 2 Beispielen gezeigt, dass das hier vorgestellte Vertraulichkeitsinteresse nicht trivial zu erfüllen ist.

Beispiel 9.1.1 (Erfolgreiches Ändern von Werten)**:**
Gegeben sei ein Datenbankschema mit der Konsistenzbedingung $(a \vee b)$*, welche dem Benutzer bekannt ist. Der Benutzer fügt nun erfolgreich den Wert a ein. Aufgrund der Systemantwort „Einfügung erfolgreich" kann dieser folgern, dass vorher* $\neg a$ *in der Datenbank enthalten gewesen sein muss. Aus dieser Information lässt sich nun wiederum schließen, dass b in der Datenbank gültig war und weiterhin ist, da ansonsten das Ändern des Wertes von a die Konsistenz verletzt hätte.*

Wäre b in diesem Beispiel ein Geheimnis gewesen, so ist klar, dass aus Sicht des Benutzers nie eine echte Änderung hätte stattfinden dürfen. Weiterhin scheint die Aktualisierung des Benutzerlogs nach einer erfolgreichen Einfügung problematisch. Würde man nach einer entsprechenden Einfügung das initiale $log_0 = \{a \vee b\}$ nur um das eingefügte Faktum a erweitern, erhält man $log' = \{a \vee b, a\}$. Das neue Log enthält nun aber keine Information darüber, dass die Datenbank zwingend b enthält, obwohl dies, wie im Beispiel gesehen, eindeutig vom Benutzer gefolgert werden kann.

Beispiel 9.1.2 (Verletzung von Konsistenzbedinungen)**:**
In eine gegebene Datenbankinstanz $db := \{\neg a, b\}$*, deren Schema der Konsistenzbedingung* $(\neg a \vee b)$ *unterliegt, möchte ein Benutzer erneut den Wert a einfügen. Dieser Änderungswunsch würde in diesem Fall vom System mit einer entsprechenden Meldung abgelehnt werden, da ansonsten eine inkonsistente Datenbank* $db' = \{a, b\}$ *erzeugt werden würde. Aus dieser Ablehnung hätte der Benutzer erneut die Möglichkeit zu folgern, dass b wahr sein muss, denn unter* $\neg b$ *hätte die Änderung des Wertes a keine Verletzung der Konsistenz zur Folge.*

Wäre b wieder ein Geheimnis, so würde die Abweisung der Benutzer-Operation mit entsprechender Meldung des Systems erneut eine Inferenz nachsichziehen. Würde die Änderung allerdings durchgeführt, hätte dies eine inkonsistente Datenbankinstanz zur Folge. Auch hier stellt sich erneut die Frage der korrekten Log-Aktualisierung. Ein einfaches Hinzufügen von $\neg a$ (a wurde ja nicht eingefügt) würde erneut nicht ausreichen. Das resultierende neue $log' = \{\neg a \vee \neg b, \neg a\}$ enthält wiederum keine Aus-

sage über die Ausprägung von b, obwohl die Wahrheit, wie oben gezeigt, eindeutig von einem Benutzer gefolgert werden kann.

9.2 Logik negierter Variablen

Wie wir im letzten Abschnitt anhand der beiden Beispiele gesehen haben, kann eine falsche Log-Aktualisierung potentiell gefährliche Inferenzen nach sich ziehen. In diesem Abschnitt sollen nun zunächst einige Definitionen und grundlegende Möglichkeiten vorgestellt werden, um solche Inferenzen für ein System erkennbar und somit auch abwehrbar zu machen.

9.2.1 Formel und Variablennegation

Definition 9.2.1 (Formeln)**:**
Die Menge FORMULA ist definiert als Menge aller nach folgender Vorschrift konstruierbaren Formeln:

- *true, false, v sind in FORMULA, wobei* $v \in VAR$ *stellvertretend für eine beliebige Variable ist*
- *für* $t_1, t_2 \in FORMULA$ *sind auch folgende Formeln in FORMULA*
  - $\neg t_1$
  - $(t_1 \wedge t_2)$
  - $(t_1 \vee t_2)$

Während eine Variable $v \in VAR$ nur einen Platzhalter darstellt, deren Wert erst mit der Angabe einer Belegung festgelegt wird, werden Literale als Angabe einer solchen Belegung benutzt.

Definition 9.2.2 (Literale)**:**
Formal definiert wird ein Literal aus der Menge LIT durch:

- *Für jede Variable* $v \in VAR$ *seien* v *und* $\neg v$ *Literale der Menge LIT*

- *Für ein Literal* $\chi \in \{v, \neg v\}$ *sei* $\chi^+ := v \in VAR$

Mit anderen Worten ist χ^+ *also syntaktisch identisch zu der Variable, deren Belegung durch das Literal* χ *angegeben wird.*

Definition 9.2.3 (Variablennegation auf einer Formel)**:**
Die Variablennegation auf einer Formel

$$neg(\cdot,\cdot) : FORMULA \times LIT \longrightarrow FORMULA$$

ist definiert durch

$$neg(\phi,\chi) := \begin{cases} \phi, & \text{für } \phi \in \{false, true\} \\ \phi, & \text{für } \phi = v \neq \chi^+, v \in VAR \\ \neg\phi, & \text{für } \phi = \chi^+ \\ \neg(neg(\phi^{'},\chi)), & \text{für } \phi = \neg\phi^{'} \\ (neg(\phi^{'},\chi) \wedge (neg(\phi^{''},\chi)), & \text{für } \phi = \phi^{'} \wedge \phi^{''} \\ (neg(\phi^{'},\chi) \vee (neg(\phi^{''},\chi)), & \text{für } \phi = \phi^{'} \vee \phi^{''} \end{cases} \tag{9.1}$$

Wir sehen, dass obwohl Variablen in den jeweiligen Formeln negiert werden sollen, neben der Formel ϕ keine Variable, sondern ein Literal χ übergeben wird. Dies hat den Grund, dass der Benutzer später durch ein Literal die von ihm gewünschte Belegung einer Variablen angibt. Der Übergang von diesem Literal zur Variable innerhalb einer Formel wird wie oben gezeigt vollzogen.

Im Weiteren wird die vorgestellte Variablennegation auf einer Formel auf eine Menge von Formeln ausgeweitet.

Definition 9.2.4 (Variablennegation auf einer Menge von Formeln)**:**
Die Variablennegation auf einer Menge von Formeln

$$neg(\cdot,\cdot) : \mathcal{P}(FORMULA) \times LIT \longrightarrow \mathcal{P}(FORMULA)$$

ist definiert durch

$$neg(M,\chi) := \{neg(\phi,\chi) \mid \phi \in M\}$$

Es wird also einfach für jede Formel aus der Menge M, jedes Vorkommen der durch χ *spezifizierten Variable negiert.*

9.2.2 Eigenschaften der Variablennegation auf Formeln

Mit dem folgenden Lemma folgt nun die Kernaussage über die Logik negierter Variablen, welches bei der Untersuchung und Überprüfung von Inferenzen, die durch die Veränderung einer Datenbankinstanz bzw. einer Benutzersicht entstehen können, von Bedeutung sein wird.

Lemma 9.2.1:
Für vollständige Datenbankinstanzen db, alle $\phi \in FORMULA$, sowie ein beliebiges Literal χ und für folgende Definition von db^χ durch db,

$$db^\chi := \begin{cases} (db \setminus \{\chi\}) \cup \{\neg\chi\}, & \textit{für } \chi \in db \\ (db \setminus \{\neg\chi\}) \cup \{\chi\}, & \textit{sonst} \end{cases} \tag{9.2}$$

gilt:

$$eval(\phi)(db) = eval(\neg(\phi, \chi))(db^\chi)$$

Informell bekommt man also gleiche Ergebnisse bei der Auswertung einer Formel ϕ auf einer Datenbank und der χ-negierten Formel auf derjenigen Datenbank, die durch Negation der Belegung von χ entstanden ist.

9.2.3 Anwendung der Variablennegation

Die gewonnenen Erkenntnisse über die Eigenschaften der Variablennegation werden nun auf die in Kapitel 1 vorgestellten Beispiele angewandt, um eine korrekte Log-Aktualisierung, ohne die dort erwähnten Inferenzen, zu gewährleisten.

Zuvor definieren wir allerdings noch die Menge der Konsistenzbedingungen constraints:

Definition 9.2.5 (Konsistenzbedingungen constraints)**:**
constraints bezeichne eine endliche Menge der im jeweiligen Datenbankschema einzuhaltenden Konsistenzbedingungen. Die Menge wird hier als Menge von immer (vor und nach jeder Änderung) gültigen Aussagen betrachtet, während das sonstige Wissen $log_0 \setminus constraints$ Änderungen unterliegen kann. Weiterhin gilt, dass die Menge der Konsistenzbedingungen dem Benutzer bekannst ist, also: $constraints \subseteq log_0$.

Beispiel 9.2.1 (Erfolgreiches Ändern von Werten)**:**
In diesem Beispiel hatte ein Benutzer erfolgreich den Wert a in eine Datenban-

kinstanz mit der Konsistenzbedingung $(a \vee b)$ *eingefügt, wodurch ihm eine Inferenz bezüglich des Wertes von* b *ermöglicht wurde.*

$$\begin{aligned} log^{'} &= neg(log, a) \cup \{a\} \cup constraints \\ &= \{\neg a \vee b\} \cup \{a\} \cup \{a \vee b\} \\ &= \{\neg a \vee b, a, a \vee b\} \equiv \{a, b\} \end{aligned}$$

Wir sehen, dass das neue $log^{'}$ nun auch die vom Benutzer folgerbare Information, dass b wahr ist enthält. Die Menge $neg(log, \chi)$ enthält dabei die nach Änderung von χ gültigen Aussagen, wenn diese in der ursprünglichen Menge log vor der Änderung auch gültig waren. Durch die Negation von χ im Log bei einer erfolgreichen Änderung enthält es genau die Sicht, die auch ein Benutzer bezüglich einer Neuauswertung auf der neuen Instanz erhalten würde.

Beispiel 9.2.2 (Verletzung von Konsistenzbedinungen)**:**
Im zweiten vorgestellten Beispiel versuchte ein Benutzer in eine Instanz $db := \{\neg a, b\}$ *mit der Konsistenzbedingung* $(\neg a \vee \neg b)$ *ebenfalls den Wert* a *einzufügen.*

$$\begin{aligned} log^{'} &= log \cup \{\neg a\} \cup \{neg(\neg(\bigwedge_{\phi \in constraints} \phi), a)\} \\ &= \{\neg a \vee b\} \cup \{\neg a\} \cup \{neg(\neg(\neg a \vee \neg b), a)\} \\ &= \{\neg a \vee \neg b, \neg a\} \cup \{\neg(a \vee \neg b)\} \\ &= \{\neg a \vee \neg b, \neg a, \neg(a \vee \neg b)\} \\ &\equiv \{\neg a \vee \neg b, \neg a, \neg a \wedge b\} \equiv \{\neg a, b\} \end{aligned}$$

Wiederum enthält das neue $log^{'}$ nun das, vom Benutzer folgerbare, Wissen, dass b wahr ist. Die Formel $neg(\neg(\bigwedge_{\phi \in constraints} \phi), \chi)$ beschreibt hierbei, dass die jeweils vorhandenen Konsistenzbedingungen nicht mehr gelten, wenn der Wert von χ geändert wird. Alle im Log enthaltenen Fakten behalten ihren Wahrheitswert in der neuen Datenbankinstanz, wenn jedes Vorkommen der in der Instanz geänderten Variable auch in alle Formeln des Logs negiert wird.

9.3 Kontrollierte Anfragen mit Sicht-Änderungen

Wäre b in den beiden Beispielen wieder ein Geheimnis gewesen, kämen die Erkenntnisse über eventuelle dem Benutzer ermöglichte Inferenzen zum Zeitpunkt der Log-Aktualisierung natürlich zu spät. Daher werden die vorgestellten Mechanismen nun hier eingesetzt, um mögliche Inferenzen einer Änderung oder Abweisung im Voraus zu berechnen. Der folgende Algorithmus ermöglicht somit inferenzfreie Sicht-Änderungen.

Definition 9.3.1:
Kontrollierte Anfragen mit Sicht-Änderungen] Wir definieren eine Folge Q von Anfrage bzw. Änderungs-Operationen:

$$Q = \langle \Theta_1, \Theta_2, ..., \Theta_i, ...\Theta_k \rangle$$

mit

$$\Theta_i = \begin{cases} \Phi_i, & \text{eine Anfrage} \\ update(\chi_i), & \text{eine Änderungs-Operation} \end{cases}$$

Dabei ist $\Phi_i \in FORMULA$ und $\chi_i \in LIT$.

Weiterhin sind:

- *$constraints \subseteq \mathcal{P}(FORMULA)$ eine endliche Menge von Konsistenzbedingungen*
- *$log_0 \subseteq \mathcal{P}(FORMULA)$ ein initiales Benutzerwissen mit $log_0 \supseteq constraints$,*
- *db_0 eine initiale (vollständige) Datenbankinstanz und*
- *$pot_sec \subseteq \mathcal{P}(FORMULA)$ eine endlich Menge von potentiellen Geheimnissen.*

Die kontrollierte Anfrageauswertung mit Sicht-Änderung ist dann wie folgt definiert:

$$control_eval_update(Q, log_0)(db_0, pot_sec)$$

$$= \langle (ans_1, log_1, db_1), ..., (ans_i, log_i, db_i), ..., (ans_k, log_k, db_k) \rangle$$

Für Anfragen Θ_i sei das Ergebnistripel (ans_i, log_i, db_i) dabei definiert wie in der kontrollierten Anfrageauswertung ohne Updates $control_eval$ mit zusätzlich $db_i := db_{i-1}$. Änderungen $update(\chi_i)$ werden durch den folgenden Algorithmus definiert.

9.3.1 Ein sicherer Algorithmus zur Sicht-Änderung

Es wird nun ein Algorithmus vorgestellt, der sowohl die am Anfang definierten Verfügbarkeitsinteressen des Benutzers als auch das Vertraulichkeitsinteresse des Administrators berücksichtigt. Zunächst sollen die vier Fälle, aus dem sich der Algorithmus zusammensetzt, kurz vorgestellt und erläutert werden. Dabei wird sequentiell geprüft, ob die jeweiligen Bedingungen für das Eintreten eines Falles gegeben sind. Der Algorithmus basiert auf dem Lügenansatz, wodurch also auch die Disjunktion der potentiellen Geheimnisse geschützt wird.

Zuvor definieren wir allerdings noch analog zu eben dieser Disjunktion der potentiellen Geheimnisse pot_sec_disj eine andere Darstellung für die Konjunktion der Konsistenzbedingungen:

$$con_conj := \bigwedge_{\phi \in constraints} \phi$$

Definition 9.3.2 (Konjunktion der Konsistenzbedingungen)**:**

1. Im ersten Fall soll überprüft werden, ob aus Sicht des Benutzers der Zustand, den dieser mit seiner Änderungs-Operation erzeugen will, bereits gilt. Ist dies der Fall, soll bereits hier der Algorithmus terminieren und die Änderung abgebrochen werden. Es ist also zu prüfen, ob für die Benutzer-Operation $update\langle\chi_i\rangle$ das Faktum χ_i schon gilt.
   Dies ist wiederum genau dann der Fall, wenn
   $$control_eval(\langle\chi_i\rangle, log_{i-1})(db_{i-1}, pot_sec) = \langle\chi_i\rangle$$
   zutrifft. Nach Definition von $control_eval$ unter dem Lügen-Zensor kann man zwei Bedingungen ableiten, bei deren einzelnem Zutreffen die obige Bedingung erfüllt ist:
   $$eval^*(\chi_i)(db_{i-1}) = \chi_i \textit{ And } log_{i-1} \cup \{\chi_i\} \not\models pot_sec_disj$$
   $$\textit{Or } eval^*(\chi_i)(db_{i-1}) = \neg\chi_i \textit{ And } log_{i-1} \cup \{\neg\chi_i\} \models pot_sec_disj$$

Die erste Bedingung beschreibt den Fall, dass χ_i tatsächlich vorhanden ist und diese Information nicht die Disjunktion der möglichen Geheimnisse implizieren würde, während die zweite Bedingung die Situation beschreibt, dass $\neg\chi_i$ wahr ist, allerdings gelogen werden muss, da diese Information zusammen mit dem bisherigen Benutzer-Wissen die Geheimnis-Disjunktion implizieren würde.
Tritt eine dieser zwei Bedingungen ein, wird die Benutzer-Änderung abgebrochen und das log durch Einfügung von χ_i entsprechend aktualisiert.

2. Falls allerdings keine dieser zwei Bedingungen erfüllt ist, wird nun im zweiten Fall überprüft, ob aus Sicht des Benutzers, bei einer erfolgreichen Änderung die Konsistenz verletzt oder die Disjunktion der Geheimnisse impliziert wird. Um dies verifizieren zu können, muss das Wissen, welches durch eine erfolgreiche Änderung vom Benutzer gefolgert werden könnte, berechnet werden: Dies ist einerseits die immer gültige Menge der Konsistenzbedingungen constraints und andererseits das Wissen, dass nun natürlich auch der vom Benutzer veränderte Wert χ_i gilt. Da aus Sicht des Benutzers durch das Nicht-Eintreten des ersten Falls gilt, dass bezüglich χ_i eine echte Änderung durchgeführt wird, ist die Variablennegation auf seine Sicht, also das log anwendbar. Insgesamt ist also folgende Bedingung zu überprüfen:

$$neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints \models pot_sec_disj$$

Anzumerken ist, dass diese Bedingung ohne Zugriff auf die tatsächliche db-Instanz ausgewertet wird. Dies bedeutet, dass der Benutzer dies selbst berechnen kann, wodurch das Ergebnis dieser Auswertung, also eine eventuelle Konsistenz- oder Geheimnisverletzung, auch offen mitgeteilt werden kann.
Terminiert der Algorithmus hier mit einer entsprechenden Meldung an den Benutzer, ist der einzige Wissensgewinn für diesen, dass offenbar der erste Fall erfolgreich passiert wurde. In der Datenbank galt also aus Sicht des Benutzers $\neg\chi_i$, was durch eine entsprechende Aktualisierung wieder im Log festgehalten werden muss.

3. Während im Vorgängerfall nur Konsistenz- und Geheimnisverletzungen, die ein Benutzer auf Grund zuvor erhaltener Systemantworten selbst folgern konnte, überprüft wurden, sollen jetzt auch die Konsistenzverletzungen getestet werden, die er aufgrund fehlender Informationen nicht selbst erkennen bzw. berechnen kann.

Die Konsistenz wird nun also unter Zuhilfenahme der (zukünftigen) db-Instanz überprüft, was durch die Bedingung

$$eval(con_conj)\ ((db_{i-1} \setminus \{\neg\chi_i) \cup \{\chi_i\}) = False$$

zum Ausdruck gebracht werden kann. Hier wird die Konjunktion der Konsistenzbedingungen auf der Instanz ausgewertet, wie sie ein Benutzer nach einer Änderung erwarten würde. Wie unser zweites Beispiel bereits zeigte, kann die Meldung einer entsprechenden Verletzung allerdings auch zu unerwünschten und gefährlichen Inferenzen führen. Deswegen muss eine weitere Bedingung erfüllt sein, damit der Algorithmus hier mit einer entsprechenden Meldung an den Benutzer abbrechen kann.

$$log_{i-1} \cup \{\neg\chi_i\} \cup \{neg(\neg con_conj, \chi_i)\} \not\models pot_sec_disj$$

Nur wenn beide Bedingungen gleichzeitig erfüllt sind, darf die Konsistenzverletzung gemeldet werden. Des Weiteren muss wieder das Log korrekt modifiziert werden. Erstens muss $\neg\chi_i$ hinzugefügt werden, da χ_i ja nicht eingefügt wurde, und zweitens muss mit $neg(\neg con_conj, \chi_i)$ festgehalten werden, dass unter Änderung des Wertes von χ_i eine Konsistenzverletzung auftreten würde.

4. Nachdem alle vorangegangen drei Fälle des Algorithmus erfolgreich passiert wurden, erfolgt nun im letzten Schritt aus Sicht des Benutzers seine gewünschte Änderung $update(\chi_i)$. Hier ist jedoch eine Fallunterscheidung notwendig, da wir bereits gesehen haben, dass es Änderungen gibt, die zwar eine Verletzung der Konsistenz zur Folge haben, was allerdings auf keinen Fall dem Benutzer mitgeteilt werden darf. Aus diesem Grund wird zwischen konsistenzverletzender und -erhaltender Änderung unterschieden.
   In dem ersten Fall wird keine modifizierte db-Instanz erzeugt, wodurch die Konsistenz der Datenbank natürlich erhalten bleibt. Die Änderung wird ausschließlich durch eine entsprechende Log-Aktualisierung erreicht. Hier wird dem Benutzer also eine konsistente Datenbank vorgelogen, obwohl die Änderung tatsächlich eine Inkonsistenz nach sich ziehen würde.
   Falls die Konsistenz der Datenbank allerdings nicht verletzt wird, erzeugt die Änderungs-Operation eine neue Datenbankinstanz.

9.3.2 Änderungs-Algorithmus für atomare Fakten

Zum Schluss fassen wir unsere Beobachtungen und Erkenntnisse zusammen und formalisieren den oben beschriebenen Alogrithmus.

Definition 9.3.3 (Änderungs-Algorithmus für atomare Fakten)**:**
Die gewünschte Änderungs-Operation update(χ_i) des Benutzers unter dem Lügenzensor wird durch sequentielle Überprüfung der bereits vorgestellten 4 Fälle durchgeführt. Bei Zutreffen der jeweils angegeben und schon angesprochenen Bedingungen wird der entsprechende Fall aktiv und die unter ihm aufgeführten Aktionen durchgeführt. Mit der Antwort ans_i an den Benutzer terminiert der Algorithmus dann jeweils.

Fall 1 *(Wert bereits enthalten)*

$$eval^*(\chi_i)(db_{i-1}) = \chi_i \text{ And } log_{i-1} \cup \{\chi_i\} \not\models pot_sec_disj$$
$$\text{Or } eval^*(\chi_i)(db_{i-1}) = \neg\chi_i \text{ And } log_{i-1} \cup \{\neg\chi_i\} \models pot_sec_disj$$

- $db_i := db_{i-1}$
- $log_i := log_{i-1} \cup \{\chi_i\}$
- $ans_i := \chi_i$ *bereits in der Datenbank enthalten.*

Fall 2 *(vom Benutzer berechenbare Konsistenz- oder Geheimnisverletzung)*

$$neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints \models pot_sec_disj$$

- $db_i := db_{i-1}$
- $log_i := log_{i-1} \cup \{\neg\chi_i\}$
- $ans_i := \chi_i$ *verletzt Konsistenz bzw. Geheimnis.*

Fall 3 *(vom Benutzer nicht berechenbare Konsistenzverletzung)*

$$eval(con_conj)\ ((db_{i-1}\ \setminus \{\neg\chi_i) \cup \{\chi_i\}) = False$$
$$And\ log_{i-1} \cup \{\neg\chi_i\} \cup \{neg(\neg con_conj, \chi_i)\} \not\models pot_sec_disj$$

- $db_i := db_{i-1}$
- $log_i := log_{i-1} \cup \{\neg\chi_i\} \cup \{neg(\neg con_conj, \chi_i)\}$
- $ans_i := \chi_i$ *verletzt Konsistenz.*

Fall 4 *(Änderung wird durchgeführt)*
Erfolgt keine Abweisung der Änderungs-Operation in den bisher genannten Fällen, so wird die Änderung durchgeführt.

Fall 4.1 *(Konsistenz verletzt)*
Wird in Fall 3 die Konsistenz zwar verletzt, die Meldung dieser Verletzung würde allerdings die Geheimnis-Disjunktion implizieren:

- *Polyinstantiiere (= nehme* χ_i *nur in* log_i *auf)*
- $db_i := db_{i-1}$
- $log_i := neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints$
- $ans_i := \chi_i$ *wurde eingefügt.*

Fall 4.2 *(Konsistenz nicht verletzt)*
Ist die Konsistenz der Datenbank, wie im Fall 4.1 angegeben, nicht verletzt, so erzeugt die Änderung des Benutzers eine neue Instanz:

- $db_i := (db_{i-1} \setminus \{\neg\chi_i\}) \cup \{\chi_i\}$
- $log_i := neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints$
- $ans_i := \chi_i$ *wurde eingefügt.*

Es wird also in beiden Fällen dem Benutzer eine erfolgreiche Änderung signalisiert.

Kapitel 10

Inferenzfreie Sichterneuerung

Als Grundlage für diese Ausarbeitung diente [Gog08]

10.1 Motivation

Die Sichterneuerung bezieht sich auf die Erneuerung der Sicht des Benutzers nach einer Aktualisierungsoperation des Administrators auf der aktuellen Datenbankinstanz. Hierbei wird die Sicht des Benutzers durch das Benutzerlog repräsentiert, welches das aktuelle Wissen des Benutzers berücksichtigt, um vor ihm geheim zu haltende Fakten zu verbergen. Bei der Sichterneuerung muss differenziert betrachtet werden, welche Änderungen dem Benutzer mitgeteilt werden, da er sonst möglicherweise Geheimnisse folgern kann.

10.1.1 Was ist eine Sicht?

Eine Sicht ist eine aus der Basisrelation abgeleitete neue Relation. Diese wird mittels einer Ableitungsfunktion definiert und liefert einen Ausschnitt der Daten eines Informationssystems. Sie ist das Ergebnis einer Anfrageauswertung.

Das Benutzerlog der kontrollierten Anfrageauswertung ist aus der Sicht des Benutzers ebenfalls ein Ausschnitt der aktuellen Datenbankinstanz, d.h. eine Sicht.

Zusätzlich ist es, weil es explizit gespeichert wird, eine materialisierte Sicht.

Im Normalfall sind die definierten Sichten als virtuelle Sichten realisiert, d.h. durch Angabe einer Ableitungsfunktion. Bei jeder Anfrage werden diese über die Ableitungsregeln neu berechnet.

10.2 Materialisierte Sichten

10.2.1 Definition

Materialisierte Sichten sind die physikalische Speicherung der bei der Anfrageauswertung entstehenden virtuellen Sichten. Das von uns betrachtete Benutzerlog ist eine materialisierte Sicht des aktuellen Benutzerwissens.

10.2.2 Anwendungsbereiche

Anfragezeitoptimiertung

Im Gegensatz zu virtuellen Sichten, bei denen jede Anfrage neu ausgewertet werden muss, entfallen durch materialisierte Sichten die ständigen Neuauswertungen der Ableitungsregeln. Dadurch werden zeitintensive Anfragen im voraus berechnet.

Datenlager

Datenlager (engl. data warehouse) sind eine besondere Art Informationssystem für betriebswirtschaftliche Überlegungen. Hierbei handelt es sich um eine zentrale Analysedatenbank aus Datenbeständen verteilter und meist heterogener Quellen. Das Ziel hierbei ist eine ganzheitliche Sicht über alle Datenquellen zu erhalten. Dies kann man auf zwei Arten realisieren. Zum einen über virtuelle Datenlager. Bei diesem Ansatz ist das System eine reine Schnittstelle zwischen den einzelnen Quellen. Die Anfragen an das Datenlager werden in der Quelldatenbank umgesetzt und einheitlich ausgewertet an den Benutzer zurückgeliefert. Der Nachteil dieser Variante

sind die Ineffizienz und der enorme Zeitaufwand. Als Ausgleich dazu entstand der zweite Ansatz. Hierbei speichert man alle Daten der einzelnen Quellen in überarbeiteter Form im Datenlager selbst.

Replikation in verteilten Datenbanken

Hier erlauben materialisierte Sichten das Führen von redundanten Informationen auf verteilten Rechnern.

10.2.3 Das Sichtwartungsproblem

Das Problem der Sichtwartung beschreibt die Tatsache, dass sich für materialisierte Sichten das Problem ergeben kann, dass diese von einer Änderung der Basisrelation betroffen sein kann. Das Ziel liegt nun darin ein geeignetes Aktualisierungsverfahren zu finden, das dieses Problem berücksichtigt. Hierzu müssen allerdings die folgenden Punkte beachtet werden:

Zeitpunkt der Aktualisierung Der Zeitpunkt der Aktualisierung muss in das Verfahren aufgenommen werden, da hier je nach Wahl des Zeitpunktes eventuell Inkonsistenzen entstehen können. Man hat die Wahl zwischen drei verschiedenen Ansätzen. Zum einen kann man die Aktualisierung direkt an eine Änderung an der Basisrelation koppeln. Hierdurch wird Konsistenz gewährleistet. Zum anderen kann man eine Aktualisierung von Ereignissen abhängig machen. Schliesslich kann man auch vor jedem Zugriff auf die materialisierte Sicht prüfen lassen, ob sich die Basisrelation geändert hat. Bei den letzten beiden Ansätzen sind die materialisierten Sichten allerdings zeitweise inkonsistent, da diese von Ereignissen bzw. Zugriffen abhängig sind und diese nicht zwangsläufig zeitnah zur Änderung geschehen.

Vorbedingung einer Aktualisierung Daher kann man als Vorbedingung einer Aktualisierung die Prüfung, ob durch die Änderung an der Basisrelation auch die materialisierte Sicht betroffen ist, festhalten. Nur in diesem Fall sollte dann die Sicht Benutzers geändert werden.

Neuauswertung oder inkrementelle Aktualisierung Eine Neuberechnung der Ableitungsfunktion kann unter Umständen sehr ineffizient sein. Als Alternative dazu betrachtet man inkrementelle Aktualisierungsalgorithmen. Das Ziel hierbei ist, dass die Änderung auf der Sichtebene dasselbe Ergebnis liefert wie eine Neuauswertung der Ableitungsfunktion auf der Basisrelation. Je nach vorliegendem Ansatz sind allerdings gewisse Einschränkungen zu beachten. Die Einschränkungen können die Informationen, Änderungsoperationen, Sprachliche Mittel oder Datenbankinstanzen betreffen.

Einschränkung der Informationen Hierbei wird betrachtet welche Informationen zur Berechnung der Unterschiede zwischen Basisrelation und Sicht zur Verfügung stehen: Sind semantische Bedingungen bekannt, Existieren weitere Beziehungen?

Einschränkung der Änderungsoperationen Hierbei wird betrachtet welchen entscheidenden Einfluss dies auf die Änderungsalgorithmen hat, welche Operationen erlaubt sind, z.B. Löschen oder Einfügen

10.3 Sichterneuerung unter atomaren Änderungen

10.3.1 Grundidee atomarer Änderungen

Hier betrachten wir einen ersten algorithmischen Ansatz unter Einschränkungen. Der Algorithmus wird auf atomare Änderungen eingeschränkt, d.h. der Administrator soll in einem Änderungsschritt nur genau ein Literal ändern dürfen. Hierbei werden wir sogenannte Operationsfolgen betrachten. Diese sind die Menge der Operationen beider Parteien, d.h. sowohl die Operationen des Administrators, als auch die Operationen des Benutzers werden hier in der Reihenfolge in der sie durchgeführt werden, d.h. nicht nach Partei getrennt, zusammengefasst.

10.3.2 Korrektheit und Vertraulichkeit

Hierbei muss man beachten, dass der Benutzer gegebenenfalls über Änderungen neues Wissen erhält oder aber das Benutzerwissen nicht mehr zur aktuellen Datenbankinstanz passt. Eine erste Lösungsidee könnte hier sein, das Benutzerlog so zu bearbeiten, dass es weiterhin zur Sicht des Benutzers passt, die er durch Anfragen und Änderungsinformationen erhält. Allerdings entsteht hier das Problem, dass entweder die Korrektheit oder die Vertraulichkeit verletzt wird, wenn man dieses nicht erfasst.

Korrektheit

Um die Korrektheit zu wahren, betrachten wir zwei Ansätze.

- Der erste Ansatz ist die Reinitialisierung des Benutzerlogs. Bei diesem Ansatz wird das aktuelle Benutzerlog auf das initiale Benutzerlog zurückgesetzt, also alle bisher vom Benutzer gestellten Anfragen und Änderungsinformationen gelöscht.
- Der zweite Ansatz beschäftigt sich mit der Neuauswertung des Benutzerlogs. Hierbei wird zunächst das aktuelle Benutzerlog auf das initiale Benutzerlog zurückgesetzt und anschliessend werden alle vom Benutzer bisher gestellten Anfragen auf der aktuellen Datenbankinstanz neu ausgewertet.

Reinitialisierung des Benutzerlogs Die Reinitialisierung des Benutzerlogs liefert offensichtlich immer Antworten, die zur aktuellen Datenbankinstanz passen.
Wir betrachten auch hier wieder zwei Unterscheidungen.
Zunächst betrachten wir den Fall, dem Benutzer Änderungen nicht mitzuteilen, und anschliessend den Fall, den Benutzer über Änderungen zu informieren.

Änderungen nicht mitteilen Teilt man dem Benutzer die Änderungen nicht mit, so kann er bei gleichzeitiger Verwendung von aktuellen und veralteten Informationen fälschlicherweise die Gültigkeit von Geheimnissen folgern, wie man in [Gog08] sieht. Daher stellt dieser Ansatz keine Alternative dar, denn bei der kontrollierten

Anfrageauswertung gilt die Forderung, dass der Benutzer unter keinen Umständen Geheimnisse folgern darf.

Änderungen mitteilen Bei diesem Ansatz, muss der Benutzer nach einer Änderungsmitteilung davon ausgehen, dass die bisher von ihm gestellten Anfragen nicht mehr gültig sind. Allerdings kann der Benutzer dieses Wissen über die Rücksetzung auch dafür ausnutzen, durch Umstellung seiner alten Anfragen Metainformationen zu erhalten.

Beispiel 10.3.1:
$DS := \{a, b, s_1\}$
$pot_sec := \{s_1\}$
$db_0 := \{\neg a, b, \neg s_1\}$
$log_0 := \{\}$
$Q := \langle((a \vee s_1) \wedge b), a, b, update^{adm}(s_1)\rangle$

Anfrage Φ	$db_0 := \{\neg a, b, \neg s_1\}$	$db_4 := \{\neg a, b, s_1\}$
$((a \vee s_1) \wedge b)$	$\neg((a \vee s_1) \wedge b)$	$((a \vee s_1) \wedge b)$
a	$\neg a$	a
b	b	b

Anhand des Beispiels aus [Gog08] kann man sehen, dass die Reinitialisierung des Benutzerlogs nicht das aktuelle Wissen des Benutzers repräsentiert. Da hier davon ausgegangen wird, dass der Administrator bei jeder Aktualisierung nur einen Ausdruck ändern darf, sich für den Benutzer allerdings zwei Ausdrücke ändern, erfährt er, dass ein Geheimnis geschützt werden musste. Durch Umstellen der Anfragen könnte der Benutzer das Geheimnis erfahren. Daher betrachten wir nun die Neuauswertung des Benutzerlogs.

Neuauswertung des Benutzerlogs Durch die Neuauswertung aller bisher vom Benutzer gestellten Anfragen nach vorherigem Reinitialisieren des Benutzerlogs erhält man ein neues Benutzerlog, welches alle Antworten bezüglich der neuen Datenbankinstanz korrekt repräsentiert. Somit sind die Antworten auf Anfragen immer passend zur aktuellen Sicht des Benutzers und zu den bisherigen Anfragen des Benutzers. Durch das Informieren des Benutzers kann er in seinen Folgerungen auch

die Änderungen berücksichtigen und das Benutzerlog repräsentiert seine aktuelle Sicht. Hierdurch wird zusätzlich verhindert, dass der Benutzer durch Umstellung alter Anfragen neues Wissen erhält.

Definition 10.3.1 (Korrektheitsdefinition)**:**
Die Sicht des Benutzers entspricht immer der einer Neuauswertung aller seiner bereits gestellten Anfragen auf die aktuell gültige Datenbankinstanz nach vorherigem Rücksetzen des Benutzerlogs auf das initiale Benutzerlog log_0.

Vertraulichkeit

Zur Einhaltung der Vertraulichkeit müssen folgende Aussagen erfüllt sein:

- Der Benutzer darf über die initial vorliegende Datenbankinstanz keinen vollständigen Informationsgewinn erhalten.
- Der Benutzer kann die Gültigkeit eines Geheimnisses nicht eindeutig folgern.
- Der Benutzer darf auch nichts Vertrauliches aus Änderungsinformationen folgern können.

Definition 10.3.2:
Vertraulichkeit Sei ein System kontrollierter Anfrageauswertung mit administrativen Änderungen $control_eval_admin_update(Q, log_0)(db_0, pot_sec)$ *gegeben, so dass*

- *der Benutzer nur bei Sichtabweichung über eine Änderung informiert wird*
- *alle betrachteten Datenbankinstanzen die semantischen Bedingungen erfüllen*
- *der Geheimnisdisjunktionszensor unter potenziellen Geheimnisse verwendet wird*

Ein solches System bewahrt die Vertraulichkeit, gdw.

Für alle endlichen Operationsfolgen Q, 10.3.1

für alle möglichen Vertraulichkeitspolitiken pot_sec,

für alle initialen Benutzerlogs log_0,

für alle gültigen Datenbankinstanzen db_0,

so dass für (log_0, pot_sec) gilt,

dass das Benutzerlog die Disjunktion der potenziellen Geheimnisse nicht impliziert:

$log_0 \not\models pot_sec_disj$

Existiert zusätzlich eine Datenbankinstanz ${db_0}^{sec}$ und eine Operationsfolge Q', so dass gilt:

(a) *Unter beiden Datenbankinstanzen und Operationsfolgen werden immer dieselben Antworten geliefert.*

(b) *In allen betrachteten Datenbankinstanzen ${db_0}^{sec}$ unter Q' sind alle Geheimnisse nicht gültig.*

(c) *Die Anfragefolge und die beobachteten Änderungen sind in beiden Operationsfolgen (aufgrund der Mitteilungen) gleich. Die Anfragefolgen müssen in beiden Operationsfolgen vorkommen, da der Benutzer sonst Q' nicht als mögliche Operationsfolge folgern würde.*

Folgerungen Aus der Sicht des Benutzers ist eine initiale Datenbankinstanz ${db_0}^{sec}$ möglich, in der alle Geheimnisse nicht gelten. Es existiert eine Operationsfolge die nie die Gültigkeit von Geheimnissen folgern lässt. Also kann der Benutzer keinen gefährlichen Informationsgewinn bezüglich der Datenbankinstanz db_0 und der Operationsfolge Q erhalten.

10.3.3 Konstante Sicht bei atomaren Änderungen

Motivation

Der Grund, aus dem man die Sicht des Benutzers unter Umständen konstant halten möchte, ist der, dass man dem Benutzer auf Grund der Korrektheit eine Änderung

mitteilen muss, dies allerdings zur Verletzung der Vertraulichkeit führen kann.

Erlaubte sprachliche Mittel für Anfragen

Hier betrachten wir die Bedingungen und Einschränkungen der Anfragesprache, damit atomare Änderungen, aus denen ein Geheimnis folgt, keinen Einfluss auf die Sicht des Benutzers haben.

Satz 10.3.1:
Erlaubte sprachliche Mittel Wenn Benutzeranfragen auf Literale beschränkt sind und ${log_{i+1}}^{db_{i+1}} \cup \{X\} \models pot_sec_disj$ *für ein Literal* X *gilt, gibt die Datenbankinstanz* db_{i+1} *auf alle gestellten Anfragen immer die gleichen Antworten wie die Datenbankinstanz* db_i.

Erläuterungen zum Satz Dabei ist X das angefragte Literal, db_i die vorherige Datenbankinstanz, db_{i+1} die aktuelle Datenbankinstanz und ${log_{i+1}}^{db_{i+1}}$ das Benutzerlog zur aktuellen Instanz. Zusätzlich bedeutet ${log_{i+1}}^{db_{i+1}} \cup \{X\} \models pot_sec_disj$, dass das aktuelle Benutzerlog zur aktuellen Datenbankinstanz die Disjunktion der potenziellen Geheimnisse impliziert.

Beispiel 10.3.2:
$DS := \{a, b, s_1\}$
$pot_sec := \{s_1\}$
$db_i := \{\neg a, \neg b, \neg s_1\}$
$log_0 := \{\}$
$Q := \langle((a \wedge b) \vee s_1), a, b, update^{adm}(s_1)\rangle$

Durch eine Aktualisierungsoperation kann immer nur ein Literal geändert werden. Aufgrund der Neuauswertung werden allerdings zwei Literale in einer Aktualisierungsoperation geändert, sodass der Benutzer weiß, dass ein Geheimnis geschützt werden musste, wie man in Abbildung 10.1 aus [Gog08]

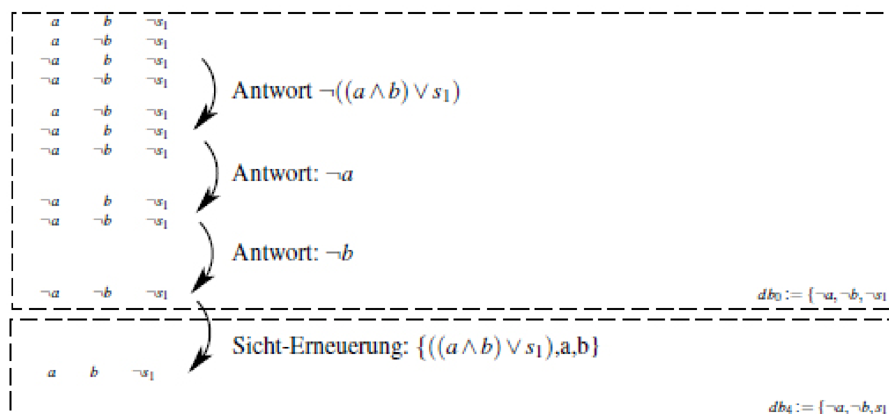


Abbildung 10.1: Mögliche sichere Datenbankinstanzen aus Sicht des Benutzers

Erlaubte sprachliche Mittel des Benutzerlogs

Hier wird betrachtet welche Auswirkung es haben kann, den Benutzer nach einer Änderung über die geänderte Aussage zu informieren.

Lemma zur Beschränkung des initialen Benutzerlogs Eine Änderungsmitteilung reicht nicht aus, um die Sicht des Benutzers immer gleich der einer Neuauswertung aller seiner bereits gestellten Anfragen auf die aktuell gültige Datenbankinstanz nach vorherigem Rücksetzen des Benutzerlogs auf log_0 zu halten. Nach einer Änderungsmitteilung kann es also sein, dass entweder die Korrektheit nicht mehr gilt, oder die Vertraulichkeit verletzt ist, wie in diesem Beispiel aus [Gog08] zu sehen ist.

Beispiel 10.3.3:
$DS := \{a, b, s_1\}$
$pot_sec := \{s_1\}$
$db_i := \{\neg a, \neg b, s_1\}$
$log_0 := \{(a \vee b \vee s_1)\}$
$db_{i+1} := update^{adm}(a)(db_i)$

Anfrage Φ	$db_i := \{\neg a, \neg b, s_1\}$	$db_{i+1} := \{a, \neg b, s_1\}$
a	$\neg a$	a
b	b	$\neg b$

Auch hier gilt wieder, der Administrator darf nur einen Ausdruck ändern. Durch die Erneuerung der Sicht des Benutzers werden hier allerdings zwei Ausdrücke geändert. Dies liefert dem Benutzer die Information, dass ein Geheimnis geschützt werden musste. Hierdurch wird die Vertraulichkeit verletzt.

Erlaubte sprachliche Mittel für potenzielle Geheimnisse

Hier wird die Menge der potenziellen Geheimnisse betrachtet.

Lemma zur Beschränkung der potenziellen Geheimnisse Nach einer Aktualisierung kann es passieren, dass es nicht ausreicht, dem Benutzer nur die geänderte Aussage mitzuteilen, um die Sicht des Benutzers immer gleich der einer Neuauswertung aller seiner bereits gestellten Anfragen auf die aktuell gültige Datenbankinstanz nach vorherigem Rücksetzen des Benutzerlogs auf log_0 zu halten, wie in diesem Beispiel aus [Gog08] zu sehen ist.

Beispiel 10.3.4: $DS := \{a, b\}$

$pot_sec := \{a \wedge b\}$

$db_i := \{\neg a, b\}$

$log_0 := \{\}$

$db_{i+1} := update^{adm}(a)(db_i)$

Anfrage Φ	$db_i := \{\neg a, b\}$	$db_{i+1} := \{a, b\}$
a	$\neg a$	a
b	b	$\neg b$

Auch hier gilt wieder, der Administrator darf nur einen Ausdruck ändern. Durch die Erneuerung der Sicht des Benutzers werden hier allerdings zwei Ausdrücke geändert. Dies liefert dem Benutzer die Information, dass ein Geheimnis geschützt werden musste. Hierdurch wird die Vertraulichkeit verletzt.

10.3.4 Atomare Änderungen in vollständigen Datenbanken

Aktualisierungsalgorithmus $update^{adm}(X)$

Eingabe:

zu aktualisierendes Literal X

gegebene Datenbankinstanz db_i

Menge potenzieller Geheimnisse pot_sec

aktuelles Benutzerlog log^{db_i}

Ausgabe:

Datenbankinstanz db_{i+1}

Änderungsinformation ans_{i+1}

neues Benutzerlog $log^{db_{i+1}}$

1.) Literal bereits in der Datenbankinstanz gültig:

Wenn $(eval^*(X)(db_i) = X)$ Dann

$db_{i+1} := db_i$

$ans_{i+1} = \{\}$

$log^{db_{i+1}} := log^{db_i}$

Information an den Administrator: Literal bereits gültig

Ende

Sonst

$db' := (db_i \backslash \{\neg X\}) \cup \{X\}$

2.) Die neue Datenbankinstanz db' erfüllt die semantischen Bedingungen nicht:

Wenn $\neg(db' model_of log_0)$ Dann

$db_{i+1} := db_i$

$ans_{i+1} = \{\}$

$log^{db_{i+1}} := log^{db_i}$

Information an den Administrator: Die semantischen Bedingungen werden nicht erfüllt.

Ende

3.) Die neue Datenbankinstanz db' erfüllt die semantischen Bedingungen:

$db_{i+1} := db'$

4.) Wann muss der Benutzer über eine Änderung informiert werden?

Wenn $(X \models pot_sec_disj)$ Dann

$ans_{i+1} = \{\}$

$log^{db_{i+1}} := log^{db_i}$

Information an den Administrator: Die Änderung wurde erfolgreich durchgeführt.

Ende

Sonst

$ans_{i+1} :=$ Wenn $log^{db_i} \models \neg X$ Dann $\{X\}$ Sonst $\{\}$

$log^{db_{i+1}} :=$ Wenn $ans_{i+1} = \{\}$ Dann log^{db_i} Sonst $(log^{db_i} \backslash \{\neg X\}) \cup \{X\}$

Information an den Administrator: Die Änderung wurde erfolgreich durchgeführt. Dem Benutzer ans_{i+1} mitteilen, sofern die Menge nicht leer ist.

Ende

Erläuterungen zu $update^{adm}(X)$

ans_{i+1} := Wenn $log^{db_i} \models \neg X$ Dann $\{X\}$ Sonst $\{\}$

Diese Zeile bedeutet, dass man dem Benutzer eine Änderung nur dann mitteilt, wenn er nicht schon bereits an das geänderte Literal glaubt beziehungsweise er diese bereits erfragt hat, sofern er aus seinem Wissen nicht bereits die geänderte Aussage folgern kann. Hat er ein Literal noch nicht erfragt, wird ihm eine Änderung eines solchen Literals auch nicht mitgeteilt.

Hierdurch wird kein Geheimnis gefährdet.

Teil II

Entwürfe

Kapitel 11

Entwurf des Mehrbenutzersystems

11.1 Einführung

Der Prototyp der PG495 basierte auf einem Einzelbenutzersystem, der derzeitige Prototyp wurde um ein Mehrbenutzersystem erweitert. Die Benutzer „jcqe_normal“ und „jcqe_lying“ wurden in Oracle angelegt und somit zwei Schemata erzeugt. In Oracle ist der Schemaname gleich dem Benutzernamen. Alle noch folgenden Erklärungen beziehen sich auf diese beiden Schemata. Dennoch sind beliebige Schemanamen der Form $< name >$ *_normal* und $< name >$ *_lying* vorgesehen (siehe Abschnitt 17). Ein neuer Schema-Entwurf muss also in doppelter Ausführung implementiert werden und die Schema- bzw. Benutzernamen müssen einmal den Suffix _normal und einmal _lying im Namen tragen. In beiden Schemata werden die notwendigen Tabellen verwaltet. Der Unterschied liegt darin, dass in jcqe_lying nur die Informationen der Benutzer mit dem Lügenzensor verwaltet werden. Diese Notwendigkeit ergibt sich aus der Tatsache, dass die anderen Zensoren (noch) nicht mit der Sichtänderung und Sichterneuerung umgehen können. Alle SQL-Anweisungen im Code wurden um den Zusatz „SQLInteraction.schema.$< table >$“ ergänzt, wobei *schema* eine statische Variable der Klasse SQLInteraction darstellt und zu Programmstart entweder mit jcqe_normal oder jcqe_lying belegt wird (dieses muss zur Zeit noch von Hand bei der Anmeldung geschehen). Ein Benutzer, der sich am Prototyp anmelden möchte, muss in der Benutzertabelle in dem Schema stehen, in dem er sich anmelden möchte. Wenn sich ein einfacher Benutzer mit Lügen-

zensor am Prototypen anmeldet, dann wird der Button „ViewUpdate“ angezeigt, anderenfalls nur der „Anfrage“ Button. Meldet sich allerdings ein Administrator am System an, dann wird statt dem „ViewUpdate“ Button, der „Update_Adm“ Button eingeblendet. Alle letzten 20 Anfragen, die ein Benutzer abschickt, werden in der $< user >$QueryHistory Tabelle gespeichert. Desweiteren werden alle Anfragen aller Benutzer in der globalen „QueryHistory“ Tabelle hinterlegt. Diese Funktion ermöglicht ein Monitoring.

11.2 Benutzer anlegen

Nur die Administratoren (zur Zeit nur die, die auch Besitzer der Schemata sind) haben das Privileg einen Benutzer anzulegen. Die Benutzer müssen über die GUI unter Benutzerverwaltung angelegt werden (dasselbe gilt für die weiteren Optionen bis auf „Benutzer entfernen“). Legt ein Administrator, d.h. jcqe_normal oder jcqe_lying, einen neuen Benutzer $< user >$ mittels der GUI an, so wird ein neuer Oracle-DB-Benutzer $< user >$, eine $< user >$Log, eine $< user >$PotSec und eine $< user >$QueryHistory Tabelle generiert, sowie alle notwendigen Privilegien vergeben. Desweiteren wird der neue Benutzer in die Benutzertabelle des jeweiligen Schemas geschrieben. Ein Benutzer mit Lügenzensor würde die Oracle-Rolle „jcqe_lying_role“, alle Benutzer mit anderen Zensortypen „jcqe_normal_role“ zugewiesen bekommen. Der Prototyp setzt die Existenz jener Schemata, Rollen und Privilegien voraus. Desweiteren können nur Benutzer hinzugefügt werden, nicht aber entfernt werden, da das Privileg Benutzer auf Oracle-Ebene entfernen zu dürfen, auch das Privileg inkludiert die System-User löschen zu dürfen. Ein weiteres Problem stellt zur Zeit die Definition der $< user >$-Geheimnisse und des $< user >$-Vorwissens dar. Sinnvoll wäre einen Benutzer erst nach Erstellung von Geheimnissen und Vorwissen durch den Admin freischalten zu können (weiteres siehe 12.3).

11.3 Benutzer entfernen

Löschvorgänge müssen also (Abschnitt 11.2) mit dem Oracle-Administrator(DBA) abgesprochen werden. Dabei müssen durch den DBA folgende Tabellen und Einträ-

ge mit entfernt werden:

- Oracle-Benutzer < *user* > löschen
- Benutzer < *user* > aus der Benutzertabelle im Schema jcqe_normal oder jcqe_lying löschen
- Tabellen < *user* >Log und < *user* >PotSec löschen
- Tabelle < *user* >QueryHistory löschen

11.4 Benutzer bearbeiten

Diese Option ermöglicht die Editierung beliebiger < *user* >PotSec- und < *user* >Log-Einträge durch die Administratoren. Der jeweilige Zensor kann nicht geändert werden, allerdings können die Rollen „einfacher Benutzer“ oder „Administrator“ neu vergeben werden.

11.5 Rollen- und Privilegienvergabe

Es wurden zwei Rollen auf Oracle-Ebene definiert. Einmal „jcqe_normal_role“ und „jcqe_lying_role“. Zur Automatisierung wurde dem *Schema*-Namen, also dem Oracle-Benutzernamen, der Suffix *_role* angehängt. Alle neuen Rollennamen sowie Schemanamen müssen sich an die Konventionen halten. Wenn also ein neues Schema verwendet werden soll, dann müssen diese Konventionen eingehalten werden, damit der Prototyp damit umgehen kann. Den Rollen wurden folgende Rechte zugewiesen:

- *jcqe_normal_role* :
  - SELECT,INSERT,UPDATE,DELETE auf < *user* >Log,< *user* >PotSec,< *tabelle_* >PS,< *tabelle_* >PS2 und Benutzertabelle. Sonst nur SELECT auf alle Tabellen im Schema jcqe_normal

- *jcqe_lying_role* :
    - SELECT,INSERT,UPDATE,DELETE auf alle Tabellen im Schema jcqe_lying

Desweiteren erhalten die Besitzer der Schemata (also zwei der Administratoren) die Rechte um Oracle-Benutzer anlegen zu können. Allen anderen Administratoren wird laut der GUI des Prototypen erlaubt, Benutzer hinzuzufügen. Dieses löst aber mangels Privilegien eine Exception aus (siehe 12.3).

11.6 ToDos

- Wenn man einen neuen Benutzer hinzufügen möchte, tauchen ab und zu Fehler (Exceptions) auf. Die Ursache(n) sollte gefunden und behoben werden. Es könnte an der Privilegienvergabe auf Oracle-Ebene liegen, muss aber nicht.
- Die GUI zur Benutzer Editierung sollte verbessert werden. Um die Funktionalität zu gewährleisten wurde kurzfristig eine veränderte GUI erstellt. Swing ermöglicht auf die schnelle keine Erstellung einer ansprechenden GUI, daher werden die Buttons auf FenstergröÃŸe skaliert. Es muss ein neuer (komplizierterer) LayoutManager Verwendung finden.

Kapitel 12

Entwurf der Sicht-Änderung

12.1 Motivation

12.1.1 Was ist das Problem?

Die Vorgängerprojektgruppe 495 hat (inferenzfreie) Sichtänderungen nicht berücksichtigt. Allerdings ist im Nachgang in der Diplomarbeit von Jens Seiler [Sei08] ein entsprechender Änderungsalgorithmus auf theoretischer Ebene entstanden. Dieser Algorithmus behandelt die Änderung der Sicht eines Benutzers nach einer Datenbankaktualisierung. Dabei wird zur Aktualisierung des Benutzerwissens aller Benutzer bei erfolgreicher Änderungsoperation die Methode $updateUserLogs()$ (Abschnitt 13.2.4) der Sicht-Erneuerung verwendet.

12.1.2 Warum ist das Problem ein Problem?

Durch eine solche update-Operation können gefährliche Inferenzen entstehen, was ausführlich in der Seminararbeit auf Seite 121 und in der Diplomarbeit von Seiler beschrieben ist. [Sch09, Sei08]

12.1.3 Wie sieht die Lösung aus?

Die theoretische Lösung wird bereits in der Diplomarbeit von Seiler erörtert, welche von uns umgesetzt wurde. Wir betrachten hier nur Änderungsoperationen, die mit Literalen arbeiten und je Änderungsoperation auch nur ein Literal ändern. Transaktionen wurden in der Arbeit zwar auch betrachtet, von uns jedoch im Rahmen dieser Projektgruppe nicht behandelt. Der Algorithmus überprüft sequentiell in 4 Fällen, ob eine solche Änderung überhaupt durchgeführt werden darf.

12.1.4 Wieso ist die Lösung eine Lösung?

Die theoretischen Einzelheiten hierzu sind in der Seminarausarbeitung von Schlotmann bzw. in der Diplomarbeit von Seiler zu finden, weshalb auf diese hier nicht noch einmal detailliert eingegangen werden soll. [Sch09, Sei08]
Der Korrektheitsbeweis des Änderungsalgorithmus findet sich sowohl in der Diplomarbeit von Seiler (Kapiel 5.4) selbst, als auch in dem Paper „Requirements and protocols for inference-proof interactions in information systems“ von Biskup, Gogolin, Seiler und Weibert. [Sei08, JBW08]

12.2 Operationalisierung

12.2.1 Aktivitätsdiagramm Sichtänderung

In Abbildung 12.1 auf Seite 155 wird der Ablauf des Algorithmus zur Sichtänderung aus der Diplomarbeit von Seiler [Sei08] dargestellt. Neu hierbei ist, dass auch die anderen Logs von Benutzern, die den Lügenzensor verwenden, nach einer update-Anfrage aktualisiert werden müssen. Dies wurde in der Diplomarbeit nicht berücksichtigt, da dort nur von einem Datenbankbenutzer ausgegangen wurde.
Wir gehen zusätzlich davon aus, dass der Administrator im Vorfeld die Konsistenzbedingungen für eine Datenbank festlegt, welche in einer Tabelle „Constraints“ festgehalten werden. Diese Bedingungen, welche vor und nach jeder Änderung gültig sein müssen, werden außerdem in jedes initiale log_0 von Benutzern des Lügenzensors über die Methode *addConstraintsToLog* (Abschnitt 12.2.11) eingefügt. Des-

weiteren wurde die Methode *checkconstraints* (Abschnitt 12.2.11) implementiert, welche kontrolliert, ob die vom Administrator gesetzen Konsistenzbedingungen auch wirklich in der Datenbank gelten. Falls dies nicht der Fall sein sollte, werden diese automatisch in die entsprechenden Tabellen eingefügt.

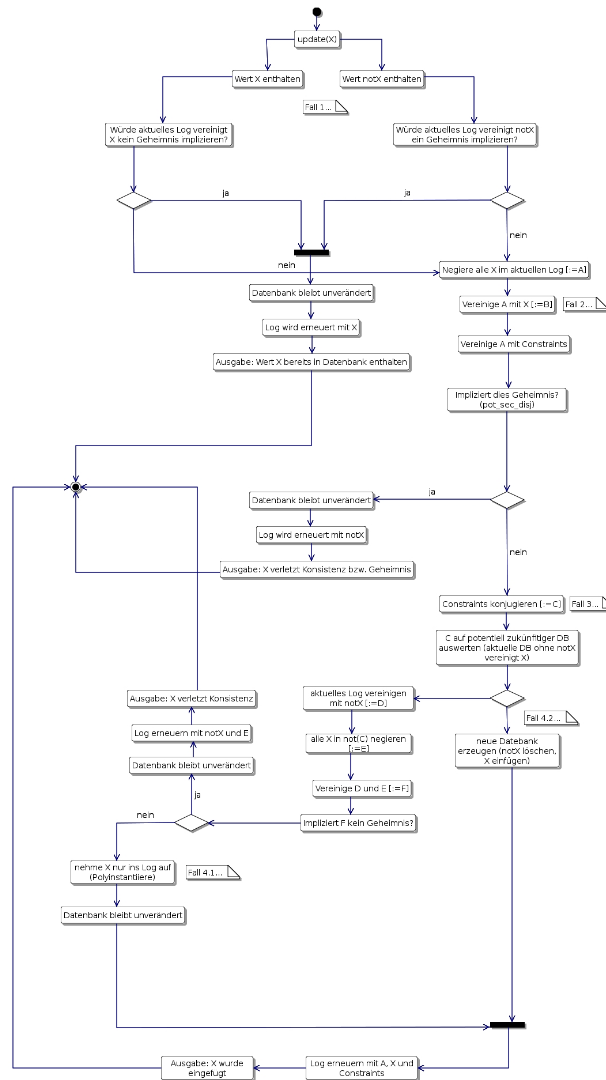


Abbildung 12.1: Aktivitätsdiagramm

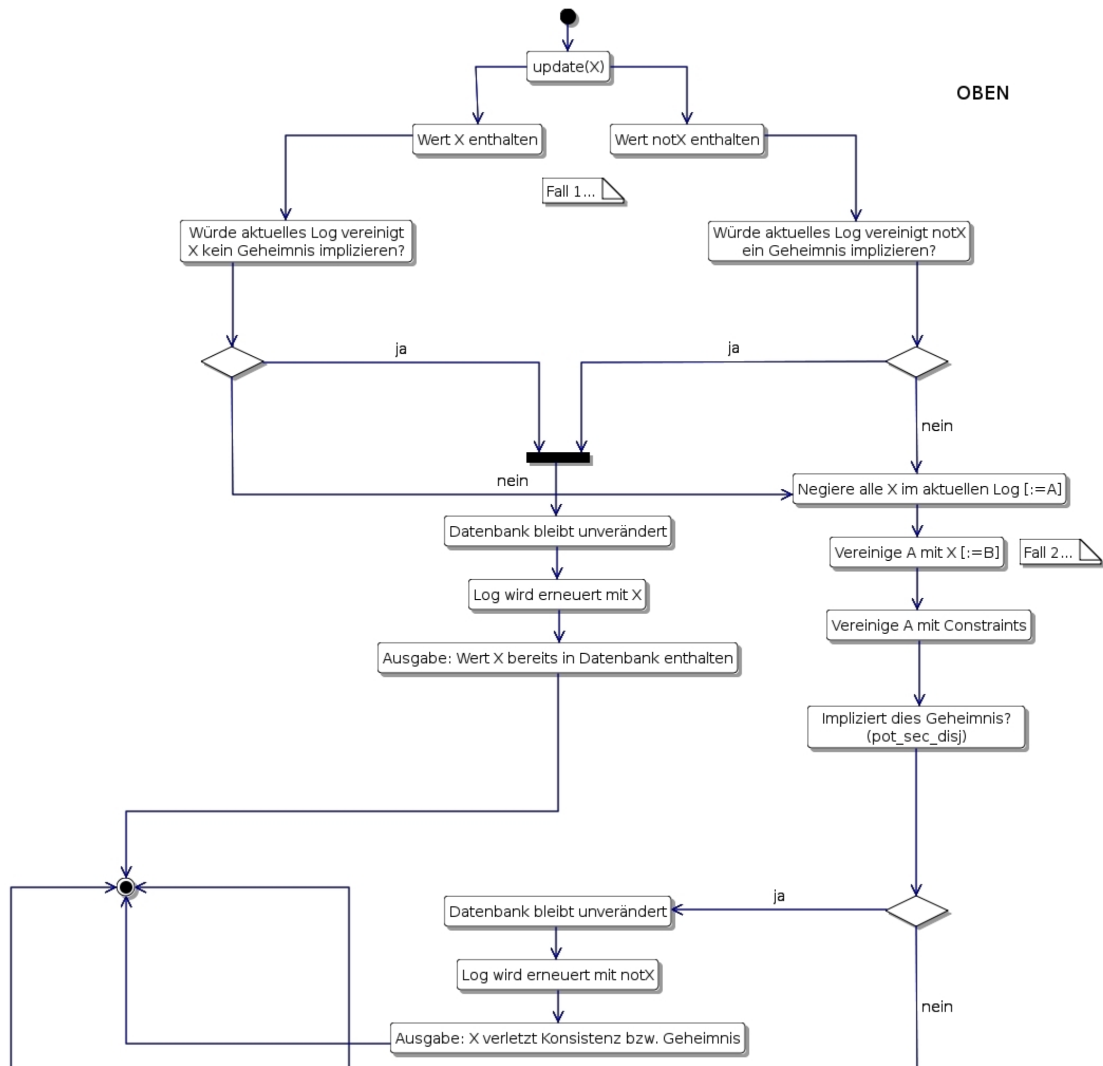


Abbildung 12.2: Aktivitätsdiagramm: oberer Teil

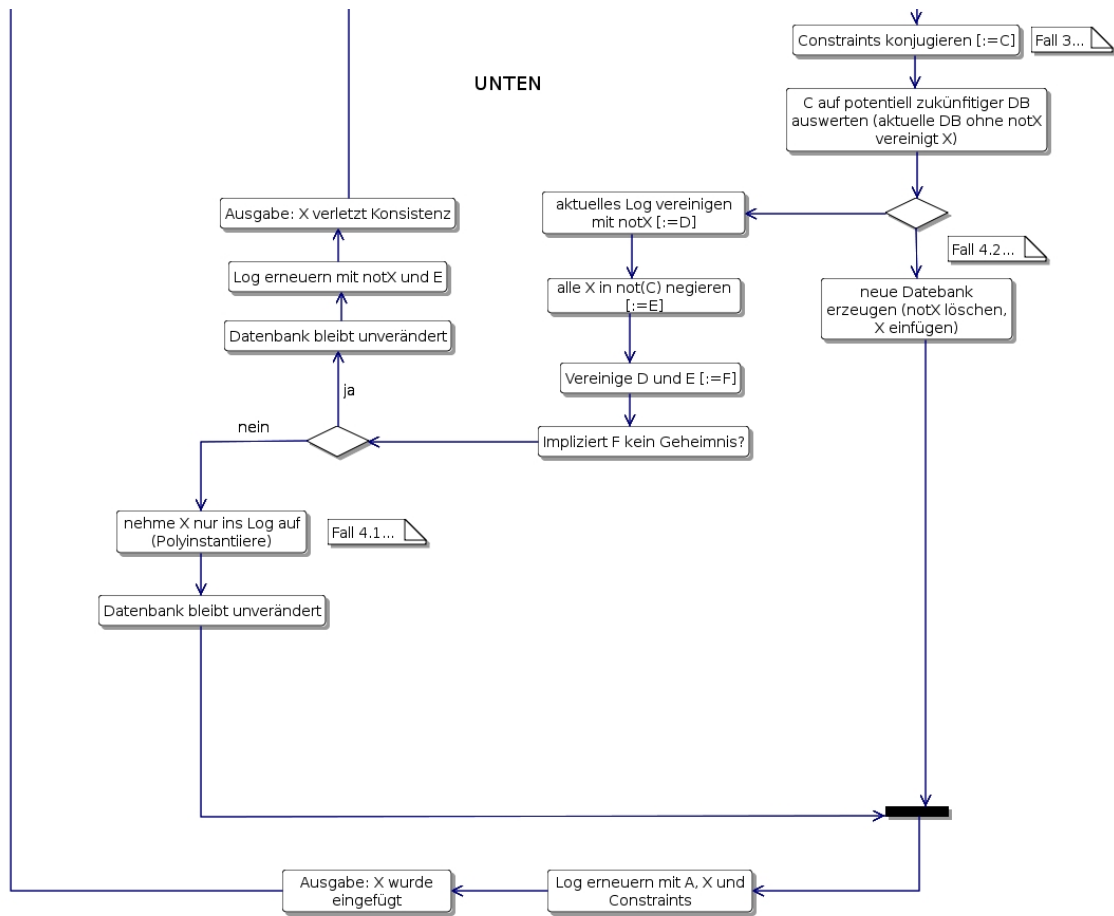


Abbildung 12.3: Aktivitätsdiagramm: unterer Teil

12.2.2 Sequenzdiagramm Sichtänderung - Fall2

In Abbildung 12.4 auf Seite 159 wird der 2. Fall des Algorithmus zur Sichtänderung von Seiler dargestellt, also die auch vom Benutzer selbst berechenbare Konsistenz- und Geheimnisüberprüfung. Nach erfolgreicher Anmeldung des „einfachen Benutzers" am Prototypen ($ChildQuery$) wird die Instanz ViewUpdate erzeugt und damit das Benutzerwissen ($getLog()$) und die potenziellen Gehimnisse ($getPot_sec()$) des aktuellen Benutzers ($UserData$) als verkettete Liste von $Formula$-Objekten ausgelesen. Weiterhin wird im Konstruktor direkt eine Instanz von $SQLInteraction$ erzeugt und die Constraints ausgelesen bzw. geparsed. Wenn der Benutzer eine Update-Operation ausführt, d.h. den viewupdate-Button drückt, wird der eingegebene String (Wert χ) einer Sprachanalyse unterzogen und zum Formula-Objekt geparsed ($cqe()$). Dieses Formula-Objekt entspricht unserem $value$, auf welchem der Update-Algorithmus durchgeführt wird ($update(value)$). Im Sequenzdiagramm wird nur Fall 2 behandelt, also alle anderen Fälle ausgeblendet. Dieser wird ausführlich auf Seite 167 beschrieben. Die konzeptuelle Darstellung [alt] beschreibt die Verwendung einer $if - Abfrage$. Falls die $Precondition$ für Fall 2 erfüllt sind, wird der [$true$]-Zweig abgearbeitet und der Algorithmus terminiert mit entsprechender Log-Aktualisierung und Benutzerbenachrichtigung. Anderenfalls ([$false$]) wird in den Fall 3 „gesprungen" (siehe modularer Aufbau 165), welcher im Diagramnm nicht mehr behandelt worden ist.

12.2.3 Kommentierung der Methode $ViewUpdate.update()$

Im Folgenden wird die Implementierung des Sicht-Änderungsalgorithmus vorgestellt. Der Algorithmus überprüft sequentiell in 4 Fällen, ob ein entsprechender Wert χ in der Datenbank geändert werden darf oder nicht. Hierzu benutzen wir die Variable $caseflag$ als „Sprungmarke", die entsprechend der vorliegenden Situation in den jeweiligen Fall und somit Methode „springt". Die Methoden informieren den Benutzer durch die GUI mittels der Methode $getAnswer()$ über den Ausgang seiner gestellten Änderungsoperation. Auf die theoretische Vorgehensweise soll hier nicht genauer eingegangen werden, da diese wiederum in der Seminararbeit von Schlotmann bzw. in der Diplomarbeit von Seiler behandelt wird. [Sch09, Sei08]
Im folgenden Programmcode wird zunächst ein Überblick über den modularen Auf-

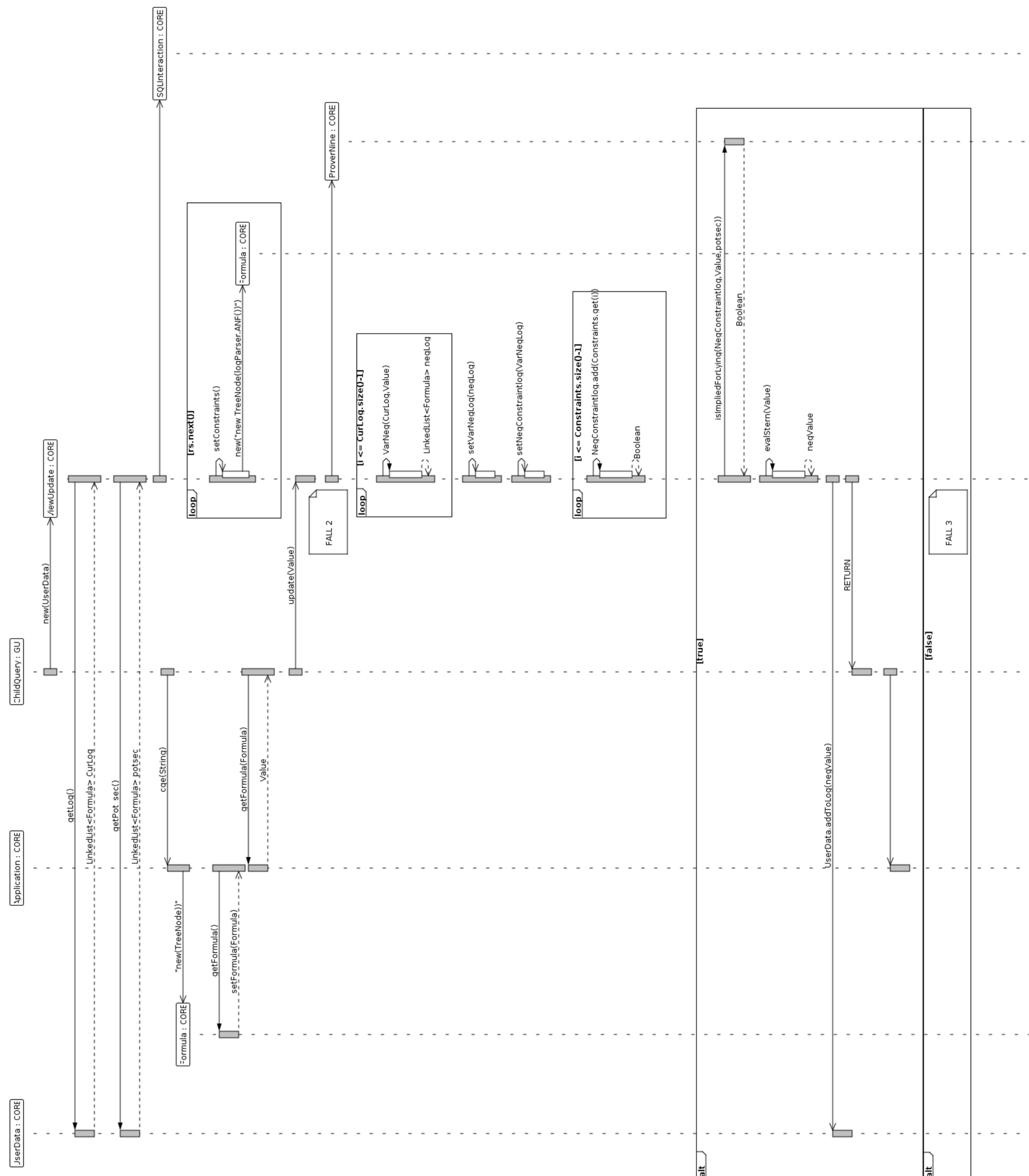


Abbildung 12.4: Sequenzdiagramm - Fall2

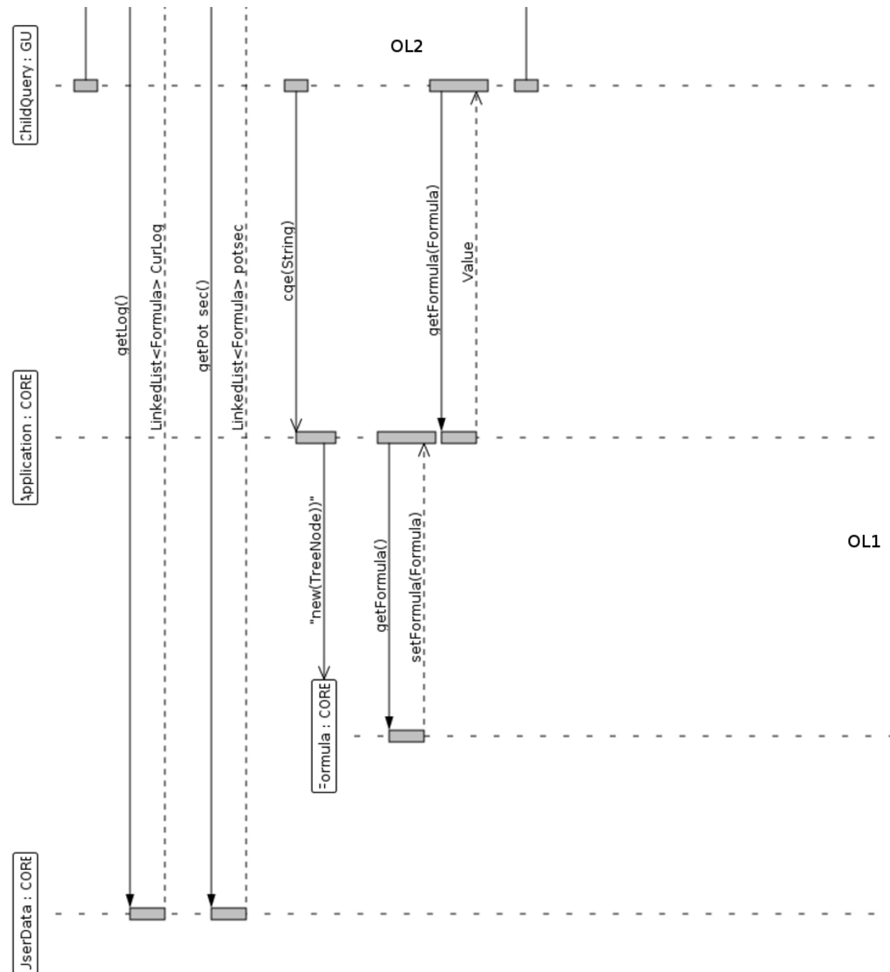


Abbildung 12.5: Sequenzdiagramm - Fall2: oben links (OL)

Abbildung 12.6: Sequenzdiagramm - Fall2: oben rechts (OR)

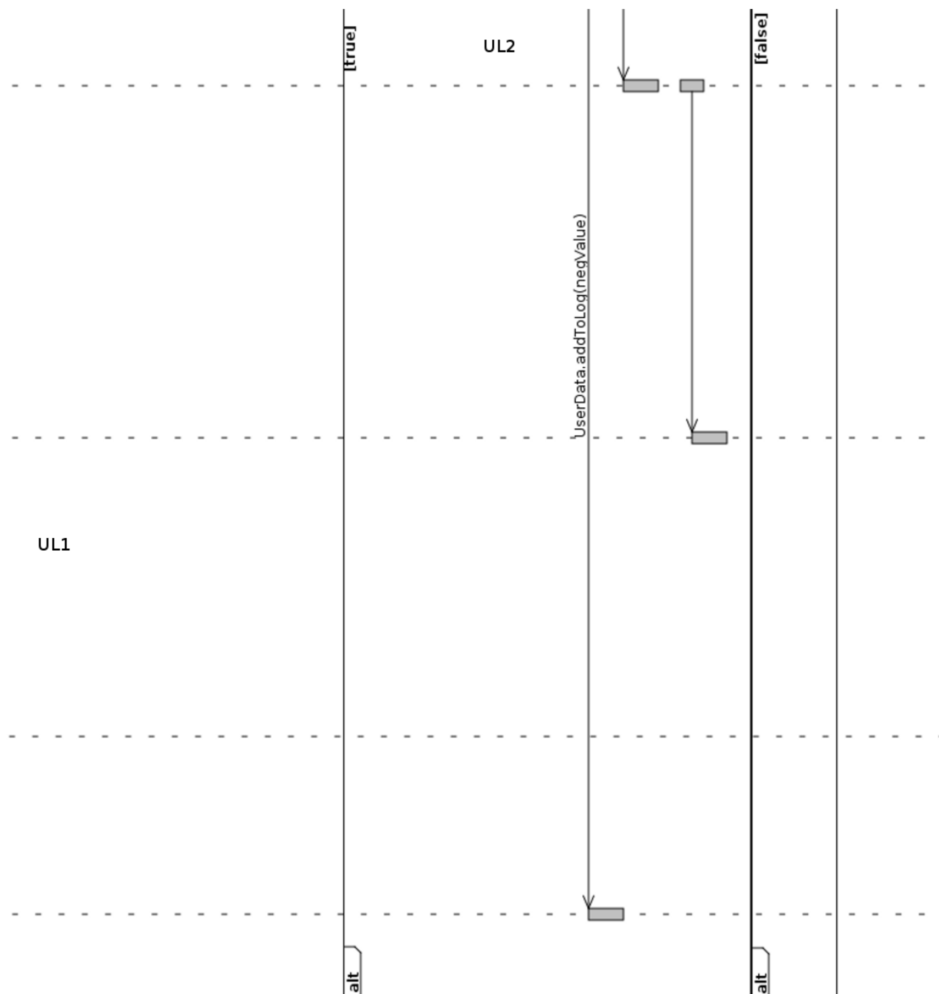


Abbildung 12.7: Sequenzdiagramm - Fall2: unten links (UL)

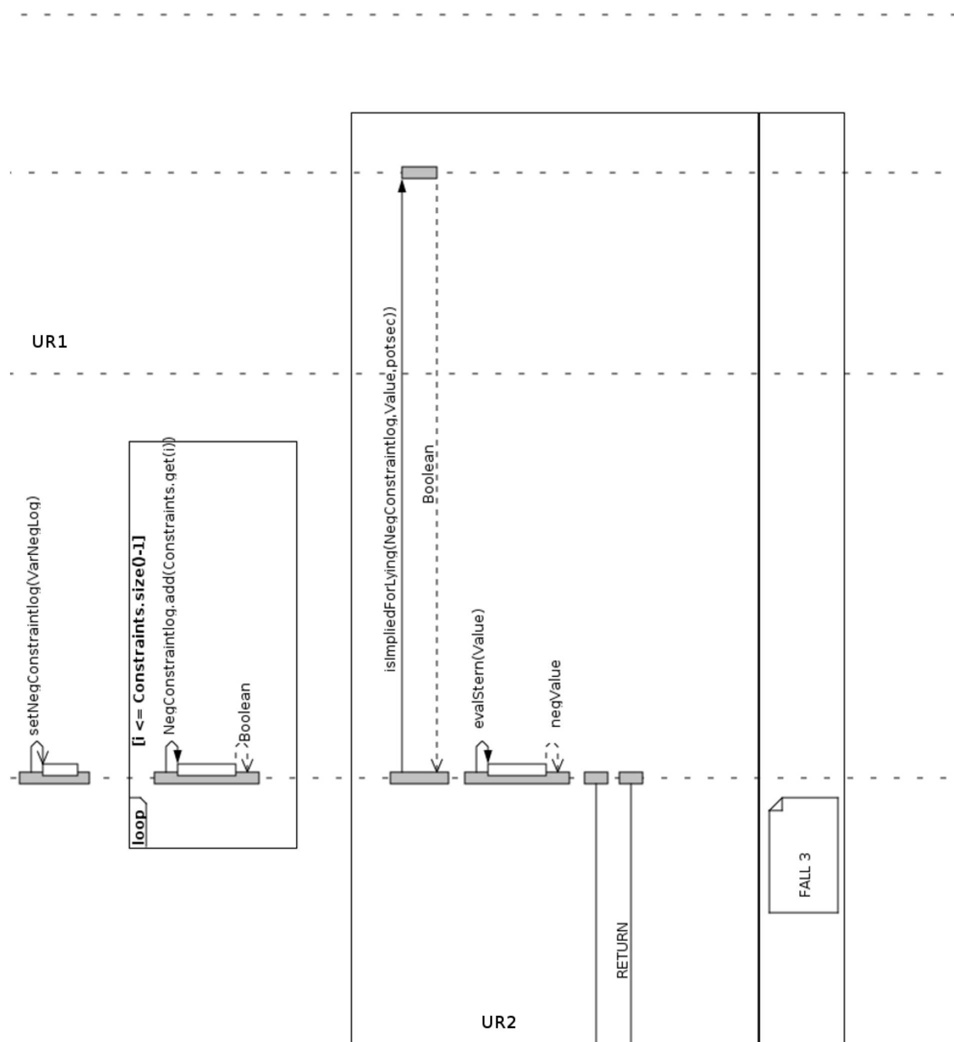


Abbildung 12.8: Sequenzdiagramm - Fall2: unten rechts (UR)

bau gegeben, wobei anschließend auf die einzelnen Methoden und ihr Vorgehen genauer eingegangen werden soll. Die Testergebnisse finden sich in Abschnitt 12.2.12 auf Seite 180.

```
1   public LinkedList<String> update(Formula value){
2     setValue(value);
3     setTable(value);
4     setKonstante(value);
5     this.setNOTvalueflag(this.getValue().toString().indexOf(this.
          getNOTvalue()));
6       try {
7       boolean caseflag = this.vuCase11();
8       if(caseflag==true){
9         return this.getAnswers();
10      }else if(this.getAnswer(0)=="FALL2"){
11        this.caseflag=true;
12      }
13      if(this.caseflag==false){
14
15        this.caseflag = vuCase12();
16        if(this.caseflag==true){
17          return this.getAnswers();
18        }
19      }
20      this.caseflag = vuCase2();
21      if(this.caseflag==true){
22        return this.getAnswers();
23      }
24      this.caseflag = vuCase31();
25      if(this.caseflag==true){
26        this.caseflag = vuCase32();
27        if(this.caseflag==true){
28         return this.getAnswers();
29        }
30        this.caseflag = vuCase41();
31      }else{
32        this.caseflag = vuCase42();
33      }
34      } catch (Exception e) {
35        System.out.println("Exception: "+e.getMessage());
36      }
```

```
37      }
38      return this.getAnswers();
39    }
```

12.2.4 Kommentierung der Methode $vucase11()$

Fall 1.1 spiegelt die Situation wieder, dass der zu ändernde Wert χ in der Datenbank bereits vorhanden ist und diese Information zusammen mit dem aktuellen Benutzerlog kein Geheimnis impliziert.
Ob der Wert tatsächlich schon enthalten ist, wird mittels der Methode *sqe*, welcher der zu ändernde Wert χ als Formel übergeben wird, überprüft. Diese Methode liefert eine einfache Anfrageauswertung. Falls diese positiv ausfällt, wird der Methode $isimpliedforLying$ der Klasse Prover9 das aktuelle Log, der Wert χ und das PotSec des aktuellen Benutzers übergeben. Die Methode hat die Aufgabe anhand dieser Werte zu kontrollieren, ob ein Geheimnis geschlussfolgert werden könnte. Für den Fall, dass diese Implikation nicht wahr ist, terminiert der Algorithmus bereits hier. Es wird dann über die Methode $addToLog$ das Log mit dem Wert χ aktualisiert (Codezeile 5) und eine Ausgabe für den Benutzer produziert. Sollte die Implikation allerdings wahr sein, so wird der Nutzer angelogen und der Algorithmus springt in Fall 2(12.2.6).

```
1  public boolean vuCase11(){
2      try {
3        if(sqe(this.getValue())){
4                          if(!prov.isImpliedForLying(this.getCurlog()
                               ,this.getValue(),this.getPotsec())){
5          this.getUserdata().addToLog(this.getValue());
6          this.getAnswers().add(1,"Wert bereits in der Datenbank
               enthalten");
7          this.getAnswers().add(0,"FALL1.1");
8          return true;
9        }
10       this.getAnswers().add(0,"FALL2");
11       return false;
12       }
13     } catch (Exception e) {
14       e.printStackTrace();
```

```
15      }
16      return false;
17    }
```

12.2.5 Kommentierung der Methode *vucase*12()

Fall 1.2 tritt ein, falls ein zu ändernder Wert χ zwar wirklich noch nicht in der Datenbank enthalten ist, diese Information aber wiederum dem aktuellen Benutzer nicht mitgeteilt werden darf, da sonst Inferenzen ermöglicht werden würden.
Dementsprechend springt der Algorithmus in den Fall 1.2, sofern die in 12.2.4 beschriebene Methode *sqe* false zurückliefert. In diesem Fall wird der Wert χ zunächst mit der Methode *evalstern* negiert, wobei *evalstern* die aktuelle Ausprägung des Wertes aus der Datenbank liest [1], und anschließend die Methode *isimpliedforLying* aufgerufen. Diese bekommt den negierten Wert, das aktuelle Log und das PotSec übergeben und hat erneut die Aufgabe zu prüfen, ob durch das Log und die Information bezüglich des negierten Wertes χ die Disjunktion der Geheimnisse impliziert werden würde. Falls diese Implikation hier wahr ist, wird der Benutzer angelogen und die Logs wie in Fall 1.1 aktualisiert. Dem Benutzer wird also mit einer entsprechenden Antwort mitgeteilt, dass der Wert schon in der Datenbank enthalten ist, wodurch der Algorithmus terminiert.

```
1  public boolean vuCase12(){
2      try{
3      Formula negvalue = this.evalStern(this.getValue());
4      if(prov.isImpliedForLying(this.getCurlog(),negvalue,this.
           getPotsec())){
5              this.getUserdata().addToLog(this.getValue());
6        this.getAnswers().add(0,"FALL1.2");
7        this.getAnswers().add(1,"Wert bereits in der Datenbank
             enthalten");
8        return true;
9        }
10     }catch(Exception e){
11     e.getStackTrace();
```

[1] im Gegensatz zur Methode *negate*, welche unabhängig von dem Vorkommen des Wertes in der DB negiert.

```
12      }
13      return false;
14    }
```

12.2.6 Kommentierung der Methode *vucase2()*

Fall 2 wird also im Programmablauf einerseits erreicht, wenn ein Wert χ schon enthalten ist und diese Information ein Geheimnis impliziert und somit nicht dem Benutzer mitgeteilt werden darf, und andererseits, wenn der Wert noch nicht enthalten ist und diese Information kein Geheimnis impliziert.
Liegt eine dieser beiden Situationen vor, so wird nun die vom Benutzer selbst berechenbare Konsistenz- bzw. Geheimnisverletzung ausgewertet. Dazu wird zunächst die Methode *VarNeg* (Abschnitt 12.2.10) mit dem aktuellen Benutzerlog und dem Wert χ aufgerufen, wodurch die Variablennegation (9.2.1 im Log realisiert wird. Das Ergebnis wird im Attribut *VarNegLog* abgespeichert und *VarNegLog* nach *NegConstraintLog* kopiert. Diese verkettete Liste vom Typ Formula wird anschließend mit Hilfe einer for-Schleife um die Konsistenzbedingungen *Constraints* der Datenbank erweitert, um diese wiederum zusammen mit dem Wert χ und dem PotSec der Methode *isimpliedforLying* zu übergeben. Bei positiver Auswertung terminiert der Algorithmus mit einer entsprechenden Meldung an den Benutzer und das Log des Benutzers wird wie oben mit *addToLog* aktualisiert. Da aus Sicht des Benutzers $\neg\chi$ galt, muss dieser Wert auch mit ins Log aufgenommen werden. Wir erhalten diesen durch den Aufruf der Methode *negate*.

```
1  public boolean vuCase2(){
2      try{
3      this.setVarNegLog(this.VarNeg(this.getCurlog(),this.getValue())
           );
4      this.setNegConstraintlog(this.getVarNegLog());
5      for(int i=0;i<=this.getConstraints().size()-1;i++){
6        boolean flag = this.getNegConstraintlog().add(this.
             getConstraints().get(i));
7                                                                    }
8      if(prov.isImpliedForLying(this.getNegConstraintlog(),this.
           getValue(),this.getPotsec())){
9        this.getValue().negate();
```

```
10        this.getUserdata().addToLog(this.getValue());
11        this.getValue().negate();
12        this.getAnswers().add(0,"FALL2");
13        this.getAnswers().add(1,"Konsistenzverletzung");
14        return true;
15      }
16      }catch(Exception e){
17      e.getStackTrace();
18      }
19      return false;
20    }
```

12.2.7 Kommentierung der Methode $vucase31()$

In Fall 3 wird die Konjunktion der Konsistenzbedingungen auf der potentiell zukünftigen Datenbank ausgewertet.
Für die Update-Operation $update(\chi)$ wird der Wert in die entsprechende Tabelle eingefügt, während für $update(\neg\chi)$ der Wert aus der Tabelle entfernt wird. Diese Fallunterscheidung wird durch die erste if-Abfrage und die vorübergehende Datenbankänderung durch $sqlInsert$ bzw. $sqlDelete$ realisiert. Anschließend wird mit Hilfe der for-Schleife jede in der Datenbank eingetragene Konsistenzbedingung ausgelesen und mittels *sqe* in der konstruierten Datenbank auf Gültigkeit hin überpüft. Sobald die erste Bedingung gefunden ist, welche in dieser Datenbank nicht mehr gelten würde, verlassen wir die if-Schleife und springen in Fall 3.2. Falls wir eine solche Bedingung jedoch nicht finden sollten, die Konsistenz also auch nach der Änderung des Wertes von χ noch gegeben ist, springt der Alogrithmus direkt in Fall 4.2(12.2.10. Unabhängig von der Auswertung der for-Schleife, wird natürlich nach dieser der urspüngliche Datenbankzustand durch die erneute Anwendung der Methoden $sqlInsert$ bzw. $sqlDelete$ wieder hergestellt (Codezeilen 17-20).

```
1 public boolean vuCase31(){
2     try{
3     SQLInteraction sqlinteract = new SQLInteraction();
4     if(this.getNOTvalueflag()==0){
5       sqlinteract.sqlDelete(this.getKonstante(), this.getTable());
6     }else{
7     sqlinteract.sqlInsert(this.getKonstante(), this.getTable());
```

```
8      }
9      Boolean constraint_flag = false;
10     for(int i=0;i<=getConstraints().size()-1;i++){
11       Boolean flag = this.sqe(this.getConstraints().get(i));
12       if(!flag){
13         constraint_flag = true;
14         break;
15               }
16       }
17     if(this.getNOTvalueflag()==0){
18       sqlinteract.sqlInsert(this.getKonstante(), this.getTable());
19     }else{
20     sqlinteract.sqlDelete(this.getKonstante(), this.getTable());
21                 if(constraint_flag){
22     return true;
23   }
24     }catch(Exception e){
25     e.getStackTrace();
26   }
27     return false;
28   }
29 }
```

12.2.8 Kommentierung der Methode $vucase32()$

Falls in Fall 3.1 eine Konsistenzbedingung auf der potentiell zukünftigen Datenbank ausgewertet wurde, welche auf dieser nicht mehr gültig wäre, so wird in Fall 3.2 nun gepüft, ob diese Konsistenzverletzung an den Benutzer gemeldet werden darf.
Dieses wird durch einen erneuten Aufruf der Methode $isimpliedforLying$ realisiert. Als erstes wird die Methode $NegConstraint$ aufgerufen, welche die Negation der in der Datenbank gültigen Konsistenzbedingungen $Constraints$ implementiert. Danach wird die Variablennegation $VarNeg$ mit dem Wert χ auf dieser Menge aufgerufen, welche bereits in Fall 2 schon benutzt wurde. Hierbei wird die geschickte Verwendung der Referenzen deutlich. Das heißt, dass die Methode $getConstraints$ in Wahrheit eine verkettete Liste von negierten Constraints zurückliefert. Mit $conditionconstraint$ wird festgehalten, dass unter Änderung des Wertes von χ eine Konsistenzverletzung auftreten würde. In diese verkettete Liste vom Typ Formula werden dann in der ersten for-Schleife die Menge der Constraints hinzu-

gefügt, während in der zweiten Schleife die Menge *conditionconstraint* mit dem aktuellen Log vereinigt wird. Die verkettete Liste *conditionconstraint*, der negierte Wert und das PotSec werden nun *isimpliedforLying* übergeben, um die Wahrheit der Implikation zu prüfen. Falls diese falsch ist, terminiert der Algorithmus mit einer entsprechenden Meldung an den Benutzer und die Logs werden aktualisiert. Mittels *addToLog* wird der negierte Wert, welchen wir über *evalstern* bekommen, und *conditionconstraint* mit dem aktuellen Log vereinigt.

```
1  public boolean vuCase32(){
2      try{
3      this.NegConstraint(this.getConstraints());
4      this.VarNeg(this.getConstraints(), this.getValue());
5      Formula negvalue = this.evalStern(this.getValue());
6      LinkedList<Formula> conditionconstraint = new LinkedList<
           Formula>();
7      for(int i=0;i<=this.getConstraints().size()-1;i++){
8        Boolean flag = conditionconstraint.add(this.getConstraints().
             get(i));
9        if(!flag){
10         throw new Exception("Fehler beim hinzufuegen der
               VarNeg_NegConstraint in Fall 3.2 in ViewUpdate");
11       }
12     }
13     for(int i=0;i<=this.getCurlog().size()-1;i++){
14       Boolean flag = conditionconstraint.add(this.getCurlog().get(i
             ));
15       if(!flag){
16         System.out.println("Fehler beim hinzufuegen der Log-
               eintraege in Fall 3.2 in ViewUpdate");
17         }
18     }
19     if(!prov.isImpliedForLying(conditionconstraint,negvalue,this.
           getPotsec())){
20       this.getUserdata().addToLog(negvalue);
21       this.setConstraints();
22       for(int i=0;i<=this.getConstraints().size()-1;i++){
23       this.getUserdata().addToLog(this.getConstraints().get(i));
24       this.getAnswers().add(0,"FALL3");
25       this.getAnswers().add(1,"Konsistenzverletzung");
26       return true;
```

```
27       }
28     }catch(Exception e){
29     e.getStackTrace();
30     }
31     return false;
32 }
```

12.2.9 Kommentierung der Methode *vucase*41()

Wird in Fall 3.1 die Konsistenz verletzt und ist in Fall 3.2 die Implikation wahr, so darf diese Verletzung dem Benutzer auf keinen Fall mitgeteilt werden.
In einer solchen Situation wird der Benutzer angelogen, was durch Polyinstantiierung erreicht wird. Dem Benutzer wird also eine konsistente Datenbank vorgelogen, obwohl seine Änderung tatsächlich eine Inkonsistenz dieser zur Folge hätte. Entsprechend wird das Benutzerlog mit dem Wert χ aktualisiert, die Datenbank bleibt jedoch unverändert. Zusätzlich werden noch das variablennegierte Log und die Konsistenzbedingungen *Constraints* mit *addToLog* dem alten Log hinzugefügt.

```
1  public boolean vuCase41(){
2      try{
3      this.getUserdata().addToLog(this.getValue());
4  
5      for(int i=0;i<=this.getVarNegLog().size()-1;i++){
6        this.getUserdata().addToLog(this.getVarNegLog().get(i));
7        }
8      for(int i=0;i<=this.getConstraints().size()-1;i++){
9        this.getUserdata().addToLog(this.getConstraints().get(i));
10       }
11     this.getAnswers().add(0,"FALL4.1");
12     this.getAnswers().add(1,"Wert wurde eingefuegt");
13     }catch(Exception e){
14     e.getStackTrace();
15     }
16     return true;
17 }
```

12.2.10 Kommentierung der Methode *vucase*42()

Erfolgt keine Abweisung der Änderungs-Operation in den bisher genannten Fällen, so wird die Änderung durchgeführt.
Die Ausgabe sieht hierbei genau so aus, wie bereits in Fall 4.1 beschrieben. Die Log-Aktualisierung wird dabei für alle Benutzer, die mit dem Lügenzensor arbeiten gemäß 13.2.4 durchgeführt. Da die Konsistenz im Gegensatz zu Fall 4.1 nicht verletzt ist, wird der entsprechende Wert über die Klasse *SQLInteraction* (12.2.13) in die Datenbank eingefügt. Zum Schluss wird in der if-Bedingung abgefragt, ob durch dieses Update ein *Insert* oder ein *Delete* auf Datenbankebene durchgeführt werden muss, also ob eine *update*(χ) oder eine *update*($\neg\chi$) Operation vom Benutzer veranlaßt wurde.

```
1  public boolean vuCase42(){
2      try{
3              this.setVarNegLog(this.VarNeg(this.getCurlog(),this.
                   getValue()));
4      this.getUserdata().addToLog(this.getValue());
5      this.updateUserLogs();
6      for(int i=0;i<=this.getVarNegLog().size()-1;i++){
7      this.getUserdata().addToLog(this.getVarNegLog().get(i));
8        }
9            for(int i=0;i<=this.getConstraints().size()-1;i++){
10         this.getUserdata().addToLog(this.getConstraints().get(i));
11       }
12       if(this.getNOTvalueflag()==0){
13         sqlinteract.sqlDelete(this.getKonstante(), this.getTable())
             ;
14       }else{
15       sqlinteract.sqlInsert(this.getKonstante(), this.getTable());
16       }
17       this.getAnswers().add(0,"FALL4.2");
18       this.getAnswers().add(1,"Wert wurde eingefuegt");
19           }catch(Exception e){
20           e.getStackTrace();
21           }
22 return true;
23 }
```

Im Folgenden soll ein Überblick über die wichtigsten Methoden und ihre Funktionalität gegeben werden.

Variablennegation in ViewUpdate.java Die Methode implementiert die Variablennegation des Logs (bzw. generell auf einer Formelmenge), welche zur Zeit nur für Literale funktioniert. Zu beachten ist hierbei, dass diese Variablennegation auf der positiven Ausprägung ausgeführt wird. Einzelheiten hierzu finden sich in Seminarausarbeitung bzw. der Diplomarbeit wieder.

```java
public LinkedList<Formula> VarNeg(LinkedList<Formula> linklist,
    Formula value) throws Exception{
  if(this.getNOTvalueflag()==0){
    this.value.negate();
  }
  for(int i=0;i<=linklist.size()-1;i++){
    if(value.equals(linklist.get(i))){
      linklist.get(i).negate();
      this.sqlinteract.sqlDelete(this.value.toString(), this.
          userdata.getUserName()+"Log");
    }
    if(!linklist.get(i).isGroundatom()){
      throw new Exception();
    }
  }
  if(this.getNOTvalueflag()==0){
    this.value.negate();
  }
  this.value.negate();
  for(int i=0;i<=linklist.size()-1;i++){
    if(value.equals(linklist.get(i))){
      linklist.get(i).negate();
      this.sqlinteract.sqlDelete(this.value.toString(), this.
          userdata.getUserName()+"Log");
    }
  }
  this.value.negate();
  return linklist;
}
```

Die Methode erwartet eine verkettete Liste aus Formel-Objekten und eine Formel als Parameter. In der if-Abfrage wird überprüft, ob ein negatives Literal $\neg\chi$ übergeben wird. Falls dies der Fall ist, wird der Wert negiert. In der ersten Schleife werden also die Log-Einträge jeweils mit χ verglichen und bei Übereinstimmung negiert. Der ursprüngliche Eintrag wird dementsprechend mit der Methode *sqlDelete* aus dem Log gelöscht. Da wir wiederum nur auf Referenzen arbeiten, muss, sofern der Wert $\neg\chi$ übergeben und somit vor der ersten Schleife (Codezeile 3) negiert wurde, zurücknegiert werden. Vor der zweiten for-Schleife (Codezeile 17) negieren wir unseren Wert, wodurch nun die Log-Einträge mit $\neg\chi$ verglichen werden. Bei Übereinstimmung wird erneut negiert und der ursprüngliche Eintrag im Log entfernt. Anschließend wird der Wert, wie auch schon oben, wieder zurück negiert. Die wiederholte Verwendung von *negate* ist unumgänglich, da wir mit Referenzen arbeiten und ein DeepCopy[2] den gesamten Syntaxbaum quasi erneut implementieren müsste.

12.2.11 Behandlung von Constraints

Die Constraints müssen wie erwähnt zu Beginn vom Administrator definiert werden. Die Constraints werden in der Datenbank in der gleichnamigen Tabelle gehalten. Weiterhin dürfen die Constraints nicht geändert werden und müssen als atomare Formeln eingetragen werden.

Constraints in ViewUpdate.java Die Methode liest die Constraints aus der Datenbanktabelle *Constraints* aus und fügt diese als Formel-Objekte einer verketteten Liste hinzu. Vorweg muss jeder String aus der ResultSet (Cursor-Konzept 7.1) geparsed werden und ein Syntaxbaum erstellt werden. Anhand dieses Syntaxbaumes wird dann die Formel erzeugt.

```
1  public void setConstraints() throws Exception {
2
3    ResultSet rs = null;
4    Statement stmt = null;
5    try {
6      Connection con = DatabaseConf.getDBConnection();
7      stmt = con.createStatement();
8      rs = stmt.executeQuery("SELECT * FROM schema.Constraints");
```

[2]Klonen von Objekten: siehe http://en.wikipedia.org/wiki/Object_copy

```
9        while (rs.next()) {
10         String s = rs.getString(1) + ";";
11         System.out.println("setConstr String vorm Parsen: " + s);
12         logParser.ReInit(new StringReader(s));
13         Formula f;
14         f = new Formula(new TreeNode(logParser.ANF()));
15         System.out.println("neue Formel f: " + f.toString());
16 
17         if (!f.isGroundatom())
18           return;
19 
20         this.constraints.add(f);
21 
22       }
23     } catch (Exception e) {
24       System.out.println(e.getMessage());
25     } finally {
26       rs.close();
27       stmt.close();
28     }
29 
30   }
```

Auslesen der Relation und der Konstante in ViewUpdate.java Um individuelle SQL-Anfragen stellen zu können, müssen die Namen der Relation und der Konstante bekannt sein. Diese Methoden liefern beides anhand des Syntaxbaumes der Formel. Der Syntaxbaum wird dabei rekursiv durchlaufen. In dem alten Prototypen existiert nur eine spezielle Methode, um eine Formel in eine spezielle Select-Anfrage zu konvertieren. Wenn man einen anderen SQL-Befehl erstellen möchte, kennt man die benötigten Tabellennamen nicht. Daher sind folgende Methoden notwendig.

```
1 public void setTable(Formula value) {
2     try {
3       String table = value.Groundatom_getRelationName(value
4           .getSyntaxTree());
5       this.table = table;
6       System.out.println("setTable: " + table);
7     } catch (Exception e) {
8       System.out.println(e.getMessage());
```

```
9      }
10   }
11 public void setKonstante(Formula value) {
12     try {
13       this.konstante = value.Groundatom_getKonstante(value
14           .getSyntaxTree());
15       System.out.println("setKonstante: " + getKonstante());
16     } catch (Exception e) {
17       System.out.println(e.getMessage());
18     }
19   }
20 }
21 public String Groundatom_getRelationName(TreeNode n) throws
     Exception {
22     String typ = n.toString();
23     String relname = "";
24     if (typ == "ANF") {
25         return (""+Groundatom_getRelationName(n.getChild(n.
             getNumChildren() - 1)));
26     }
27     else if (typ == "EXP") {
28       switch (n.getNumChildren()) {
29       case 1: {
30         TreeNode r = n.getChild(0);
31         relname = (r.getChild(0)).toString();
32         System.out.println("Tabellenname: "+relname);
33         break;
34       }
35       case 2: {
36         TreeNode r = n.getChild(1);
37         return (""+Groundatom_getRelationName(r));
38                                 }
39       }
40     }
41     return relname;
42   }
43 public String Groundatom_getKonstante(TreeNode n) throws Exception
     {
44     String typ = n.toString();
45     String konstante = "";
46     if (typ == "ANF") {
```

```
47     return (""+Groundatom_getKonstante(n.getChild(n.getNumChildren
           () - 1)));
48     }
49     else if (typ == "EXP") {
50       switch (n.getNumChildren()) {
51       case 1: {
52         TreeNode r = n.getChild(0);
53         return (""+Groundatom_getKonstante(r));
54       }
55       case 2: {
56         TreeNode r = n.getChild(1);
57         return (""+Groundatom_getKonstante(r));
58       }
59       }
60     }
61     if (typ == "R") {
62       TreeNode r = n.getChild(1);
63       konstante = (""+r.getChild(0).toString().substring(7));
64       System.out.println("Konstante: "+konstante);
65
66     }
67     return konstante;
68   }
69 }
```

Parsen der Constraints in UserData.java Da die Constraints als Strings in der Tabelle Constraints gehalten werden, müssen diese erst zu Formula-Objekten umgewandelt werden, um später die Informationen wie Relationenname und Konstante erhalten zu können.

```
1 public void constraintsToFormula() throws Exception {
2     ResultSet rs = null;
3     Statement stmt = null;
4     try {
5       Connection con = DatabaseConf.getDBConnection();
6       stmt = con.createStatement();
7       rs = stmt.executeQuery("SELECT * FROM Constraints");
8       while (rs.next()) {
9         String s = rs.getString(1) + ";";
10        logParser.ReInit(new StringReader(s));
11        Formula f;
```

```
12          f = new Formula(new TreeNode(logParser.ANF()));
13          if (!f.isGroundatom()){
14            System.out.println("Constraint kein Grundatom");
15          }else{
16          this.constraintsAsFormula.add(f);
17          }
18          }
19      } catch (Exception e) {
20        System.out.println(e.getMessage());
21      } finally {
22        rs.close();
23        stmt.close();
24      }
25    }
26  }
```

Kontrolle, ob die Constraints in der Datenbank gelten in UserData.java
Nachdem die Constraints mittels der Methode *constraintsToFormula* ausgelesen und zur Formel geparsed worden sind, wird für jede Formel der Syntaxbaum durchlaufen und der Relationenname und die Konstante ausgelesen. Anhand dieser Informationen wird die Existenz der Konstante in der Relation überprüft. Falls die Konstante nicht in der Datenbank enthalten ist, wird die Konsistenz gefordert und die Konstante der Relation (Datenbanktabelle) hinzugefügt oder umgekehrt.

```
1  public void checkConstraints(){
2    try {
3        for(int i=0;i<=this.constraintsAsFormula.size()-1;i++){
4        String table = this.constraintsAsFormula.get(i).
             Groundatom_getRelationName(this.constraintsAsFormula.get(
             i).getSyntaxTree());
5        String konstante = this.constraintsAsFormula.get(i).
             Groundatom_getKonstante(this.constraintsAsFormula.get(i).
             getSyntaxTree());
6              if(!this.constraintsAsFormula.isEmpty()){
7        if(!(this.sqlinteract.sqlExistsInTable(konstante, table))){
8          this.sqlinteract.sqlInsert(konstante, table);
9        }
10       }
11     }
12   } catch (Exception e) {
```

```
13      e.printStackTrace();
14    }
15  }
```

Constraints dem initialen Log des aktuellen Benutzers hinzufügen in UserData.java Diese Methode überprüft, ob der Benutzer den Lügenzensor ausgewählt hat. Erst dann werden die Constraints, sofern sie nicht schon im Log enthalten stehen, beim ersten Mal (d.h. ins log_0) eingetragen. Zu dieser Überprüfung dient die Spalte *ConstraintFlag* in der Benutzertabelle der Datenbank. Der Wert ist zu Beginn false, nach aktualisiertem initialen log_0 true.

```
1  public void addConstraintsToLog(){
2    char luegencensor = '1';
3    try {
4    if(this.censor==luegencensor){
5            LinkedList<String> constrflag = this.sqlinteract.
                 getSingleColumnContent("benutzertabelle", "
                 constraintflag", "login",this.getUserName() );
6      if(constrflag.get(0).equals("false")){
7      LinkedList<String> constr = this.sqlinteract.
           getTableContentAsStringSet("Constraints");
8      for(int i=0;i<=constr.size()-1;i++){
9      if(!(this.sqlinteract.sqlExistsInTable(constr.get(i), this.
           getUserName()+"Log")))
10     this.sqlinteract.sqlInsert(constr.get(i), this.getUserName()+"
           Log");
11         }
12       }
13       this.sqlinteract.sqlUpdate("benutzertabelle", "constraintflag
             ","true","login",this.getUserName());
14     }
15   }
16 } catch (Exception e) {
17   e.printStackTrace();
18   }
19 }
```

12.2.12 Tests der Methoden

Im Folgenden werden die Testfälle mit Ergebnissen aufgeführt, welche wir an unserer Implementierung des Sichtänderungsalgorithmus durchgeführt haben. Die Tests sind nach Fällen des Algorithmus unterteilt, wie in den Abschnitten 11.3.4 bis 11.3.10 geschildert. Zunächst werden noch einmal die Bedingungen des jeweiligen Falls aufgeführt, welche die Terminierung zur Folge haben und anschließend die Reaktionen auf eine solche Terminierung. Zu den Testfällen selbst ist einerseits die Ausgangssituation bezüglich der Datenbank und Benutzerwissen bzw. Geheimnissen dargestellt und andererseits das Ergebnis und entsprechende Änderungen an den Benutzerlogs bzw. der Datenbank selbst. Dabei sind nur die für uns interessanten und relevanten Einträge in der Datenbank aufgeführt. Ein Eintrag *not beinbruch*$(nils)$ bedeutet hier, dass kein Eintrag für $Nils$ in der Tabelle *beinbruch* existiert. In allen hier von uns aufgeführten und getesteten Fällen, stimmen gewünschtes und tatsächliches Ergebnis überein.

Test - Fall 1.1

Wert enthalten und neues Log impliziert KEIN Geheimnis

$$eval^*(\chi_i)(db_{i-1}) = \chi_i \textit{ And } log_{i-1} \cup \{\chi_i\} \not\models pot_sec_disj$$

$$db_i := db_{i-1}$$
$$log_i := log_{i-1} \cup \{\chi_i\}$$
$$ans_i := \chi_i \text{ bereits in der Datenbank enthalten.}$$

a)
PotSec: $\{beinbruch(torsten)\}$
Log: $\{armbruch(torsten)\}$
DB: $\{beinbruch(nils), armbruch(torsten)\}$
Constraints: $\{armbruch(torsten)\}$
Update-Anfrage: $beinbruch(nils)$;

<u>gewünschtes Ergebnis:</u>
Log: $\{armbruch(torsten), beinbruch(nils)\}$

Ausgabe: Wert bereits in der Datenbank enthalten.

tatächliches Ergebnis:
Log: $\{armbruch(torsten), beinbruch(nils)\}$
Ausgabe: Wert bereits in der Datenbank enthalten.

b)
PotSec: $\{beinbruch(torsten)\}$
Log: $\{armbruch(torsten)\}$
DB: $\{armbruch(torsten)\}$
Constraints: $\{armbruch(torsten)\}$
Update-Anfrage: $(not\ beinbruch(nils))$;

gewünschtes Ergebnis:
Log: $\{(armbruch(torsten), not\ beinbruch(nils))\}$
Ausgabe: Wert bereits in der Datenbank enthalten.

tatächliches Ergebnis:
Log: $\{(armbruch(torsten), not\ beinbruch(nils))\}$
Ausgabe: Wert bereits in der Datenbank enthalten.

Test - Fall 1.2

Wert NICHT enthalten und neues Log impliziert EIN Geheimnis

$$eval^*(\chi_i)(db_{i-1}) = \neg\chi_i\ And\ log_{i-1} \cup \{\neg\chi_i\} \models pot_sec_disj$$

$$db_i := db_{i-1}$$
$$log_i := log_{i-1} \cup \{\chi_i\}$$
$$ans_i := \chi_i \text{ bereits in der Datenbank enthalten.}$$

a)
PotSec: $\{(not\ beinbruch(nils))\}$

Log: $\{armbruch(torsten)\}$
DB: $\{armbruch(torsten)\}$
Constraints: $\{armbruch(torsten)\}$
Update-Anfrage: $beinbruch(nils)$;

gewünschtes Ergebnis:
Log: $\{armbruch(torsten), beinbruch(nils)\}$
Ausgabe: Wert bereits in der Datenbank enthalten.

tatächliches Ergebnis:
Log: $\{armbruch(torsten), beinbruch(nils)\}$
Ausgabe: Wert bereits in der Datenbank enthalten.

b)
PotSec: $\{beinbruch(nils)\}$
Log: $\{armbruch(torsten)\}$
DB: $\{beinbruch(nils), armbruch(torsten)\}$
Constraints: $\{armbruch(torsten)\}$
Update-Anfrage: $(not\ beinbruch(nils))$;

gewünschtes Ergebnis:
Log: $\{(armbruch(torsten), not\ beinbruch(nils))\}$
Ausgabe: Wert bereits in der Datenbank enthalten.

tatächliches Ergebnis:
Log: $\{(armbruch(torsten), not\ beinbruch(nils))\}$
Ausgabe: Wert bereits in der Datenbank enthalten.

Test - Fall 2

Wert enthalten und impliziert Geheimnis ODER Wert nicht enthalten und impliziert kein Geheimnis

$neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints \models pot_sec_disj$

$$db_i := db_{i-1}$$
$$log_i := log_{i-1} \cup \{\neg\chi_i\}$$
$$ans_i := \chi_i \text{ verletzt Konsistenz.}$$

a)
PotSec: {}
Log: $\{beinbruch(nils), (not\ armbruch(nils))\}$
DB: $\{beinbruch(nils)\}$
Constraints: $\{(not\ armbruch(nils))\}$
Update-Anfrage: $armbruch(nils)$;

gewünschtes Ergebnis:
Log: $\{beinbruch(nils), (not\ armbruch(nils))\}$
Ausgabe: Wert verletzt Konsistenz bzw. Geheimnis.

tatächliches Ergebnis:
Log: $\{beinbruch(nils), (not\ armbruch(nils))\}$
Ausgabe: Wert verletzt Konsistenz bzw. Geheimnis.

b)
PotSec: $\{beinbruch(nils)\}$
Log: $\{armbruch(torsten)\}$
DB: $\{armbruch(torsten), beinbruch(nils)\}$
Constraints: $\{armbruch(torsten)\}$
Update-Anfrage: $beinbruch(nils)$;

gewünschtes Ergebnis:
Log: $\{armbruch(torsten), (not\ beinbruch(nils))\}$
Ausgabe: Wert verletzt Konsistenz bzw. Geheimnis.

tatächliches Ergebnis:
Log: $\{armbruch(torsten), (not\ beinbruch(nils))\}$
Ausgabe: Wert verletzt Konsistenz bzw. Geheimnis.

c)
PotSec: $\{(not\ beinbruch(nils))\}$
Log: $\{armbruch(torsten)\}$
DB: $\{armbruch(torsten)\}$
Constraints: $\{armbruch(torsten)\}$
Update-Anfrage: $(not\ beinbruch(nils))$;

gewünschtes Ergebnis:
Log: $\{armbruch(torsten), beinbruch(nils)\}$
Ausgabe: Wert verletzt Konsistenz bzw. Geheimnis.

tatächliches Ergebnis:
Log: $\{armbruch(torsten), beinbruch(nils)\}$
Ausgabe: Wert verletzt Konsistenz bzw. Geheimnis.

Test - Fall 3

vom Benutzer nicht berechenbare Konsistenzverletzung

$$eval(con_conj)\ ((db_{i-1} \setminus \{\neg\chi_i) \cup \{\chi_i\}) = False$$
$$And\ log_{i-1} \cup \{\neg\chi_i\} \cup \{neg(\neg con_conj, \chi_i)\} \not\models pot_sec_disj$$

$db_i := db_{i-1}$
$log_i := log_{i-1} \cup \{\neg\chi_i\} \cup \{neg(\neg con_conj, \chi_i)\}$
$ans_i := \chi_i$ verletzt Konsistenz.

Fall tritt bei atomaren Konsistenzbedingungen nicht auf.

Test - Fall 4.1

Konsistenz in Fall 3.1 verletzt

$$\text{Polyinstantiiere } (= \text{nehme } \chi_i \text{ nur in } log_i \text{ auf})$$
$$db_i := db_{i-1}$$
$$log_i := neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints$$
$$ans_i := \chi_i \text{ wurde eingefügt.}$$

Fall tritt bei atomaren Konsistenzbedingungen nicht auf.

Test - Fall 4.2

Konsistenz nicht verletzt

$$db_i := (db_{i-1} \setminus \{\neg\chi_i\}) \cup \{\chi_i\}$$
$$log_i := neg(log_{i-1}, \chi_i) \cup \{\chi_i\} \cup constraints$$
$$ans_i := \chi_i \text{ wurde eingefügt.}$$

a)
PotSec: $\{beinbruch(torsten)\}$
Log: $\{armbruch(torsten), beinbruch(nils)\}$
DB: $\{beinbruch(nils), armbruch(torsten)\}$
Constraints: $\{armbruch(torsten)\}$
Update-Anfrage: $\{(not\ beinbruch(nils));\}$

gewünschtes Ergebnis:
Log: $\{armbruch(torsten), (not\ beinbruch(nils))\}$
DB: $\{beinbruch(), armbruch(torsten)\}$
Ausgabe: Änderungs-Operation durchgeführt.

tatächliches Ergebnis:
Log: $\{armbruch(torsten), (not\ beinbruch(nils))\}$
DB: $\{armbruch(torsten)\}$
Ausgabe: Änderungs-Operation durchgeführt.

b)
PotSec: $\{beinbruch(torsten)\}$
Log: $\{armbruch(torsten), (not\ beinbruch(nils))\}$
DB: $\{armbruch(torsten)\}$
Constraints: $\{armbruch(torsten)\}$
Update-Anfrage: $\{beinbruch(nils);\}$

gewünschtes Ergebnis:
Log: $\{armbruch(torsten), beinbruch(nils)\}$
DB: $\{armbruch(torsten), beinbruch(nils)\}$
Ausgabe: Änderungs-Operation durchgeführt.

tatächliches Ergebnis:
Log: $\{armbruch(torsten), beinbruch(nils)\}$
DB: $\{armbruch(torsten), beinbruch(nils)\}$
Ausgabe: Änderungs-Operation durchgeführt.

12.2.13 Die Klasse SQLInteraction

Diese Klasse ist als SQL-Schnittstelle für die Datenbankzugriffe gedacht und soll dafür dienen, die bereits in den anderen Klassen enthaltenen SQL-Methoden hierhin auszulagern. In dieser Klasse sind hauptsächlich Methoden implementiert, die Oracle SQL-Anfragen generieren. Die Methoden beinhalten noch keine Transaktionsverwaltung, d.h. die Tabellen werden nicht ge*lock*t und mittels *commit* wieder freigegeben (siehe 12.3). Da die Methoden dementsprechend SQL-Syntax enthalten, verweisen wir auf die Oracle Dokumentation. Welche Methode welche SQL-Anfrage generiert, kann der API des Prototypens auf der beigelegten CD und den Folien im Anhang entnommen werden.

12.3 ToDos

- Automatische Anmeldung im zugehörigen Schema gemäß Zensor. Falls der Lügenzensor ausgewählt wurde, findet die Anmeldung im Schema jcqe_lying statt. Ansonsten, also bei Wahl des Verweigerungs- oder des kombinierten Zensors, wird der Nutzer im Schema jcqe_normal angemeldet. Beachte den Umgang mit den Administratoren jcqe_normal und jcqe_lying.
- Auslesen der $Oracle - MetaData$ in SQLInteraction effizienter gestalten. Jede Kommunikation mit der Datenbank kostet viel Zeit und daher auch jeder Zugriff auf Metadaten (Spaltennamen usw.) einer Tabelle. Deswegen der Vorschlag beim Programmstart schon einige $MetaData$ viel verwendeter Tabellen auszulesen und in geeigneten Datenstrukturen (wie z.B. HashTables) zwischenzuspeichern. Zusätzlich muss eine effiziente Methode gefunden werden, die $MetaData$ der Tabellen auszulesen. Zur Zeit geschieht das über eine Anfrage $SELECT *$ ... über welche dann die MetaData ausgelesen werden können. Es ist bis jetzt keine bessere Lösung bekannt, sich aber alle Tupel ausgeben zu lassen, nur um an die Spaltennamen zu gelangen, ist sehr ineffizient.
- Änderungsoperationen auf mehrspaltigen Relationen wurde nicht implementiert. Das Auslesen der Benutzereingabe wurde bis jetzt nur für einspaltige Relationen implementiert. Damit ein ViewUpdate auf allen atomaren Formeln vollzogen werden kann, muss das Parsen des Wertes bezüglich des Syntaxbaumes in Konstanten und Relation angepasst werden. (vgl. Abschnitt 12.2.11)
- Konzept der Constraints überarbeiten. Zur Zeit werden die Constraints in einer Tabelle als Formeln (bzw. Strings) gehalten und jeder Benutzer hat ein Flag in der Benutzertabelle, ob die Constraints schon im initialen Log stehen. Falls nein, werden diese einmalig hinzugefügt.
- Überlegen, ob eine Umstellung auf „PreparedStatements“ für SQL-Anfragen in SQLInteraction in Hinblick auf Performance und Sicherheit sinnvoll ist.
- Ein weiteres Problem stellt zur Zeit die Definition der $< user >$ -Geheimnisse und des -Vorwissens dar. Sinnvoll wäre, einen Benutzer erst nach Erstellung von Geheimnissen und Vorwissen durch den Admin freischalten zu können.

Wenn ein Admin einen Benutzer hinzufügt, dann sollte die Option implementiert werden, dass der Oracle-Benutzer erst deaktiviert ist und noch freigeschaltet werden muss. Direkt nach der Erstellung eines neuen Benutzers per GUI, müsste dem Admin die Frage per Java-Interaktionsfenster „JOptionPane" gestellt werden, ob er die Geheimnisse jetzt oder zu einem späteren Zeitpunkt definieren möchte. Wenn er die Geheimnisse direkt anlegen möchte, dann müsste ihm der „UserEditFrame" eingeblendet werden, in dem er die Geheimnisse und das Vorwissen anlegen kann. Beim „Speichern" der Daten müsste der Benutzer auf Oracle-Ebene aktiviert werden. Falls der Admin sich dazu entscheidet die Geheimnisse zu einem späteren Zeitpunkt anzulegen, dann muss der Status des Benutzers in der Benutzertabelle festgehalten werden und eine Möglichkeit gegeben sein, dass der Admin den Benutzer nachträglich aktivieren kann. Ob in diesem Fall Geheimnisse angelegt worden sind, obliegt dem Wissen des Admins.

- Die Besitzer der Schemata (also zwei der Administratoren) erhalten die Rechte um Oracle-Benutzer anlegen zu können. Allen anderen Administratoren wird laut der GUI des Prototypen erlaubt, Benutzer hinzuzufügen. Dieses löst aber mangels Privilegien eine Exception aus. Es muss über ein geeignetes „Separation of Duty"-Konzept nachgedacht werden.

- In der Klasse SQLInteraction müssen die Zugriffe auf die Datenbanktabellen mittels $Lock()$ ($Row - Level - Locking$) abgesichert werden. Die Methode $Lock()$ ist schon implementiert, sie muss allerdings noch in allen anderen Methoden eingebunden werden, damit bei einem Mehrbenutzersystem keine Inkonsistenz entsteht, weil zwei Prozesse des Prototypen gleichzeitig auf eine Tabelle „schreibend oder lesend" zugreifen. (Transaktionsverwaltung)

Kapitel 13

Entwurf der Sicht-Erneuerung

13.1 Idee beschreiben

13.1.1 Was ist das Problem?

Die Vorgängerprojektgruppe 495 hat (inferenzfreie) Sicht-Erneuerungen nicht berücksichtigt. Allerdings ist im Nachgang in der Diplomarbeit von Gogolin [Gog08] ein Sicht-Erneuerungsalgorithmus auf theoretischer Ebene entstanden. Dieser Algorithmus behandelt die Erneuerung der Sicht des Benutzers nach einer Datenbankaktualisierung durch den Administrator.
Zusätzlich haben wir noch folgende Probleme aufgedeckt bzw. behandelt:

- Kompatibilität zum Sicht-Änderungsalgorithmus
- Kompatibilität mit Optimierer für Anfragen
- Kompatibilität mit Prototyp
- Prototyp in Prädikatenlogik, Arbeiten in Aussagenlogik

13.1.2 Warum ist das Problem ein Problem?

Dieser Algorithmus wurde jedoch nicht umgesetzt. Daher lag es an uns, diesen umzusetzen. Dies ist zusätzlich ein Problem, da hierdurch Inferenzen entstehen können. Siehe hierzu die Seminarausarbeitung von Benet [Ben09].

13.1.3 Wie sieht die Lösung aus?

Die theoretische Lösung existierte bereits in der Diplomarbeit von Gogolin.

Wir betrachten zunächst nur Sicht-Erneuerungsoperationen, die mit Literalen arbeiten und je Änderung nur ein Literal ändern.
Transaktionen wurden in der theoretischen Arbeit auch betrachtet. Um diese auch noch zu implementieren, fehlte uns jedoch die Zeit. Die Kompatibilität mit dem Sichtänderungsalgorithmus von Seiler wird zudem in „Requirements and protocols for inference-proof interactions in information systems“ von Biskup, Gogolin, Seiler, Weibert untersucht. Auch für die Umsetzung dieses Ansatzes fehlte uns die Zeit.

13.1.4 Wieso ist die Lösung eine Lösung?

Die theoretischen Einzelheiten hierzu sind in der Seminarausarbeitung von Benet bzw. in der Diplomarbeit von Gogolin zu finden, weshalb auf diese hier nicht noch einmal detailliert eingegangen werden soll. [Ben09, Gog08]
Der Korrektheitsbeweis des Änderungsalgorithmus findet sich sowohl in der Diplomarbeit von Gogolin (Kapiel 6.3) selbst als auch in dem Paper „Requirements and protocols for inference-proof interactions in information systems“ von Biskup, Gogolin, Seiler und Weibert. [Gog08, JBW08]

13.2 Operationalisierung

13.2.1 Aktivitätsdiagramm Sicht-Erneuerung

In Abbildung 13.1 auf Seite 192 wird der Ablauf des Algorithmus zur Sicht-Erneuerung aus der Diplomarbeit von Gogolin [Gog08] dargestellt. Hinzugefügt wurden die Abfragen bezüglich der Benutzerrolle und der ausschließlichen Verwendung von Literalen in den Benutzerlogs. Dies wurde in der Diplomarbeit von Gogolin noch nicht berücksichtigt. Dort wurde davon ausgegangen, dass dies nur für einen Benutzer stattfinden muss. Auch hier muss berücksichtigt werden, dass alle Benutzerlogs aktualisiert werden müssen.

13.2.2 Sequenzdiagramm Sicht-Erneuerung - Logaktualisierung

In Abbildung 13.4 auf Seite 195 wird die Aktualisierung der Benutzerlogs nach einer Datenbankänderung dargestellt. Dies stellt nur einen Ausschnitt dieser Aktualisierung dar und erhebt daher keinen Anspruch auf Vollständigkeit. Hierdurch soll ausschließlich die Prüfung auf Inferenzen und die Benachrichtigung des Benutzers dargestellt werden. Bevor dies geschieht, wurde die Datenbank bereits aktualisiert, nachdem sie auf eventuelle Inkonsistenzen geprüft wurde. Der besseren Übersicht wegen wurden hier die Interaktionen mit den Parserklassen, den Klassen Database Conf und TreeNode und den Javaklassen ausgeblendet. Der im Sequenzdiagramm dargestellt Ablauf wird ausführlich im Abschnitt „Änderungsmitteilung “auf 201 dargestellt.

13.2.3 Die neue Klasse ViewUpdateAdmin

Da die Klasse ViewUpdate mit den Methoden beider Sichtänderungsalgorithmen zu voll und zu unübersichtlich wurde, haben wir uns dazu entschlossen, die Methoden für beide Sichtänderungsalgorithmen mittels zweier Klassen voneinander zu trennen. Das Ziel dieser Trennung war es, besagte Fülle und Unübersichtlichkeit zu verhindern und die Implementierung strukturierter zu gestalten. Desweiteren bezweckten wir

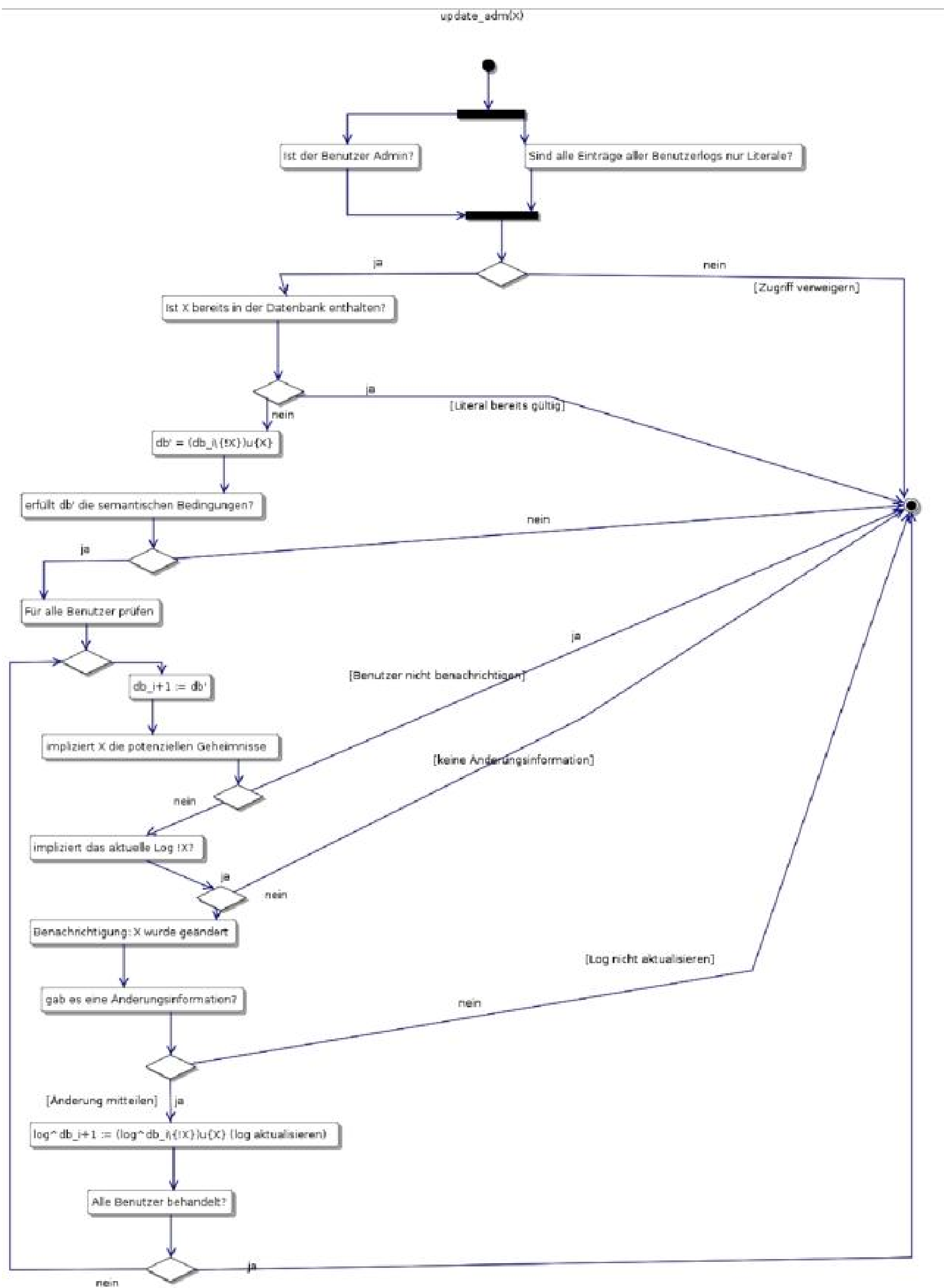


Abbildung 13.1: Aktivitätsdiagramm Sicht-Erneuerung

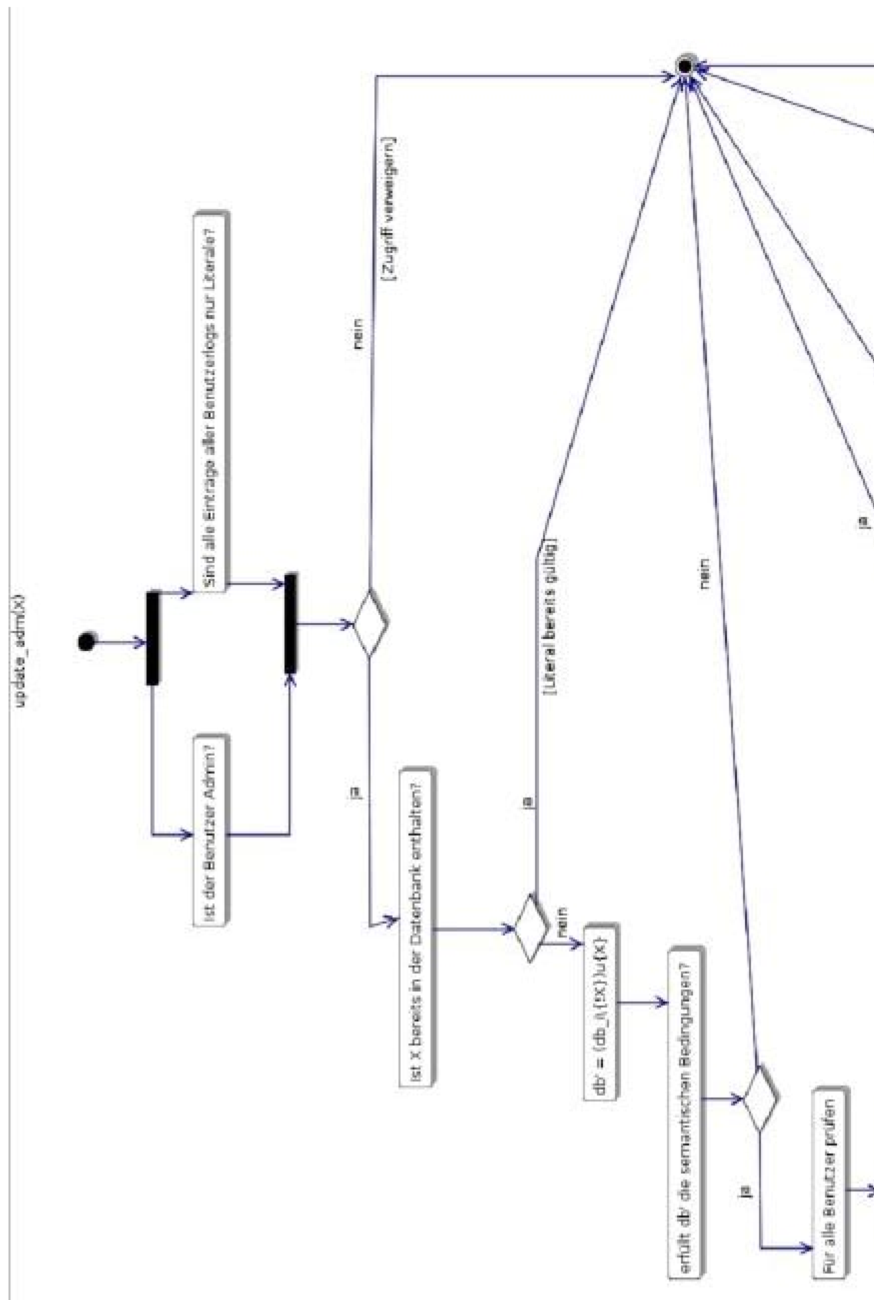


Abbildung 13.2: Aktivitätsdiagramm Sicht-Erneuerung - Oberer Teil

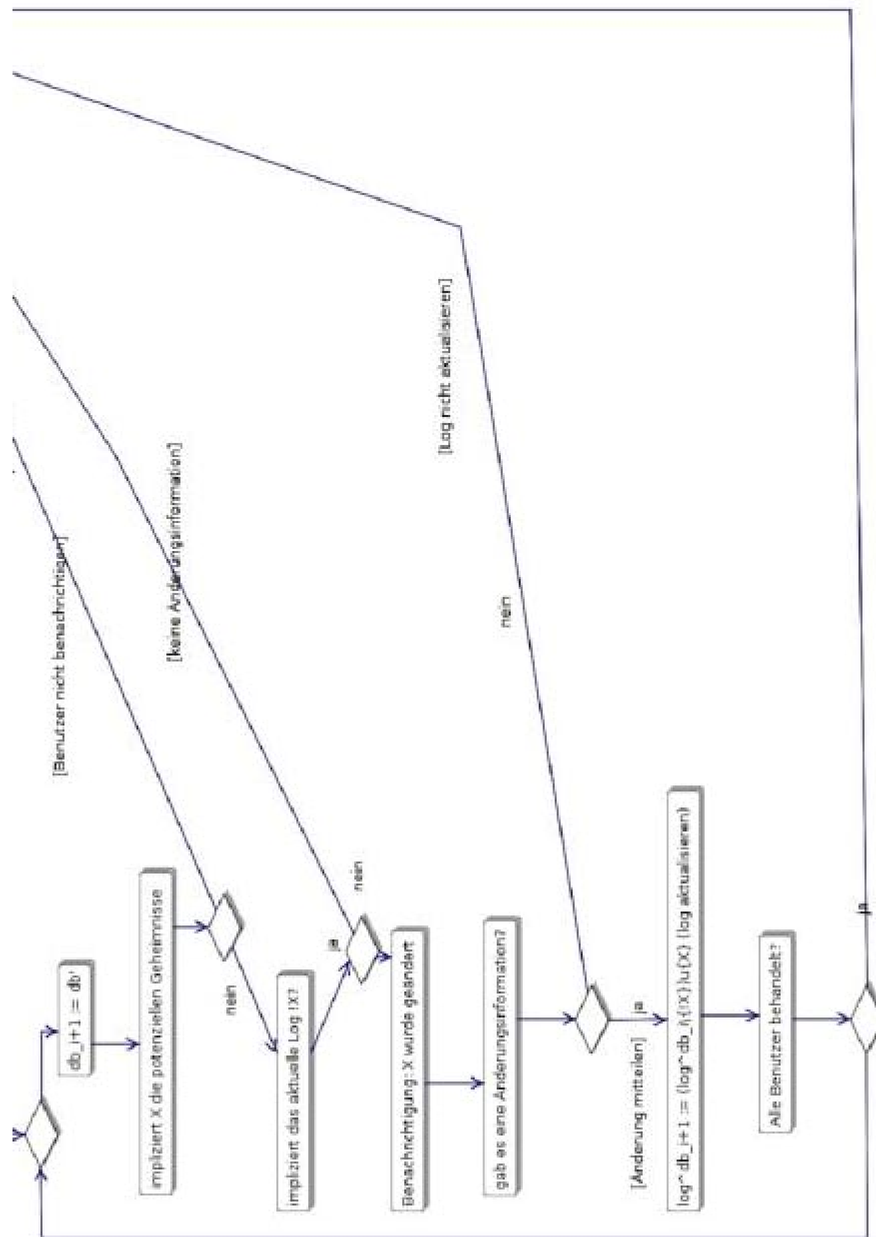


Abbildung 13.3: Aktivitätsdiagramm Sicht-Erneuerung - Unterer Teil

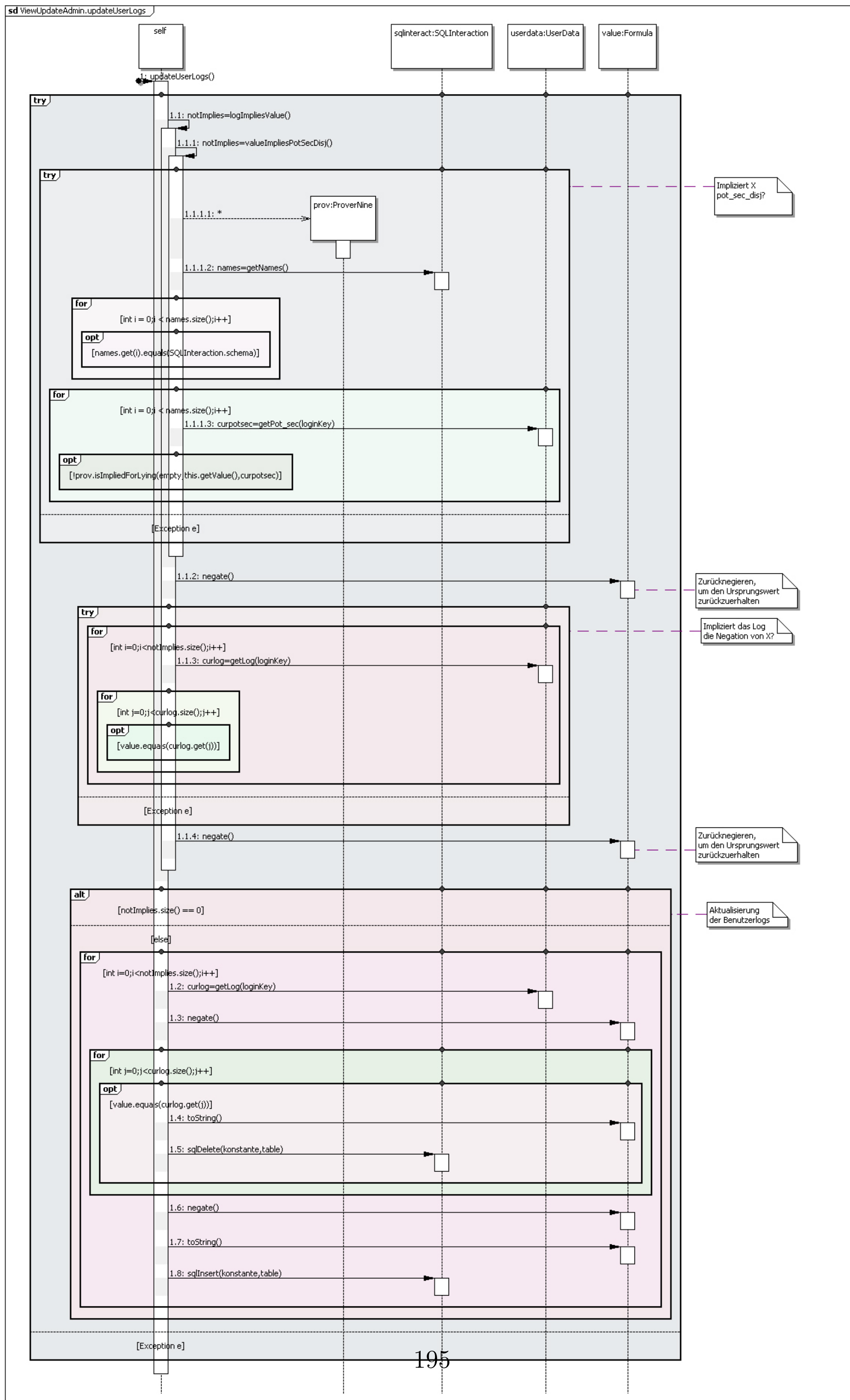


Abbildung 13.4: Aktualisierung des Benutzerlogs nach Admin Update

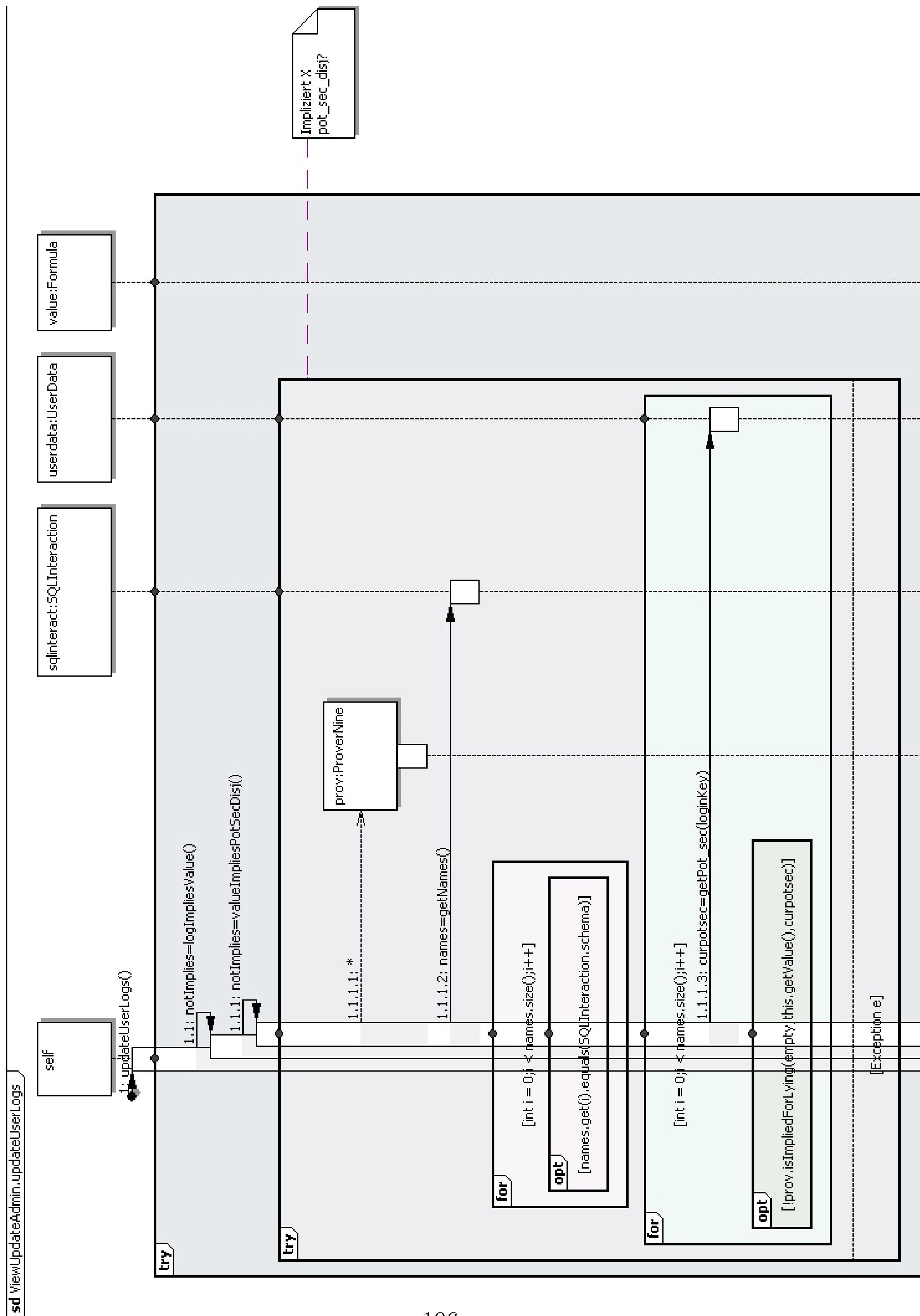


Abbildung 13.5: Aktualisierung des Benutzerlogs nach Admin Update - Oberer Teil

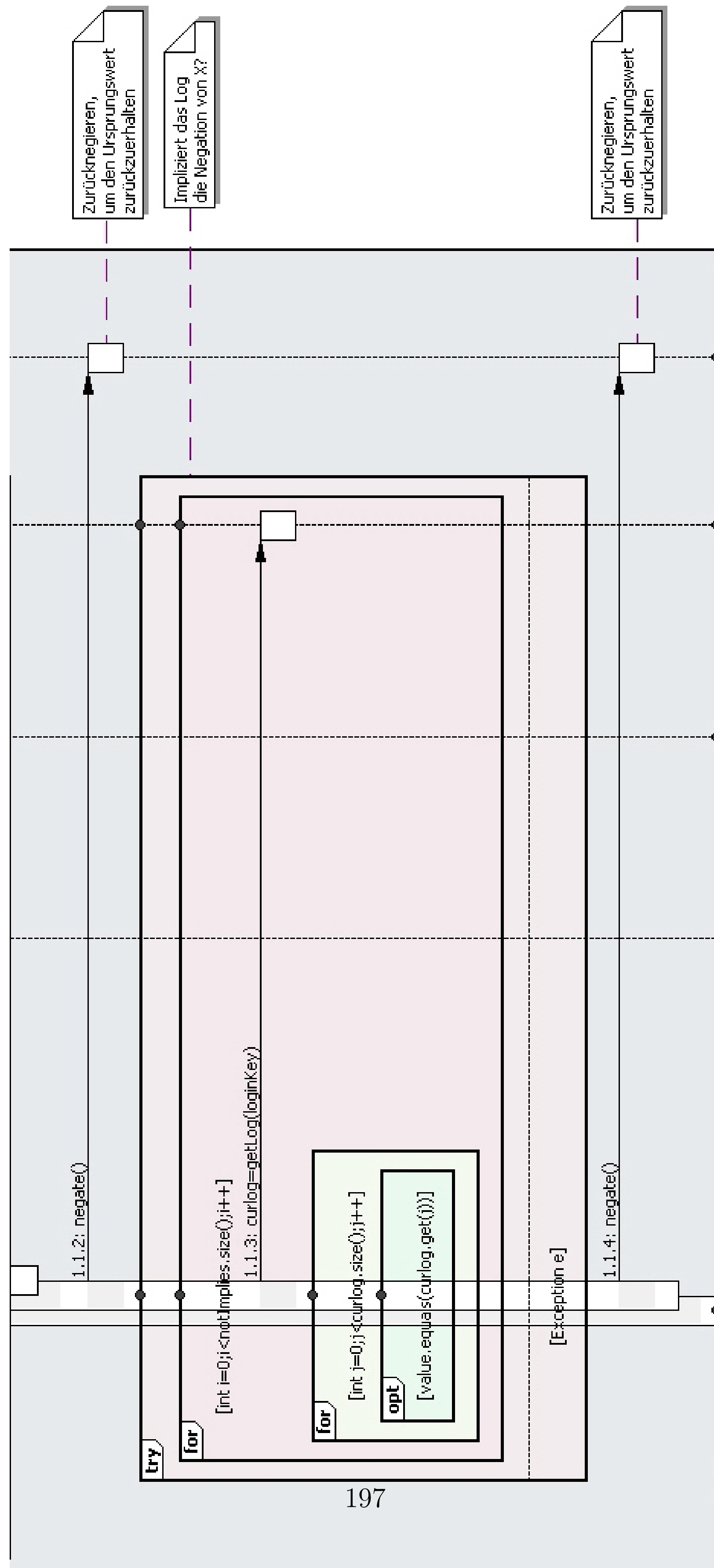


Abbildung 13.6: Aktualisierung des Benutzerlogs nach Admin Update - Mittlerer Teil

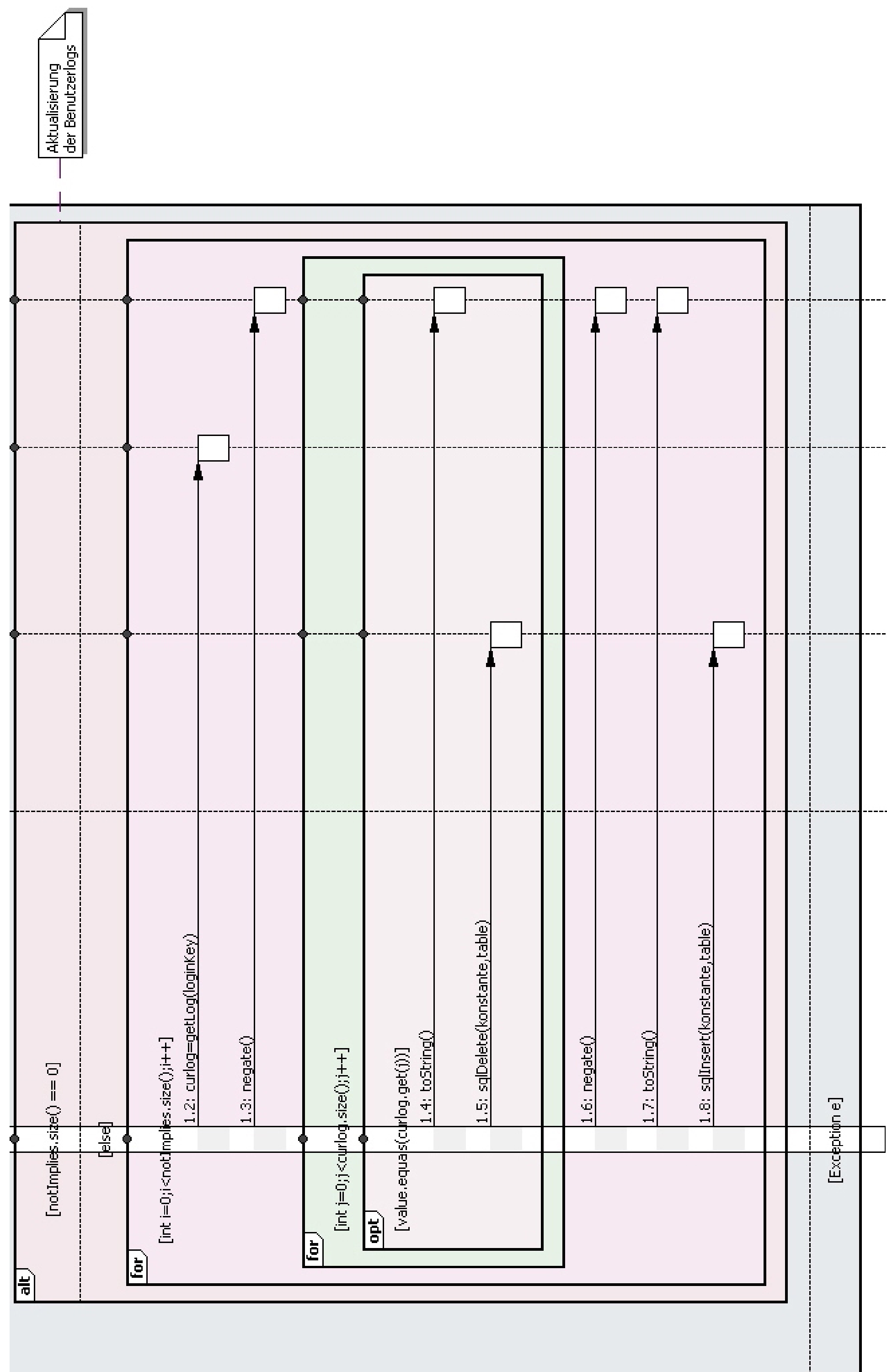


Abbildung 13.7: Aktualisierung des Benutzerlogs nach Admin Update - Unterer Teil

damit, uns bei der gleichzeitigen Arbeit an der Klasse nicht gegenseitig zu behindern. Die Klasse initialisiert in ihrem Konstruktor ein Objekt vom Typ ViewUpdate und kann somit auf die Funktionalität der Klasse ViewUpdate zurückgreifen.

13.2.4 Der atomare Sicht-Erneuerungsalgorithmus

Im Folgenden wird der Stand der Implementierung des Sicht-Erneuerungsalgorithmus (*update_adm*()) vorgestellt. Dieser stellt die vollständige Implementierung des Sicht-Erneuerungs-algorithmus unter ausschließlicher Verwendung von Literalen dar. Die volle Funktionalität unter besagten Voraussetzungen ist gegeben und wird durch die Testfälle in Abschnitt 13.2.5 auf Seite 206 für den Administrator und in Abschnitt 13.2.5 auf Seite 207 für die Nutzer belegt.

Prüfen der Nutzerrolle

Zusätzlich zu den von [Gog08] vorgestellten Fällen, muss hier noch die Abfrage stattfinden, ob sich der aktuelle Benutzer in der Rolle des Administrators befindet. Diese Abfrage wird in der Diplomarbeit von [Gog08] - von der Kapitelüberschrift abgesehen - nicht explizit erwähnt. Außerdem fragen wir noch ab, ob der aktuell zu ändernde Wert ein Literal ist, da wir hier nur den Sicht-Erneuerungsalgorithmus für atomare Änderungen umsetzen und dieser nur auf Literalen arbeitet.
Hier wird geprüft, ob der aktuell angemeldet Benutzer die Rolle des Administrators inne hat und ob der zu ändernde Wert ein Literal ist. Ist eines der beiden nicht der Fall, so wird die Antwort mittels der LinkedList für den Typ String mit dem Namen *answers* (siehe nachfolgendem Code) an die GUI übergeben und von dieser auf dem Bildschirm ausgegeben.

```
1 if((userdata.getRole() != "Administrator") && (value.isGroundatom()
      != true)){
2   System.out.println("Zugriff verweigert");
3   setAns("Zugriff verweigert");
4   this.answers.add(0,"Zugriff verweigert");
5   this.answers.add(1,"Abbruch");
6   return this.getAnswers(); //Abbruch
7 }
```

Existieren in der Datenbank nur Literale?

Anschließend wird geprüft, ob in den Benutzerlogs ausschließlich Literale verwendet werden.

```
for(int i=0;i<names.size()-1;i++){
  String usname = names.get(i);
  curlog = userdata.getLog(usname);
  for(int j=0;j<curlog.size()-1;j++){
    if(!curlog.get(j).isGroundatom()){
    this.answers.add(0,"Im Log werden keine Literale verwendet.");
    this.answers.add(1,"Abbruch");
    return this.getAnswers();
    }
  }
}
```

Prüfen der semantischen Bedingungen

Die Gültigkeit der semantischen Bedingungen, welche in der Arbeit von [Gog08] mit $db' \models log_0$ notiert wird, wobei log_0 die semantischen Bedingungen darstellt, welche auch als *constraints* bezeichnet werden, wird bereits beim Starten des Prototypen geprüft. Daher brauchen wir zur Laufzeit des Algorithmus nur noch zu prüfen, ob der zu aktualisierende Wert die semantischen Bedingungen verletzt. Dies erledigt die Methode $checkValueViolatesConstraints()$. Die in der Diplomarbeit von [Gog08] vorgestellte „vorübergehende Datenbankinstanz“ und das damit einhergehende temporäre Auslesen derselben sparen wir uns hiermit.

```
public boolean checkValueViolatesConstraints(Formula value){
  boolean check = false;
  try{
    value = this.getValue();
    value.negate();
    if(this.constraints.contains(value)){
      check = true;
    }
    value.negate(); //Zuruecknegieren

  }catch(Exception e){
    System.out.println(e.getMessage());
```

```
13   }return check;
14 }
```

Einfügen des Wertes

Werden die Konsistenzbedingungen nicht verletzt, so fährt der Algorithmus fort und prüft, ob der Wert bereits enthalten ist, oder nicht. Anschließend wird entsprechend dem Literal (im Code *value*) eine Zeile in die entsprechende Tabelle eingefügt (bei positivem Vorzeichen) bzw. gelöscht (bei negativem Vorzeichen). Dies erledigt die Methode *insert*().

```
1  public void insert(){
2    try{
3      if (this.getNOTvalueflag()==0){ //falls NOTvalueflag = 0
           beginnt Wert mit (NOT
4        sqlinteract.sqlDelete(this.getKonstante(), this.getTable());
             //loescht den Wert aus der Tabelle
5      }
6      else{
7      sqlinteract.sqlInsert(this.getKonstante(), this.getTable()); //
           fuegt den Wert in die Tabelle ein
8      }
9    } catch(Exception e){
10     System.out.println("Exception: Fehler in insert "+e.getMessage
           ());
11   }
12 }
```

Änderungsmitteilung

Wenn etwas in der Datenbank geändert wurde, müssen die Benutzer es erfahren, die schon einmal danach gefragt hatten (siehe Sequenzdiagramm 13.4 Punkt 1.1) und bei denen dies nicht zu Inferenzen führt.
Anders als im von [Gog08] vorgestellten Algorithmus, bei dem nur ein Benutzer berücksichtigt wurde, müssen hier mehrere Benutzer berücksichtigt werden. Die Aktualisierung aller Benutzerlogs erledigt die Methode *updateUserLogs*(). Diese ruft zuerst die Methode *logImpliesValue*() auf, welche überprüft, ob die Negation des

Wertes, der geändert wurde, in dem Benutzerlog steht (siehe Sequenzdiagramm 13.4 Punkt 1.1).
Diese Methode wiederum ruft die Methode *valueImpliesPotSecDisj*() auf (siehe Sequenzdiagramm 13.4 Punkt 1.1.1).
valueImpliesPotSecDisj() bekommt die Liste aller Benutzer als Eingabe (siehe Sequenzdiagramm 13.4 Punkt 1.1.1.2) und filtert die Benutzer raus, bei denen ein Geheimnis impliziert wird. Hierbei muss beachtet werden, dass man anstelle des Logs eine leere Liste an Prover9
(*isImpliedForLying*()) übergibt, da sonst die Benutzer, die eine Aktualisierung des Benutzerlogs erhalten sollten keine bekommen, weil Prover9 dann Implikationen aus dem Log auswertet. Desweiteren wird hier im Vorfeld schon der Besitzer des aktuellen Schemas, der gleichzeitig Administrator des Schemas ist, herausgefiltert, bevor die Liste der Benutzer durchlaufen wird (siehe Sequenzdiagramm 13.4 *names.get*(*i*).*equals*(*SQLInteraction.schema*)).

```
1  public LinkedList<String> valueImpliesPotSecDisj(){
2    LinkedList<String> temp = new LinkedList<String>();
3    try{
4      ProverNine prov = new ProverNine();
5      this.names =sqlinteract.getNames();
6      for (int i = 0; i < names.size(); i++){
7        if (names.get(i).equals(SQLInteraction.schema)) {
8          names.remove(i);
9          break;
10       }
11     }
12     for(int i = 0; i < names.size(); i++){
13       String username = names.get(i);
14       LinkedList<Formula> empty = new LinkedList<Formula>();
15       this.curpotsec = userdata.getPot_sec(username);
16       if (!prov.isImpliedForLying(empty,this.getValue(),curpotsec))
           {
17         temp.add(username);
18       }
19     }
20   } catch(Exception e){
21     System.out.println("Exception: Fehler in valueImpliesPotSecDisj
          "+e.getMessage());
22   }
23   return temp;
```

```
24 }
```

Diese Namensliste wird dann an *logImpliesValue* übergeben, welche alle Benutzer rausfiltert, bei denen das Log nicht die Negation des geänderten Wertes impliziert (siehe Sequenzdiagramm 13.4 *value.equals*(*curlog.get*(j))), also alle Benutzer, die noch nie nach diesem Wert gefragt haben. Man möchte nur den Benutzern eine Änderungsmitteilung geben, die schon einmal nach dem Wert gefragt haben.

```
1  public LinkedList<String> logImpliesValue(){
2    LinkedList<String> temp = new LinkedList<String>();
3    LinkedList<String> notImplies = this.valueImpliesPotSecDisj();
4    value.negate();
5    try{
6      for(int i=0;i<notImplies.size();i++){
7        String username = notImplies.get(i);
8        curlog = userdata.getLog(username);
9        for(int j=0;j<curlog.size();j++){
10         if(value.equals(curlog.get(j))){
11           temp.add(username);
12           break;
13         }
14       }
15     }
16   } catch(Exception e){
17     System.out.println("Exception: Fehler in logImpliesValue"+e.
          getMessage());
18   }
19   value.negate();
20   return temp;
21 }
```

Die Benutzer, die dann noch übrig bleiben, erhalten die Änderungsinformation, in Form eines Logeintrags. D.h. der alte Wert wird aus dem Log gelöscht (siehe Sequenzdiagramm 13.4 Punkt 1.5) und der neue Wert wird in das Log eingefügt (siehe Sequenzdiagramm 13.4 Punkt 1.8 und die hier folgenden Code Zeilen 14 bis 26).

```
1  public void updateUserLogs(){
2    try{
3  //      4.1)
4      LinkedList<String> notImplies = logImpliesValue();
5  //      4.2)
6      if (notImplies.size() == 0){
```

```
7         System.out.println("Es gibt keine Aenderungsinformation fuer
              andere Nutzer.");
8         this.answers.add(0,"Es gibt keine Aenderungsinformation fuer
              andere Nutzer.");
9         this.answers.add(1,"Aenderung in der Datenbank erfolgreich");
10        return;
11      }
12      else{
13        System.out.println("Es gibt eine Aenderungsinformation fuer
              andere Nutzer.");
14        for(int i=0; i<notImplies.size(); i++){
15          String username = notImplies.get(i);
16          curlog = userdata.getLog(username);
17          value.negate();
18          for(int j=0;j<curlog.size();j++){
19            if (value.equals(curlog.get(j))){
20              this.sqlinteract.sqlDelete(value.toString(), notImplies
                    .get(i)+"Log");
21              break;
22            }
23          }
24          value.negate();
25          this.sqlinteract.sqlInsert(value.toString(), notImplies.get
                (i)+"Log");
26          }
27        System.out.println("Es gibt eine Aenderungsinformation fuer
              andere Nutzer.");
28        this.answers.add(0,"Es gibt eine Aenderungsinformation fuer
              andere Nutzer.");
29        this.answers.add(1,"Aenderung in der Datenbank erfolgreich");
30      }
31    } catch(Exception e){
32      System.out.println("Exception: Fehler in updateUserLogs "+e.
          getMessage());
33    }
34  }
```

In den Methoden *logImpliesValue*() und *updateUserLogs*() muss zusätzlich noch die Mehrfachnegation des Wertes beachtet werden, die durchgeführt wird, um den negierten bzw. ursprünglichen Wert zu erhalten. Dieses wiederholte Negieren ist notwendig, da der Wert bei jeder Änderung in derselben Variable gespeichert wird

und der Wert dadurch dem zuletzt berechneten Wert entspricht.

13.2.5 Test des Sicht-Erneuerungsalgorithmus

Bei den Testfällen gibt es zwei Unterscheidungen zu machen. Die Testfälle, die ausschließlich den Administrator betreffen, und die Testfälle, die ausschließlich die Benutzer betreffen. Der Administrator führt die Aktualisierung durch und muss daher Konsistenzen beachten. Die Benutzer müssen über die Änderungen informiert werden, sofern keine Geheimnisse impliziert werden und sie schon einmal nach dem Wert gefragt haben. Die Fälle des Benutzers fallen alle in den Fall des Administrators, in dem der Wert geändert wird, da er vorher noch nicht so in der Datenbank enthalten war.

Administrator **Test - Fall 1** [$Wert\ enthalten$]

DB: $armbruch\{tim, uli, horst\}$
Update-Anfrage: $armbruch(horst)$;
Ergebnis: Ausgabe: Die Formel ist bereits in der Datenbank enthalten.

gewünscht:
DB: $armbruch\{tim, uli, horst\}$
Ergebnis: Ausgabe: Die Formel ist bereits in der Datenbank enthalten.

Test - Fall 2 [$Wert\ NICHT\ enthalten$]

DB: $armbruch\{tim, uli, horst\}$
Update-Anfrage: $armbruch(fred)$;
Ergebnis: Ausgabe: Abhï¿½ngig von BenutzerLog - Aktualisierung.

gewünscht:
DB: $armbruch\{tim, uli, horst, fred\}$
Ergebnis: Ausgabe: Abhï¿½ngig von BenutzerLog - Aktualisierung.

Test - Fall 3 [$Konsistenzverletzung$]

Constraints: $\{(notarmbruch(horst))\}$
DB: $armbruch\{tim, uli, horst\}$
Update-Anfrage: $armbruch(horst)$;
Ergebnis: Ausgabe: Konsistenzverletzung. Abbruch

gewünscht:
DB: $armbruch\{tim, uli, horst\}$
Ergebnis: Ausgabe: Konsistenzverletzung. Abbruch

Benutzer Test - Fall 1 [$Wert\ impliziert\ ein\ Geheimnis$]

$pot_sec(User1_jcqelying)$: $\{armbruch(fred)\}$
$Log(User1_jcqelying)$: $\{(not\ armbruch(fred))\}$
DB: $armbruch\{tim, uli, horst\}$
Update-Anfrage: $armbruch(fred)$; (vom Administrator durchgeführt)
Ergebnis: $Log(User1_jcqelying)$: $\{(not\ armbruch(fred))\}$

gewünscht:
Ergebnis: $Log(User1_jcqelying)$: $\{(not\ armbruch(fred))\}$

Test - Fall 2 [$Wert\ impliziert\ kein\ Geheimnis\ und\ Negation\ des\ Wertes\ steht\ nicht\ im\ Log$]

$pot_sec(torsten_jcqelying)$: $\{\}$
$Log(torsten_jcqelying)$: $\{\}$
DB: $armbruch\{tim, uli, horst\}$
Update-Anfrage: $armbruch(fred)$; (vom Administrator durchgeführt)
Ergebnis: $Log(torsten_jcqelying)$: $\{\}$

gewünscht:
Ergebnis: $Log(torsten_jcqelying)$ $\{\}$

Test - Fall 3 [$Wert\ impliziert\ kein\ Geheimnis\ und\ Negation\ des\ Wertes\ steht\ im\ Log$]

$pot_sec(gruppe4_jcqelying)$: {}
$Log(gruppe4_jcqelying)$: $\{(not\ armbruch(fred))\}$
DB: $armbruch\{tim, uli, horst\}$
Update-Anfrage: $armbruch(fred)$; (vom Administrator durchgeführt)
Ergebnis: $Log(gruppe4_jcqelying)$: {}

gewünscht:
Ergebnis: $Log(gruppe4_jcqelying)$: $\{armbruch(fred)\}$

13.2.6 Erstellung der Klasse SQLInteraction

Diese Klasse ist als SQL-Schnittstelle für die Datenbankzugriffe gedacht und soll dazu dienen, die bereits in den anderen Klassen enthaltenen SQL-Methoden hierhin auszulagern.

13.2.7 Änderunggen in der Klasse SQLInteraction

In der Klasse SQLInteraction haben wir die Variable *schema* eingefügt, in die das Datenbankschema aus der GUI heraus übergeben wird und die als statische Variable in alle Klassen übergeben wird, in denen noch SQL-Anfragen sind.
Desweiteren haben wir die Methode $getNames()$ umgeschrieben. Sie enthält jetzt nicht mehr die gesamte Funktionalität, sondern nur noch einen Aufruf der allgemeinen Methode $getSingleColumnContent(talbe, column)$ mit den Argumenten $schema +$ “$.Benutzertabelle''$ und “$LOGIN''$. Letztere enthält nun die Funktionalität.

13.2.8 Änderungen in der Klasse ChildQuery

Zu Testzwecken hatten wir bereits den Button Update_adm eingefügt.
Dieser ist nun fester Bestandteil der GUI und wird nur angezeigt, wenn man sich mit der Rolle des Administrators in der Datenbankinstanz mit dem Lügenzensor anmeldet, was erst durch die Einführung des Mehrbenutzersystems (siehe Kapitel

11) möglich wurde.
Da sich durch die Änderungen bzgl. der freien Variablen auch die Methode *cqe*() in der Klasse Application geändert hat und unsere Änderungsalgorithmen nicht mehr funktionierten, waren wir gezwungen eine neue Methode *cqe_update*() zu schreiben, welche zusätzlich die Laufzeit unserer Algorithmen verringert.
Daher mussten wir auch in der Klasse ChildQuery den Methodenaufruf an der entsprechenden Stelle ändern.
Weiterhin ändert sich durch die neue Klasse ViewUpdateAdmin auch das in besagtem Button verwendete Objekt zu ViewUpdateAdmin.

13.3 Einpassung in den Prototypen

13.4 ToDos

- Transaktionen
  Da wir es zeitlich nur geschafft haben den theoretischen Ansatz von Gogolin für Transaktionen (siehe hierzu [Gog08] Kapitel 7) zu implementieren, ist dies noch ein offener Punkt. Dessen Korrektheitsbeweis findet man in „Requirements and protocols for inference-proof interactions in information systems " [JBW08] von Biskup, Gogolin, Seiler, Weibert zur Kompatibilitätsprüfung der Algorithmen von Seiler zur Sichtänderung und Gogolin zur Sicht-Erneuerung.

- Kompatibilität zu Sicht-Änderungen
  Durch die von uns implementierte Trennung von Administrator Update und Benutzer Update mittels Rolle und GUI, sind bereits beide Algorithmen voneinander getrennt und parallel einsetzbar. Meldet man sich als einfacher Benutzer an, kann man nur Benutzer Updates durchführen und als Administrator nur die Administrator Updates. Die Aktualisierung der Benutzerlogs funktioniert bei beiden Implementierungen gleich.

- Literale mit mehreren Attributen
  Auf Grund eines Missverständnisses glaubten wir Literale wären atomare Formeln mit nur einem Attribut, d.h. es können nur Tabellen mit einer Spalte

korrekt bearbeitet werden. Diesen Fehler gilt es noch zu beheben.

Kapitel 14

Entwurf optimierter offener Anfragen

14.1 Problembeschreibung

Die PG 495 hat in ihrem Prototypen für geschlossene Anfragen einen Lügen-, einen Verweigerungs- und einen kombinierten Zensor implementiert.
Für offene Anfragen wurde bisher nur ein alternativer kombinierter Zensor implementiert, wobei die PG 495 die Anzahl der freien Variablen für ihren Prototypen auf kleiner oder gleich zwei beschränkt hat.[1] Für Anfragen mit mehr freien Variablen (also größer zwei) wird einfach nur eine Fehlermeldung ausgegeben.
Offene Anfragen werden im alternativen kombinierten Zensor durch die Simulation mit geschlossenen Anfragen ausgewertet. Dabei werden die freien Variablen durch Konstanten ersetzt. Weil die Menge der Konstanten unendlich sein kann, wird in der ersten Phase zunächst mit einem *Vollständigkeitstest* bestimmt, ab welchem Element der Konstantenmenge keine Substitutionen der Variablen mehr betrachtet werden müssen, weil diese die Anfrage falsch machen würden. Dabei muss aber auch beachtet werden, dass der Vollständigkeitstest selber auch kein Geheimnis aufdecken darf. Sollte dies dennoch der Fall sein, muss die Vollständigkeitsaussage solange angepasst werden, bis sie kein Geheimnis mehr impliziert. Das heißt, dass in Bezug auf das letzte positive Element gelogen wird. In der zweiten Phase können dann alle Elemente bis zum letzten positiven Element als geschlossene Anfragen betrachtet

[1] Die Prototypen [Xia08] und [Zha08] haben weitere Zensoren für offene Anfragen implementiert, der Quellcode konnte aber nicht integriert werden.

und dann mit dem kombinierten Zensor untersucht werden. Weitere Details zu der Vorgehensweise finden sich im Endbericht der PG495 [PG408] und in [BB07].

Wir wollen nun den Prototypen von Sebastian Sonntag um einen optimierten Verweigerungszensor für offene Anfragen erweitern. Dabei wollen wir uns aber nicht in der Anzahl der freien Variablen beschränken. Zudem soll die Auswertung der offenen Anfragen so geschehen, dass der Vollständigkeitstest nicht mehr benötigt wird, da dieser Vollständigkeitstest aufgrund der Bestimmung des Ausstiegspunktes der Simulation und der Implikationsüberprüfung der Vollständigkeitsaussage sehr aufwendig ist und eine Menge Laufzeit kostet.

14.2 Lösungsansätze

In diesem Kapitel wollen wir nun zwei Möglichkeiten vorstellen den Prototypen von Sebastian Sonntag zu erweitern, um bestimmte offene Anfragen optimiert beantworten zu können. Optimiert bedeutet, es soll keine Vollständigkeitsanalyse mehr benötigt werden und die Substitutionen sollen auf Konstanten, die in der Datenbank tatsächlich vorkommen, eingeschränkt werden. Zudem kann bei der Simulation durch geschlossene Anfragen der statische Zensor benutzt werden. Wir definieren zuerst eine Anfragesprache $L^{o*}_{q_nds}$, welche die optimierbaren Anfragen repräsentiert.

Definition 14.2.1 (Anfragesprache $L^{o*}_{q_nds}$)**:**
$L^{o}_{q_nds} := \{(\exists X_1)\dots(\exists X_l)R(v_1,\dots,v_n)\}$, *mit*
$0 \leq l \leq n,\ X_i \in Var,\ v_i \in Const\ \cup\ Var$
$\{X_1,\dots,X_l\} \subseteq \{v_1,\dots,v_n\}$, *und* $v_i \neq v_j$, *sofern* $v_i, v_j \in Var$
$\{v_1,\dots,v_n\}\backslash(\{X_1,\dots,X_l\}\ \cup\ Const) \subseteq Var$ *sind die freien Variablen*

Desweiteren muss auch die Geheimnissprache bestimmten Bedingungen genügen, diese sind jedoch die gleichen wie im geschlossenen Fall (siehe dazu [Bui09], [Jab09], [Son08]), also muss an dieser Stelle keine neue Sprache definiert werden. Außerdem dürfen Anfragen nur optimiert behandelt werden, sofern vorher kein Benutzerwissen, das im dynamischen Modus gewonnen wurde, Bedingung 5.1 verletzt. Diese Bedin-

gung besagt, dass keine Disjunktionen in der Formel vorkommen dürfen, Konjunktionen zwar erlaubt sind, aber eine existenzquantifizierte Variable nur in genau einem Konjunkt vorkommen darf und die Negation nur unmittelbar vor einem geschlossenen oder existenzquantifizierten Satz stehen darf (genaueres dazu siehe [Son08]). Nach jeder dynamischen Anfrage wird nun das neu hinzugewonnene Benutzerwissen auf Bedingung 5.1. geprüft und bei Verletzung dieser die neue boole'sche Variable *opt_possible* auf „false“ gesetzt. Ist diese Variable einmal auf „false“ gesetzt, ist eine weitere optimierte Prüfung ausgeschlossen.
Im Laufe dieser Projektgruppe trat zudem ein Problem bezüglich der zugrundeliegenden Theorie auf ([JBL09]), die den Schutz von Geheimnissen gefährdet. Dieses Problem soll nun erstmal anhand eines Beispiels aufgezeigt werden, um dann die daraus resultierenden Konsequenzen bei der Umsetzung des theoretischen Ansatzes zu erklären.
Die Interpretation der verwendeten Begriffe ist dabei wie folgt (angelehnt an [JBL09]):
Anfrage ϕ(X) $\equiv$ R(X) und r als aktuelle Datenbankinstanz

Menge der Antworten ans=*positive**refused*, wobei:

- *positive(ϕ(X),r)* = {ϕ(c)| c $\in$ Konst und r $\models_M$ ϕ(c)}
- *refused(ϕ(X),pot_sec)* = {ϕ(c)| c $\in$ Konst und ϕ(c) $\models$ Ψ mit Ψ $\in$ pot_sec}

Beispiel 14.2.1:
Es gelten die folgenden Belegungen der Relation R:

Relation R	*A*	*B*
	a	*b*
	c	*d*

$\Psi \equiv$ *R(a,b) ist ein Geheimnis*

Es wird nun eine Anfrage gestellt: $\phi_1 \equiv R(X,Y)$.
Die richtige Antwort ist dann ans = {R(c,d)}.

Nun wird eine zweite Anfrage der folgenden Form gestellt:
$\phi_2 \equiv (\exists X)R(a,X) \not\models \Psi$.
Die Antwort in dem Fall ist dann ans = (∃ X)R(a,X), weil diese Antwort an sich noch kein Geheimnis verrät. Durch die Kombination mit dem ersten Anfrageergebnis und dem Wissen des Nutzers über seine Geheimnispolitik, könnte der Nutzer

daraus allerdings schließen, dass R(a,b) auch gelten muss. Er weiß durch die zweite Anfrage, dass es ein R(a,X) gibt, das ihm bei der ersten Anfrage allerdings nicht ausgegeben wurde. Daraus, dass es ihm nicht ausgegeben wurde, kann er schliessen, dass es sich dabei um ein Geheimnis handelt und der Nutzer kann wegen dem Wissen über seine Geheimnispolitik sehen, dass R(a,b) in dem Fall das einzige Geheimnis ist, dass auf seine zweite Anfrage passt.

Es gibt bisher zwei noch nicht fundierte erste Ansätze, wie man verhindern könnte, dass dieses Problem ausgenutzt werden kann. Der erste wäre, dass die Geheimnispolitik dem Nutzer unbekannt bleibt. Da die theoretische Grundlage und der Prototyp allerdings grundsätzlich von bekannten Geheimnispolitiken ausgeht, wären die zu machenden Änderungen entweder sehr gravierend (bei Umsetzung unbekannter Geheimnispolitik im gesamten Prototyp) oder würden zu einer Abkapselung der offenen Anfragen führen (Nutzer müsste von Anfang an entscheiden, ob er irgendwann einmal offene Anfrage stellen möchte, so dass die Geheimnispolitik für ihn unbekannt bleibt bzw. darf bei bekannter Geheimnispolitik keine optimierte offene Anfragen stellen).
Der zweite Ansatz schränkt die Möglichkeiten der Anfragen ein. Wenn ein Nutzer einmal eine optimierte offene Anfrage gestellt hat, dann kann er später keine existentielle Anfrage mehr stellen bzw. wenn der Nutzer eine existentielle Anfrage gestellt hat, dann darf er danach keine optimierte offene Anfrage mehr stellen.
In dieser Version des Prototypen wurde der zweite Ansatz umgesetzt, der am Ende dieses Kapitels noch etwas ausführlicher dargestellt wird. Zur Umsetzung dieses Ansatzes wurden zwei weitere boole'sche Variablen (*openOpt* und *boundVar*) eingeführt, die zur Überprüfung der bereits gestellten Anfragen dienen. Die Variable *openOpt* gibt dabei an, ob bereits eine optimierte offene Anfrage gestellt wurde, die Variable *boundVar* gibt an, ob es bereits eine geschlossene Anfrage mit einer gebundenen Variable gab. Damit die Zustandsinformation des Benutzers auch bei einem Absturz bzw. Neustart des Prototypen nicht verloren geht, werden diese beiden Variablen und die bereits vorher erwähnte Variable *opt_possible* in der Datenbank in der Benutzertabelle für jeden Nutzer gespeichert.
Wegen dieses Problems muss die Anfragesprache der optimierbaren Anfragen noch weiter eingeschränkt werden:

Definition 14.2.2 (Anfragesprache $L^o_{q_nds}$)**:**

$L^o_{q_nds} := \{R(v_1, \ldots, v_n)\}$, *mit*
$\{v_1, \ldots, v_n\} \backslash Const)$ *sind die freien Variablen*

Nach der Überprüfung der Sprache geben wir Anfragen der Sprache $L^o_{q_nds}$ als SELECT-Anfrage an die Datenbank weiter. Die Antworten sind dann jene Teiltupel, bei gebundenen Variablen werden Teile wegprojeziert, in der Datenbank, die der Anfrage genügen. In der derzeitigen Version werden solche Projektionen durch gebundene Variablen, wie erwähnt, nicht erlaubt. Allerdings hat noch keinerlei Inferenzkontrolle stattgefunden und deshalb können wir diese Tupel nicht einfach ausgeben. In den Unterkapiteln 14.2.1 und 14.2.2 stellen wir je eine Möglichkeit vor, die gefährlichen Tupel auszusortieren. Vorher wollen wir jedoch kurz veranschaulichen, wieso keine Vollständigkeitsanalyse mehr benötigt wird. Gehen wir jetzt erstmal davon aus, die gefährlichen Tupel wurden schon aus der Antwortmenge entfernt und es werden nur solche Tupel zurückgegeben, die der Anfrage entsprechen und kein Geheimnis verraten. Die Zensoren in Sebastian Sonntags Prototypen gehen stets von *bekannten* Vertraulichkeitspolitiken aus. Also weiß der Nutzer, welche Antworten er nicht bekommt, weil sie für ihn geheim sind. Nun gelten auf jeden Fall die Tupel in der Datenbank nicht, die dem Benutzer nicht zurückgegeben wurden und nicht geheim sind. Auf diese Weise erschließt sich die Vollständigkeit dem Benutzer und es ist keine explizite Vollständigkeitsanalyse mehr nötig. Dabei ist das Wissen des Benutzers über die Vollständigkeit nicht gefährlich und muss im statischen Modus nicht berücksichtigt werden.
Allerdings muss das Wissen, das sich dem Nutzer erschließt, trotzdem im Benutzerlog gespeichert werden, falls eine Anfrage nicht mehr optimiert behandelt werden kann und im dynamischen Modus auf das Benutzerlog zugegriffen werden muss. Dieses Wissen lässt sich als Formel konstruieren, siehe [JBL09].

14.2.1 Nutzen der VPD-Funktion von Oracle PL/SQL

Eine Methode wäre, per Nutzen der VPD-Funktion von Oracle PL/SQL die gefährlichen Tupel auszusortieren. Die Oracle Virtual Private Database (VPD) bietet eine Zugriffskontrolle für Datenbanken, um Sicherheitsstrategien umzusetzen. Dies geschieht durch Erweiterung der Datenbankanfrage um eine, für den Nutzer nicht

sichtbare, WHERE-Klausel. (Näheres dazu in Kapitel 7 - Oracle-Virtual Private Database. [May09]) Bei uns soll das *pot_sec* per VPD-Funktion in diese WHERE-Klausel transformiert und an die im Vorfeld in Kapitel 14.2 erwähnte SELECT-Anfrage angehängt werden. So werden nur die Tupel zurückgegeben, die der Anfrage entsprechen und kein Geheimnis verraten. Diese Tupel können dann unverändert dem Benutzer mitgeteilt werden. Man müsste die Methode cqe(String) der Klasse Application des Prototyps von Sebastian Sonntag dann wie folgt modifizieren:

if Anzahl freie Variablen $\neq 0$
then
if Anfrage $\in L^{o}_{q_nds}$ $\wedge$ Geheimnissprache= $L^{*}_{ps_nds}$ $\wedge$ opt_possible=*true*
then

1. Durch SELECT-Anfrage mit VPD-Funktion in Datenbank gültige Antworten bestimmen, die kein Geheimnis verraten.
2. Antwortmenge ausgeben.

else Dynamischen Verweigerungszensor für offene Anfragen ausführen (noch nicht implementiert)
else Dynamischen Verweigerungszensor für geschlossene Anfragen ausführen

Dieser Ansatz wurde in diesem Prototypen noch nicht umgesetzt, sondern stattdessen der im Folgenden beschriebene Ansatz.

14.2.2 Nutzen des Zensors für geschlossene Anfragen

Es ist ebenfalls möglich, die gefährlichen Tupel durch den Zensor für geschlossene Anfragen zu erkennen. Dazu konstruieren wir für jedes Teiltupel, das uns die SELECT-Anfrage aus Kapitel 14.2 zurückgeliefert hat, die entsprechende geschlossene Anfrage und übergeben sie an den Zensor. Alle Tupel, die der Zensor mit gefährlich beurteilt, werden dann aus der Antwortmenge entfernt, bevor diese ausgegeben wird. Hier sähe die Modifikation der Methode cqe(String) der Klasse Application des Prototyps von Sebastian Sonntag wie folgt aus:

if Anzahl freie Variablen $\neq 0$
then
if Anfrage $\in L^{o}_{q_nds} \wedge$ Geheimnissprache= $L^{*}_{ps_nds} \wedge$ opt_possible=*true*
then

1. Durch SELECT-Anfrage in Datenbank gültige Antworten bestimmen.
2. Für jede Antwort Zensor für geschlossene Anfragen aufrufen, bei Rückgabe von „true“ Antwort aus Antwortmenge entfernen.
3. Antwortmenge ausgeben.

else Dynamischen Verweigerungszensor für offene Anfragen ausführen (noch nicht implementiert)
else Dynamischen Verweigerungszensor für geschlossene Anfragen ausführen

Wie bereits vorher erwähnt gibt es ein Problem mit den optimierten offenen und existentiellen Anfragen. Um dieses Problem zu lösen, wurde eine Fallunterscheidung eingeführt, die in dem Diagramm (14.3) dargestellt ist. Bei der Fallunterscheidung werden die Variablen *boundVar*, *openOpt* und *opt_possible*, sowie der Zensor und die Geheimnissprache bzw. die Anfrage betrachtet.

Stellt der Nutzer nach einer optimierten offenen Anfrage eine existentielle Anfrage, dann soll ein Fenster erscheinen, wo er darauf hingewiesen wird, dass das nur bedingt möglich ist und der Nutzer mit Einschränkungen im weiteren Verlauf zu rechnen hat. Daraufhin hat er dann entweder die Möglichkeit, dass er die existentielle Anfrage doch nicht mehr stellt oder er stellt die Anfrage und muss im weiteren mit Einschränkungen leben. Ausserdem gibt es noch eine dritte Möglichkeit, ein voreingestellter Defaultwert für die Nutzer, die sich nicht entscheiden können. Dieses Fenster wurde in der Methode *cqeclosed* der Klasse *Application* implementiert, wird allerdings derzeit noch nicht angezeigt, da dieser Fall bereits früher abgefangen wird(siehe Abschnitt 14.6.1).

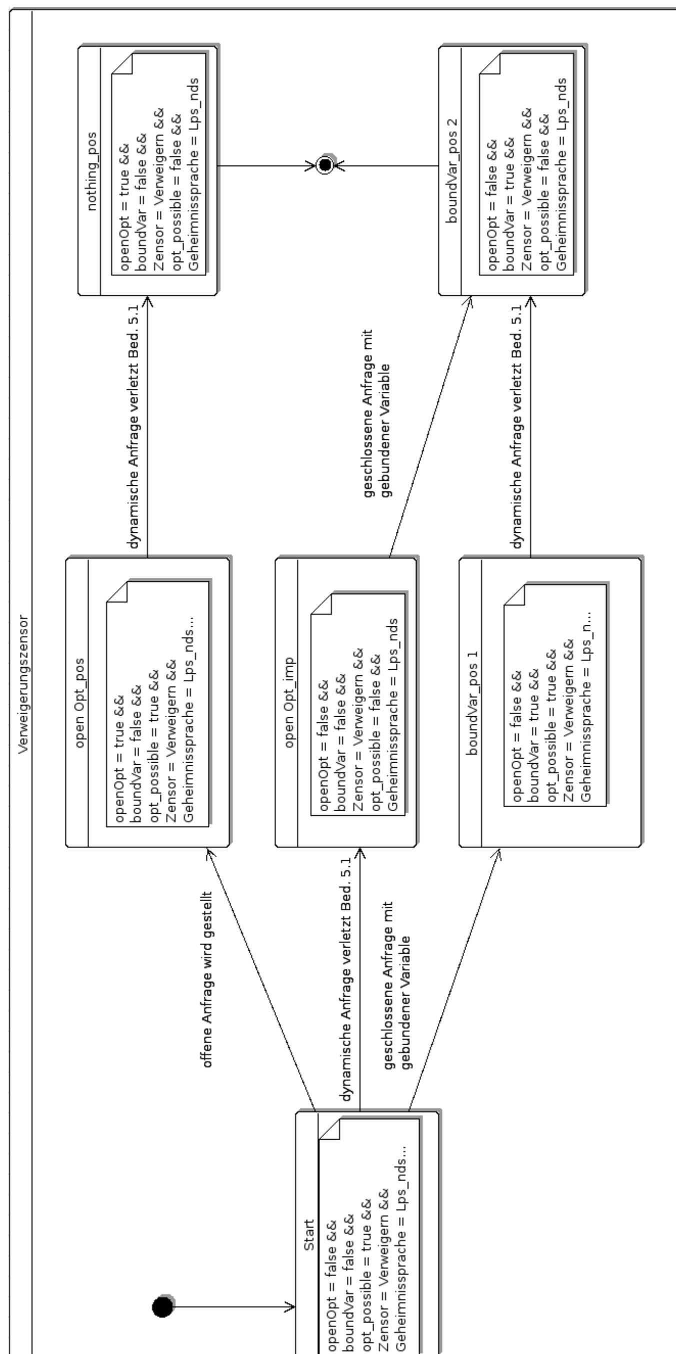


Abbildung 14.1: Endlicher Automat der Fallunterscheidung

14.3 Optimierbare Sprache erkennen

Wir müssen den Prototypen von Sebastian Sonntag nun auch noch dahingehend erweitern, dass er unsere in Kapitel 14.2 definierte Sprache $L^o_{q_nds}$ als solche erkennen kann. Aus Performancegründen sehen wir davon ab, einen zweiten Parser zu schreiben, sondern wollen den vorhandenen, sowie die Methode *cqe(String)* der Klasse *Application* des Prototyps von Sebastian Sonntag erweitern. Wir führen drei *Counter* in der Klasse *queryParser* ein. Der *CounterDisCon* zählt die Vorkommen der Zeichen $\vee$ und $\wedge$, der *CounterNeg* die Vorkommen des Zeichens $\neg$ und der *CounterExists* die Vorkommen des Existensquantors($\exists$). Wir erkennen die Sprache $L^o_{q_nds}$ dann, indem wir die *Counter* entsprechend dem folgenden Algorithmus auswerten. Die Schritte 1 und 2 werden bereits in der Methode cqe(String) umgesetzt. Schritt 3 erfolgt dann jeweils bei der Zustandserkennung in der Methode cqeOpenOpt(Formula) ebenfalls aus der Klasse Application.

Algorithmus 14.3.1:

1. *Parser ausführen*
2. *in SRNF transformieren*
3. ***if*** $(CounterDisCon = 0 \wedge CounterExists = 0 \wedge CounterNeg \mod 2 = 0)$
   then $L^o_{q_nds} = true$

Behauptung:
Alle von Algorithmus 14.3.1 erkannten Formeln liegen in der Sprache $L^o_{q_nds}$.
Beweis:
Eine Anfrage kann eine beliebige Formel der folgenden BNF PG495 sein:

BNF 14.3.1 (BNF der Anfragesprache:)**:**

N = ANF,EXP,R, VAR,KONST,RELSYMB, PARAM, JUNC, KBUCHST,GBUCHST
$T = \{\vee, \wedge, \neg, 9, (,), ;, a, \ldots, z, A, \ldots, Z, \exists\}$[2]

[2] R kommt als Terminal- und als Nichtterminalsymbol vor, der Parser behandelt diese jedoch unterschiedlich, weswegen dies in dem Fall kein Problem ist.

S = ANF
ANF ::= (∃ *VAR)* EXP;*
EXP ::= R / (EXP JUNC EXP) / (¬*EXP)*
JUNC ::= ∨ */* ∧
*R ::= RELSYMB(PARAM(, PARAM)**)
PARAM ::= KONST / VAR
VAR ::= (GBUCHST)+
KONST ::= (KBUCHST)+
RELSYMB ::= (KBUCHST)+
KBUCHST ::= a / . . . / z
GBUCHST ::= A / . . . / Z

Ferner müssen folgende kontextbezogene Nebenbedingungen gelten (die Überprüfung dieser erfolgt im Prototyp bereits in der Methode isValid(TreeNode,LinkedList):Boolean PG495):

1. *Jedes Relationensymbol muss eine Relation beschreiben, die auch tatsächlich in der aktuellen DB-Instanz mit der entsprechenden Stelligkeit existiert.*
2. *Alle vorkommenden Variablen müssen entweder freie Variablen sein oder unter ANF auch gebunden worden sein.*

Es können nun 2 Fälle betrachtet werden:

Sonderfall: $(\mathit{CounterDisCon} = 0 \wedge \mathit{CounterExists} = 0 \wedge \mathit{CounterNeg} = 0) = \mathit{true}$

In diesem speziellen Fall ist offensichtlich, dass der Algorithmus 14.3.1 Formeln mit den Zeichen ∨, ∧, ∃ und ¬ aussortiert. Es bleiben also nur Sprachen mit der auf Seite 219 dargestellten BNF' übrig.

Allgemeiner Fall: $(\mathit{CounterDisCon} = 0 \wedge \mathit{CounterExists} = 0 \wedge \mathit{CounterNeg} \mod 2 = 0) = \mathit{true}$

In diesem Fall sortiert der Algorithmus 14.3.1 Formeln mit den Zeichen ∨, ∧ und ∃ aus, lässt aber das Zeichen ¬ in gerader Anzahl zu. Dies geschieht, um zu prüfen, ob ein Sonderfall der Form $\neg(\neg R(x))$ vorliegt. Das Problem bei diesem Sonderfall ist: Die Counter zählen während des Parsens in Schritt 1, aber Doppelnegationen werden erst während der SRNF-Transformation in Schritt 2 erkannt und aufgelöst.

So könnte eine Formel, die in Schritt 1 noch nicht in der Sprache $L^o_{q_nds}$ lag, nach Schritt 2 sehr wohl in dieser Sprache liegen.
Da die Bedingung CounterDisCon = 0 in Kombination mit der BNF 14.3.1 der Anfragesprache ausschließt, dass Expressions *EXP* Geschwister im Syntaxbaum sein können, können *Negationen* nur vor dem Relationssymbol auftreten. Bei der SRNF-Transformation fallen diese *Negationen* weg, falls ihre Anzahl gerade sein sollte, also liegt diese Anfrage in $L^o_{q_nds}$.
Aufgrund dessen sortiert auch in diesem Fall der Algorithmus 14.3.1 Formeln mit den Zeichen ∨, ∧, ∃ und ¬ aus. Es bleiben also auch in diesem Fall nur Formeln der Sprache mit der nun folgenden BNF' übrig.
BNF' nach Algorithmus [14.3.1]:

BNF 14.3.2:

N = {ANF,EXP,R, VAR,KONST,RELSYMB, PARAM, KBUCHST,GBUCHST}
T = { 9, (,), ; , a, . . . , z, A, . . . ,Z, ∃}
S = ANF
ANF ::= EXP;
EXP ::= R
R ::= RELSYMB(PARAM(, PARAM))*
PARAM ::= KONST | VAR
VAR ::= (GBUCHST)+
KONST ::= (KBUCHST)+
RELSYMB ::= (KBUCHST)+
KBUCHST ::= a | . . . | z
GBUCHST ::= A | . . . | Z

Ferner müssen folgende kontextbezogene Nebenbedingungen gelten (die Überprüfung dieser erfolgt im Prototyp bereits in der Methode isValid(TreeNode,LinkedList):Boolean PG495), sowie durch den Counter CounterExists:

1. *Jedes Relationensymbol muss eine Relation beschreiben, die auch tatsächlich in der aktuellen DB-Instanz mit der entsprechenden Stelligkeit existiert.*
2. *Alle vorkommenden Variablen müssen freie Variablen sein, d.h. sie dürfen unter ANF nicht gebunden worden sein.*

Wir müssen nun noch zeigen $L^{BNF'} \subseteq L^o_{q_nds}$. Betrachten wir hierzu die BNF' genauer. Eine Formel der Sprache $L^{BNF'}$ beginnt immer mit einem Ausdruck *EXP*, der durch ein Relationssymbol *RELSYMB* gefolgt von einem bis endlichen vielen Parametern *PARAM* ersetzt wird. Diese Parameter *PARAM* sind freie Variablen *VAR*, dargestellt durch endlich viele Großbuchstaben *GBUCHST*, oder Konstanten *KONST*, dargestellt durch endlich viele Kleinbuchstaben *KBUCHST*. Diese Reihenfolge von Relationssymbol gefolgt von Groß- und Kleinbuchstaben darf ebenfalls in der Sprache $L^o_{q_nds}$ vorkommen. Alle möglichen Formeln der BNF' sind somit Teil der Sprache $L^o_{q_nds}$. ∎

Mit dem Beweis haben wir gezeigt, dass wir nichts Falsches erkennen. Den Beweis, dass jede Formel aus $L^o_{q_nds}$ von Algorithmus 14.3.1 erkannt wird, werden wir vernachlässigen. Diese Beweisrichtung würde zeigen, dass wir alle optimierbaren Fälle erkennen, worauf wir allerdings keinen Anspruch erheben.

14.4 Implementierung

14.4.1 Variablen

Die bereits in Kapitel 14.2 erwähnten Variablen *boundVar*, *opt_possible* und *openOpt* wurden in der Klasse UserData definiert. In der Klasse SQLInteraction wurden dazugehörig Methoden definiert, die diese Variablen in die Datenbank schreiben bzw. aus der Datenbank auslesen.
Die Counter *counterDisCon*, *counterNeg* und *counterExists* wurden in der Klasse *queryParser* implementiert, so dass beim Parsen der gestellten Anfrage diese Counter das Vorkommen von Konjunktionen, Disjunktionen, Negationen und Existenzquantoren direkt zählen. Da vor der Anfrageauswertung die Anfrage mehrmals geparst wird, wurde zudem eine boole'sche Hilfsvariable *first* implementiert, die beim ersten Parsen *true* ist und danach auf *false* gesetzt wird. Die Counter zählen nur, wenn die Hilfsvariable *true* ist, also nur beim ersten Parsen.

14.4.2 Anfrageauswertung

Der Aufruf der optimierten offenen Anfrageauswertung findet in der Methode *cqe* der Klasse *Application* statt. Bisher gab es offene Anfragen nur für den kombinierten Zensor, bei den anderen beiden Zensoren wurde eine Fehlermeldung ausgegeben, dass offene Anfragen nur mit dem kombinierten Zensor möglich sind. An dieser Stelle wurde der Code nun dahingehend geändert, dass für den Verweigerungszensor ebenfalls eine Abfrage stattfindet und in dieser die neue Methode *cqeOpenOpt* der Klasse *Application* aufgerufen wird (Zeile 10f).

```
1  if (f.getFreeVariables().size() == 0) {
2    LinkedList<String> tmp;
3    tmp = cqeClosed(f);
4    for (int k = 0; k < tmp.size(); k++)
5      answers.add(tmp.get(k));
6  } else {
7    if (userData.getCensor() == 'c') {
8      answers = cqeOpen(f);
9    }
10   else if (userData.getCensor() == 'r'){
11     answers = cqeOpenOpt(f);
12   }
13 else
14   throw new Exception("Freie Variablen nicht mit
15   Luegenzensor moeglich");
```

In der Methode *cqeOpenOpt* findet nun die bereits vorher erwähnte Fallunterscheidung statt. Der einzige Fall, der bisher implementiert wurde, ist der Fall *open Opt_pos* (siehe 14.3). Bei den anderen Fällen wird vorerst nur eine Fehlermeldung ausgegeben.

Die Vorgehensweise des Falles *open Opt_pos* ist dabei die, die bereits im Algorithmus im Kapitel 14.2.2 erwähnt wurde. Für die Umsetzung werden dabei insbesondere drei neue Methoden gebraucht. Deren Funktionalität soll im Folgenden kurz erklärt werden. Ursprünglich sollte nach der Auswertung der Anfrage auch ein entsprechender Logeintrag generiert werden, der für die optimierte offene Anfragenauswertung zwar nicht gebraucht wird, der aber gegebenfalls bei einer späteren dynamischen Anfrageauswertung gebraucht wird. Es wurden bereits Ansätze zur Generierung des Logeintrags implementiert, allerdings sind diese noch nicht funktionsfähig (siehe 14.6.1).

Die wichtigsten drei neuen Methoden sind die Methode *TreeNodeForSQL* in der Klasse *Formula*, sowie die Methoden *getTupel* und *getTupelfromPotsec* in der Klasse *SQLInteraction*.
Die Funktionalität der Methoden ist die Folgende:

- **TreeNodeForSQL**: Erstellt ein Stringarray aus dem Syntaxbaum der Formel. Durchläuft den Syntaxbaum und speichert an der ersten Stelle des Arrays den Relationennamen der Anfrage, an den weiteren Stellen werden die Variablen und Konstanten der Anfrage gespeichert. Damit soll die Struktur der Anfrage dargestellt werden, um daraus dann später z.B. in den Methoden *getTupel* und *getTupelfromPotsec* die WHERE-Klauseln der SELECT-Anfragen zu basteln. Um Variablen und Konstanten in dem Array unterscheiden zu können, wird vor die Variablen der String *yyy* geschrieben, auf den später bei if-Abfragen hin geprüft wird.

- **getTupel**: Erstellt eine LinkedList vom Typ String[], in der die Tupel gespeichert werden, die in der Datenbank gelten. Diese Tupel werden durch eine SELECT-Anfrage an die Datenbank geholt, die WHERE-Klausel der Anfrage wird mit den Konstanten der Anfrage gebildet. Sind in der Anfrage keine Konstanten, wird eine SELECT-Anfrage ohne WHERE-Klausel gestellt.

- **getTupelfromPotsec**: Erstellt eine LinkedList vom Typ String[], in der die Tupel gespeichert werden, die bezüglich der Anfrage und des Pot_sec geheim gehalten werden müssen. Die SELECT-Anfrage wird genauso aufgebaut wie bei der Methode *getTupel*. Die Anfrage wird an spezielle Geheimnistabellen bei der optimierten offenen Anfragenauswertung gestellt (siehe Abschnitt 14.4.3 Klassifikationsinstanz). Das Ergebnis dieser Methode wird insbesondere für das Erstellen des Logeintrags benötigt, damit dargestellt werden kann, für welche Tupel dem Nutzer nicht verraten werden kann, ob sie in der Datenbank gelten oder nicht gelten.

Diese Methoden werden nun in der Methode *cqeOpenOpt* benutzt, um den Algorithmus mit dem Zensors für geschlossene Anfragen (Abschnitt 14.2.2) umzusetzen. Dazu wird zunächst mal mit *TreeNodeForSQL* ein Stringarray von der Struktur der Anfrage erstellt. Danach werden aus der Datenbank mit *getTupel* die gültigen Tupel gemäß der Anfrage gefiltert und im nächsten Schritt an den Zensor weitergegeben.

Die Tupel, für die der Zensor kein *MUM* zurück gibt, werden dann der Liste der Antworten hinzugefügt. Als letztes Element der Antwortliste wird schlussendlich die Angabe des Optimierungsfalls "*Open Opt - (Stat. IK)*" hinzugefügt, der dann im Antwortfenster dem Nutzer auch angezeigt wird. Diese Liste wird dann von der Methode zurückgegeben und gemäß dem bereits vorhandenem Programm für die Ausgabe weiterverarbeitet.
In dem folgenden Sequenzdiagramm werden die wichtigsten Schritte dieses Prozesses nochmal aufgezeigt.

14.4.3 Klassifikationsinstanz

Da es bei der Implementierung zu Konflikten durch die Veränderung alter Methoden aus dem Prototyp von Sebastian Sonntag für die Umsetzung von freien Variablen in der Geheimnissprache kam, mussten für die optimierte offene Anfrageauswertung ein paar alte Methoden kopiert und umbenannt werden, um so die Funktionalität der optimierten offenen Anfrageauswertung zu gewährleisten.
Im Speziellen sind dies die neuen Methoden *addtoPotsecRelOpenOpt* und *deletePot_secRelOpenOpt* in der Klasse *UserData*. Diese werden zur Erzeugung und zum Löschen von Klassifikationsinstanzen des Pot_sec gebraucht. Desweiteren war es auch noch notwendig in der Datenbank für diesen Fall neue Tabellen anzulegen, in der die Instanzen für die Auswertung zeitweise gespeichert werden können. Der Tabellenname setzt sich aus dem Relationennamen und der Endung _PS2 zusammen.

Beispiel 14.4.1:
Für die Relation ARMBRUCH würde die entsprechende Tabelle z.B. ARMBRUCH_PS2 lauten.

In den neuen Methoden *addtoPotsecRelOpenOpt* und *deletePot_secRelOpenOpt* wurden ausserdem die SQL-Anfragen entsprechend der neuen Tabellen angepasst.

14.4.4 Zensor

Da bei der Implementierung ebenfalls Probleme durch Änderungen bezüglich des Entwurfs freier Variablen in der Geheimnissprache in der Klasse des Verweigerungszensors gab, mussten deswegen auch in den Klassen *Censor* und *RefusalCensor* alte

Abbildung 14.2: Sequenzdiagramm

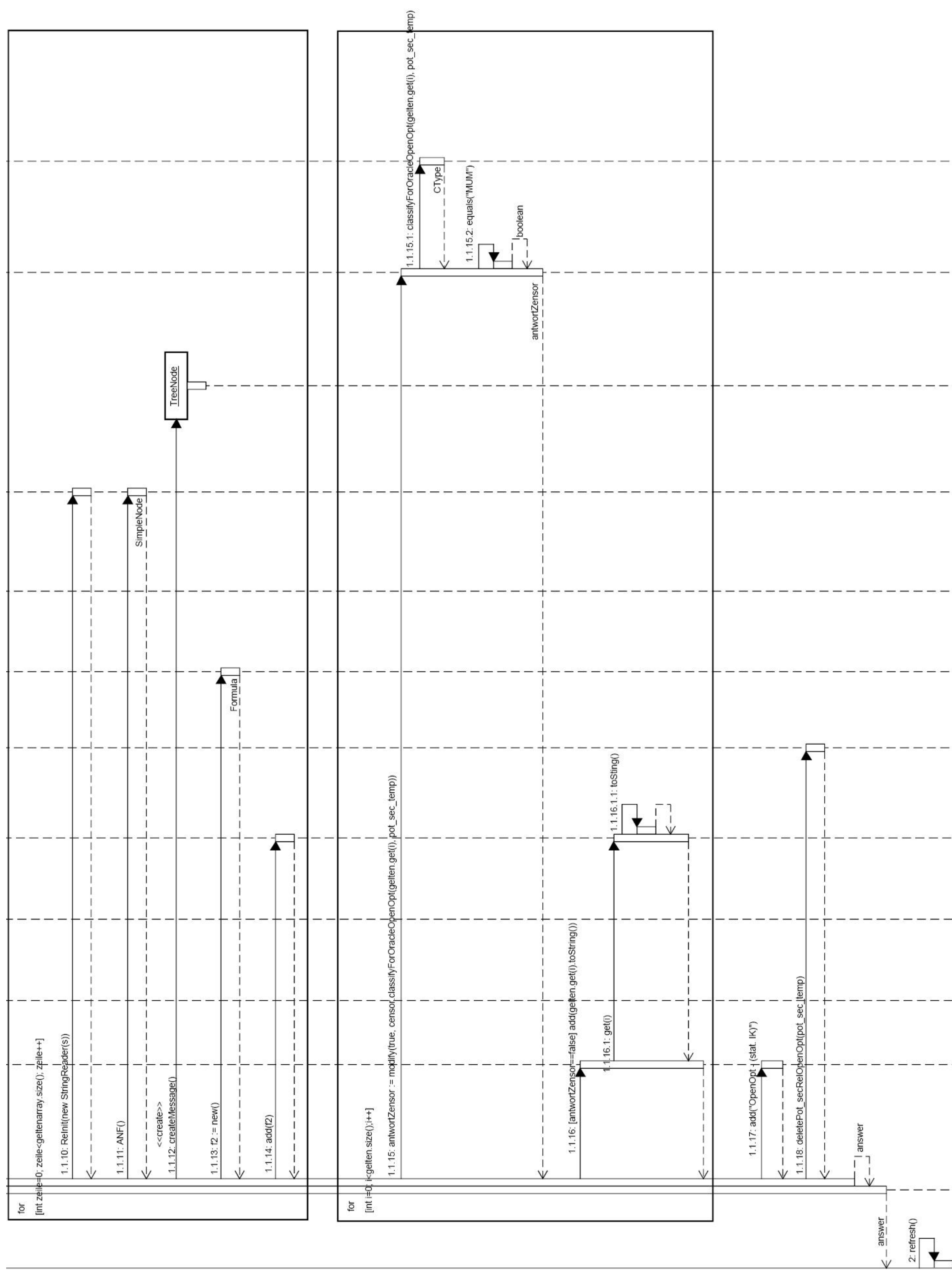


Abbildung 14.3: Sequenzdiagramm(weiter)

Methoden kopiert und mit einem neuem Namen versehen werden, um auch hier die Funktionalität der optimierten offenen Anfrageauswertung zu garantieren. In der Klasse *Censor* wurden zu dem Zweck die beiden abstrakten Methoden *isImpliedforOracleOpenOpt* und *classifyforOracleOpenOpt* hinzugefügt. Diese Methoden wurden dann in der Klasse *RefusalCensor* implementiert.
Damit zusammenhängend musste in der Klasse *Formula* eine aus *isImpliedforOracleOpenOpt* aufgerufene Methode ebenfalls von der alten Methode kopiert und als die neue Methode *getConstantsOpenOpt* implementiert werden, um dann in *isImpliedforOracleOpenOpt* aufgerufen zu werden.

14.5 Testfälle

Diese Tests dienen dem Test der Gesamtfunktionalität der implementierten Funktionen und wurden im Prototypen selbst durchgeführt.
Bei den Tests wurde auf den Tabellen Armbruch, Krankheit und Reha gearbeitet. Beim ersten Teil wurden ein- und mehrstellige Relationen mit einer oder mehreren freien Variablen getestet. Im zweiten Teil ging es darum die Fallunterscheidung des Automaten zu testen (siehe Diagramm 14.3).
Die Belegung der Tabellen bei den Tests war die Folgende:

ARMBRUCH	NAME
	jude
	lena
	wolfgang
	gustav
	dumpfbacke
	ndula
	hans

KRANKHEIT	NAME	KRANKHEIT
	yu	cold
	alfred	cold
	fred	lykanthropie

REHA	NAME	ORT	DAUER	KRANKHEIT
	fred	dortmund	lange	beinbruch
	horst	bochum	mittel	armbruch
	frank	dortmund	kurz	death
	hans	dortmund	kurz	armbruch

14.5.1 Offene optimierte Anfragen

Relation ARMBRUCH
Testfall a: Einstellige Relation, eine freie Variable
pot_sec : –
log : –
query : armbruch(X)
Gewünschte Ausgabe : armbruch(jude), armbruch(lena), armbruch(wolfgang), armbruch(gustav), armbruch(hans), armbruch(dumpfbacke), armbruch(ndula)
Ausgabe : wie gewünscht

Testfall b: Einstellige Relation,eine freie Variable
pot_sec : armbruch(hans),armbruch(lena)
log : –
query : armbruch(X)
Gewünschte Ausgabe : armbruch(jude), armbruch(wolfgang), armbruch(gustav), armbruch(dumpfbacke), armbruch(ndula)
Ausgabe : wie gewünscht

Relation KRANKHEIT
Testfall c: Zweistellige Relation, eine freie Variable
pot_sec : krankheit(yu,cold)
log : –
query : krankheit(X,cold)
Gewünschte Ausgabe : krankheit(alfred,cold)

Ausgabe : krankheit(alfred,cold)

Testfall d: Zweistellige Relation, zwei freie Variablen
pot_sec : krankheit(yu,cold)
log : –
query : krankheit(X,Y)
Gewünschte Ausgabe : krankheit(alfred,cold), krankheit(fred,lykanthropie)
Ausgabe : krankheit(alfred,cold), krankheit(fred,lykanthropie)

<u>Relation REHA</u>
Testfall e: Vierstellige Relation, eine freie Variable
pot_sec : reha(frank,dortmund,kurz,death)
log : –
query : reha(X,dortmund,kurz,death)
Gewünschte Ausgabe : –
Ausgabe : –

Testfall f: Vierstellige Relation, eine freie Variable
pot_sec : reha(frank,dortmund,kurz,death)
log : –
query : reha(X,dortmund,kurz,armbruch)
Gewünschte Ausgabe : reha(hans,dortmund,kurz,armbruch)
Ausgabe : reha(hans,dortmund,kurz,armbruch)

Testfall g: Vierstellige Relation, drei freie Variable
pot_sec : reha(frank,dortmund,kurz,death)
log : –
query : reha(X,dortmund,Y,Z)
Gewünschte Ausgabe : reha(fred,dortmund,lange,beinbruch),
reha(hans,dortmund,kurz,armbruch)
Ausgabe : (reha(fred,dortmund,lange,beinbruch), reha(hans,dortmund,kurz,armbruch))

Testfall g2: Vierstellige Relation, drei freie Variablen, Existenzquantor im Pot_sec

pot_sec : exists X reha(X,dortmund,kurz,death)
log : –
query : reha(X,dortmund,Y,Z)
Gewünschte Ausgabe : reha(fred,dortmund,lange,beinbruch),
reha(hans,dortmund,kurz,armbruch)
Ausgabe : reha(fred,dortmund,lange,beinbruch), reha(hans,dortmund,kurz,armbruch)

Testfall h: Vierstellige Relation, vier freie Variable
pot_sec : reha(frank,dortmund,kurz,death)
log : –
query : reha(W,X,Y,Z)
Gewünschte Ausgabe : reha(fred,dortmund,lange,beinbruch),
reha(horst,bochum,mittel,armbruch), reha(hans,dortmund,kurz,armbruch)
Ausgabe : wie gewünscht

14.5.2 Fallunterscheidung

Testfall i: Offene Anfrage
Vorbedingungen : openOpt=false oder true, boundVar=false, opt_possible=true
log : –
pot_sec : –
query : armbruch(X)
Gewünschte Ausgabe : siehe Testfall a
Gewünschte Nachbedingung : openOpt=true, boundVar=false, opt_possible=true
Ausgabe : siehe Testfall a
Nachbedingungen : openOpt=true, boundVar=false, opt_possible=true

Testfall j: Existenzquantor nach einer offenen Anfrage
Vorbedingungen : openOpt=true, boundVar=false, opt_possible=true
log : –
pot_sec : –
query : exists X armbruch(X)

Gewünschte Ausgabe : Fehlermeldung(*Es gab bereits eine offene Anfrage, daher dürfen Variablen nur noch frei in der Anfrage vorkommen*)
Gewünschte Nachbedingung : unverändert
Ausgabe : Fehlermeldung(*Es gab bereits eine offene Anfrage, daher dürfen Variablen nur noch frei in der Anfrage vorkommen*)
Nachbedingungen : unverändert

Testfall k: nothing_pos
Vorbedingungen : openOpt=true, boundVar=false, opt_possible=true
log : –
pot_sec : –
query : (armbruch(hans) or armbruch(lena))
Gewünschte Ausgabe : armbruch(hans) or armbruch(lena)
Gewünschte Nachbedingungen : openOpt=true, boundVar=false, opt_possible=false
Ausgabe : armbruch(hans) or armbruch(lena)
Nachbedingungen : openOpt=true, boundVar=false, opt_possible=false

Testfall l: openOpt_imp
Vorbedingungen : openOpt=false, boundVar=false, opt_possible=true
log : –
pot_sec : –
query : (armbruch(hans) or armbruch(lena))
Gewünschte Ausgabe : (armbruch(hans) or armbruch(lena))
Gewünschte Nachbedingungen : openOpt=false, boundVar=false, opt_possible=false
Ausgabe : (armbruch(hans) or armbruch(lena))
Nachbedingungen : openOpt=false, boundVar=false, opt_possible=false

Testfall m: boundVar_pos2
Vorbedingungen : openOpt=false, boundVar=false, opt_possible=false
log : (armbruch(hans) or armbruch(lena))
pot_sec : –
query : exists X armbruch(X)
Gewünschte Ausgabe : exists X armbruch(X)
Gewünschte Nachbedingungen : openOpt=false, boundVar=true, opt_possible=false
Ausgabe : exists X armbruch(X)

Nachbedingungen : openOpt=false, boundVar=true, opt_possible=false

Testfall n: boundVar_pos1
Vorbedingungen : openOpt=false, boundVar=false, opt_possible=true (Initialzustand)
log : –
pot_sec : –
query : exists X armbruch(X)
Gewünschte Ausgabe : exists X armbruch(X)
Gewünschte Nachbedingungen : openOpt=false, boundVar=true, opt_possible=true
Ausgabe : exists X armbruch(X)
Nachbedingungen : openOpt=false, boundVar=true, opt_possible=true

Testfall o: boundVar_pos2 (aus boundVar_pos1)
Vorbedingungen : openOpt=false, boundVar=true, opt_possible=true
log : –
pot_sec : –
query : (armbruch(hans) or armbruch(lena))
Gewünschte Ausgabe : (armbruch(hans) or armbruch(lena))
Gewünschte Nachbedingungen : openOpt=false, boundVar=true, opt_possible=false
Ausgabe : (armbruch(hans) or armbruch(lena))
Nachbedingungen : openOpt=false, boundVar=true, opt_possible=false

14.6 Weiterführende Aufgaben

Aufgaben, die nicht mehr im Rahmen der PG536 erledigt werden konnten, aber noch nötige Erweiterungen in Zusammenhang mit der Auswertung optimierbarer offener Anfragen sind, werden in diesem Kapitel thematisiert. Jedes der nachfolgenden Unterkapitel hat eine dieser Aufgaben zum Thema.

14.6.1 Fehlende Funktionalität in einem Zustand

In dem Zustandsdiagramm 14.3 ist der Zustand *open Opt_pos* aufgeführt. In diesem Zustand können aufgrund des Problems mit den Existenzquantoren, das den Schutz von Geheimnissen gefährdet (siehe Abschnitt 14.2) keine Anfragen mehr gebundene Variablen enthalten. Es wäre aber möglich gebundene Variablen in geschlossenen Anfragen zuzulassen, wenn man:

1. Diese Anfragen dynamisch bearbeiten würde.
2. Das neu hinzugekommene Benutzerwissen, durch vorangegangene offene Anfragen, im Benutzerlog speichern würde.

Der Fehler tritt nämlich in dem Fall einer offenen Anfrage gefolgt von einer Anfrage mit mind. einer gebundenen Variablen nur auf, wenn bei der Anfrage mit gebundener Variable das Benutzerlog nicht berücksichtigt wird oder dieses nicht vollständig ist.
Wir haben die Methode cqeOpen(Formula) dahingehend modifiziert, dass geschlossene Anfragen mit Existenzquantoren, die nach einer offenen Anfrage erfolgen, ein Dialogfeld öffnen. Dieses bietet die Möglichkeiten, die Anfrage dynamisch bearbeiten zu lassen oder die Anfrage umzuformulieren.
Leider war es uns nicht möglich, das durch optimierte offene Anfragen gewonnene Benutzerwissen korrekt zu speichern. Es wird im Rahmen der Methode cqeOpenOpt(Formula) der Klasse Application lediglich ein String erzeugt, in dem die Formel gespeichert ist, die das neu hinzugekommene Benutzerwissen korrekt ausdrückt. Leider kommt es aber zu einer Fehlermeldung, bei dem Versuch diesen in der Oracle Datenbank zu speichern, so dass nach offen optimierten Anfragen kein neues Benutzerwissen erzeugt wird.
Deswegen haben wir die Methode cqeOpen(Formula) der Klasse Application dahingehend abgeändert, dass geschlossene Anfragen mit Existenzquantor per Fehlermeldung unterbunden werden. Offene Anfragen mit Existenzquantor müssen zur Zeit nicht berücksichtigt werden, da nur optimierte offene Anfragen behandelt werden, welche per Definition keine Existenzquantoren enthalten (BNF 14.3.2). Offene Anfragen, die nicht optimiert behandelt werden können, müssen nicht berücksichtigt werden, da diese nur mit einem anderen Zensortypen vorkommen können.

Das Unterbinden jeglicher geschlossener Anfrage mit Existenzquantor kann rückgängig gemacht werden, sobald das neu gewonnene Benutzerwissen einer optimierten offenen Anfrage korrekt gespeichert wird, weswegen dies eine wichtige zukünftige Aufgabe darstellt.

14.6.2 Fehlender Zensor

Wie man dem Zustandsdiagramm 14.3 entnehmen kann, gilt das ganze Diagramm nur für den Verweigerungszensor, aus dem einfachen Grund, dass ein statischer Ansatz bei offenen sowie geschlossenen Anfragen bisher nur bei diesem Zensortypen möglich ist. Im aktuellen Prototypen (PG536) gibt es allerdings keinen Verweigerungszensor für offene Anfragen, die nicht optimiert behandelt werden können.
Die Implementierung eines dynamischen Ansatzes mit Verweigerungszensor für offene Anfragen würde folgendes erreichen:

1. Anfragen mit freien Variablen, die die zusätzlichen Bedingungen einer optimierten Anfrageauswertung, die in Kapitel 14.2 beschrieben sind, nicht erfüllen, könnten bearbeitet werden.
2. Analog zu dem Unterkapitel 14.6.1 könnte man für die Zustände open Opt_imp, boundVar_pos1 und boundVar_pos2 offene Anfragen zulassen, diese müssten dann aber von dem noch zu implementierenden dynamischen Ansatz der offenen Anfrageauswertung bearbeitet werden. Das Benutzerwissen wird bei geschlossenen Anfragen in jedem Fall schon richtig gespeichert.

Wie man sieht, wird durch die Implementierung eines dynamischen Ansatzes mit Verweigerungszensor für offene Anfragen die Menge der auswertbaren Anfragen weiter steigen, sowie eine Symmetrie im Zustandsdiagramm hergestellt, was der Verständlichkeit zu Gute kommt.

14.6.3 Persistente Klassifikationsinstanz

Nachdem wir uns bei den vorangegangenen Unterkapiteln um eine Steigerung der Funktionalität bemüht haben, wollen wir uns nun ein Performanceproblem ansehen.

Im aktuellen Prototyp (PG536) werden im statischen Fall, sowohl bei offenen als auch bei geschlossenen Anfragen, Klassifikationsinstanzen wie folgt verwaltet:

1. Für jede Datentabelle wird bereits bei der Schemaerzeugung eine Klassifikationsinstanztabelle erzeugt.
2. Während der Laufzeit werden für jede Anfrage die entsprechenden Klassifikationsinstanztabellen bzgl. der Geheimnismenge des aktuellen Benutzers gefüllt und nach Ausgabe der Antwort wieder geleert.

Nun ist dieses Vorgehen nicht sonderlich effizient. Es wäre effizienter, wenn die Klassifikationsinstanztabellen eines Benutzers einmal gefüllt werden würden und nur noch angepasst würden, wenn sich die Geheimnismenge des Benutzers ändert.
Allerdings bräuchte man dann pro Datentabelle pro Benutzer eine entsprechende Klassifikationsinstanztabelle. Um das wiederum verwalten zu können, müssten die Klassifikationsinstanztabellen eines Benutzers bei der Erzeugung eines neuen Benutzers auf Oracleebene angelegt werden. Man kann aber nicht einfach für jede Tabelle des entsprechenden Oracleschemas eine Klassifikationsinstanztabelle erzeugen, da es auch reine Verwaltungstabellen gibt, wie z.B. die Benutzertabelle, für die man keine Klassifikationsinstanz benötigt. Leider kann die Entscheidung, für welche Tabellen man eine Klassifikationsinstanz benötigt, noch nicht automatisiert getroffen werden, da dies bisher nur aus dem Namen der Tabelle ersichtlich ist.
Dieses Problem zu lösen ist ein erster Schritt, die Klassifikationsinstanzen persistent zu speichern und somit die statische Anfrageauswertung effizienter zu gestalten.

Kapitel 15

Entwurf freier Variablen in der Geheimnissprache

Wir haben in der Datenbank eine Tabelle von potentiellen Geheimnissen, die folgenderweise aussieht.

$$
\begin{aligned}
pot_sec = \{ \quad & (\exists X_{bal})db(BoS, 100001, Smith, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100002, Johnson, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100003, Williams, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100004, Jones, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100005, Brown, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100006, Davis, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100007, Miller, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100008, Wilson, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100009, Moore, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100010, Taylor, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100011, Anderson, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100012, Thomas, X_{bal}),\\
& (\exists X_{bal})db(BoS, 100013, Jackson, X_{bal}),
\end{aligned}
$$

Abbildung 15.1: Eine Geheimnismenge mit unendlichen Einträgen [Loc09]

Das Bild zeigt, dass die Datenbank unendlich viele Einträge enthalten kann, falls der Admin alle Verbindungen zwischen Kontonummer und Kontoinhaber geheim halten will. Um dieses Problem zu lösen, wollen wir freie Variablen wie folgt einsetzen:

$$pot_sec = \{(\exists X_{bal}) db(BoS, X^{free}_{K.Nr}, X^{free}_{K.Inhaber}, X_{bal})\}$$

Durch das Einbetten von freien Variablen in der Geheimnissprache können in der Menge der potentiellen Geheimnisse *pot_sec* redundante Informationen hervorgerufen werden. Es ist von Vorteil, dass diese redundanten Informationen aus *pot_sec* entfernt werden, damit *pot_sec* redundanzfrei und minimal bleibt. Die Minimalität heißt hier, dass kein einziges Element von der Menge entfernt werden kann, ohne dass die ganze Menge ihre semantische Bedeutung verliert. Die Vorteile von Minimalität sind

- Je kleiner die *pot_sec* Menge ist, desto weniger Speicherplatz wird benötigt.
- Da Kontrollierte Anfrageauswertung jedes Element von *pot_sec* in Betracht zieht, wird durch die Minimierung der Menge *pot_sec* die Laufzeit der Kontrollierten Anfrageauswertung verkürzt.

Wie wir bereits erwähnt haben, kann unser Prototyp im dynamischen und statischen Modus arbeiten.

15.1 Dynamischer Modus

15.1.1 Theorembeweiser

Jede Anfrage an das System wird zur Laufzeit auf ihre Gefährlichkeit hin überprüft. Dies geschieht durch Ermittlung von Inferenzen. Für den dynamischen Ansatz der Inferenzkontrolle kann dazu ein Theorembeweiser die zu überprüfenden Implikationen auf ihren Wahrheitswert untersuchen. Als Theorembeweiser ist Prover9 zum Einsatz gekommen.

Die Schnittstelle des Programms zu dem Prover9 besteht aus der Interface-Klasse „TpInterface“ (siehe Abbildung 15.2). Übergeben werden ihr das Benutzerwissen, die Anfrage, und die Menge der potentiellen Geheimnisse.

Ob die Anfrage als gefährlich erkannt wird, hängt von dem Rückgabewert der Methode *isImplied* ab, denn mit dieser Methode wird überprüft, ob beispielsweise bei

TpInterface ≪ *interface* ≫
public boolean isImplied (LinkedList log, Formula query, LinkedList *pot_sec*) public String translate (LinkedList log, Formula query) public boolean prove()

Abbildung 15.2: Interface-Klasse des Theorembeweisers

dem Verweigerungszensor eines der Geheimnisse oder seine Negation durch das Benutzerwissen und die Anfrage impliziert wird. Mit anderen Worten, wenn der Rückgabewert der Methode *isImplied true* ist, ist die Anfrage gefährlich.

Prover9 arbeitet mit Input-Dateien, welche folgende Gestalt annehmen müssen:

formulas(sos).
Benutzerwissen (log) wird zeilenweise eingefügt.
In der letzten Zeile steht die Anfrage.
end_of_list.

formulas(goals).
Hier werden die übergebenen potentiellen Geheimnisse einzeln eingefügt.
D.h. jedes Element der potentiellen Geheimnisse bildet zusammen mit dem Benutzerwissen und der Anfrage eine Input-Datei.
end_of_list.

Kurze Erläuterung:

- Die Formeln zwischen *formulas(sos).* und dem ersten *end_of_list.* werden konjunktiv verknüpft und bilden die Prämisse.
- Die Formel zwischen *formulas(goals).* und dem letzten *end_of_list.* ist die Konklusion, die überprüft wird.

Beispiel 15.1.1:
Die gesamte Prover9 Input-Datei sieht beispielsweise wie folgt aus:

formulas(sos).

% Das Benutzerwissen:
$armbruch(Hans).$
$beinbruch(Peter).$
%Die Anfrage:
exists x $(armbruch(x)\ \&\ beinbruch(x)).$
end_of_list.

formulas(goals).
%Ein Geheimnis:
$beinbruch(Marc).$
end_of_list.

Prover9 führt einen Unerfüllbarkeitstest aus. Dabei wird überprüft, ob aus dem Benutzerwissen und der Anfrage das Geheimnis geschlussfolgert wird.

15.1.2 Freie Variablen in der Geheimnissprache

Definition 15.1.1 (freie Variablen in Geheimnissprache [Loc09])**:**
Ein Element von $\mathcal{L}_{ps}^{f}$ wird mit $\Psi(V)$ bezeichnet, wobei $V = (X_1, X_2, \ldots, X_l)$ ein Vektor aller freien Variablen ist, die in $\Psi(V)$ vorkommen.
Ein potentielles Geheimnis mit freien Variablen $\Psi(V) \in \mathcal{L}_{ps}^{f}$ wird durch Substitution der freien Variablen V mit allen Konstanten zu einer (unendlichen) Menge $ex(\Psi(V)) \subset \mathcal{L}_{ps}$ expandiert.
Ein Element von $ex(\Psi(V))$ wird als $\Psi(c)$ bezeichnet, wobei c ein Vektor von Konstanten ist. Die Expansion der Politik $pot_sec \subset \mathcal{L}_{ps}^{f}$ ist definiert als $ex(pot_sec) = \bigcup_{\Psi(V) \in pot_sec} ex(\Psi(V)) \subset \mathcal{L}_{ps}$.

Mit dieser Definition hat der Zensor für die Anfrageauswertung die Form:

- Im statischen Modus:
  $censor_{stat}^{f}(pot_sec, \Phi)$:= Es existiert $\Psi(c) \mid \Psi(c) \in ex(pot_sec)$ und $\Phi \models \Psi(c)$
- Im dynamischen Modus: ($logfile$ berücksichtigen)
  $censor_{dyn}^{f}(pot_sec, log, \Phi)$:= Es existiert $\Psi(c) \mid \Psi(c) \in ex(pot_sec)$ und $(log \cup \{\Phi\} \models \Psi(c)$ oder $log \cup \{\neg\Phi\} \models \Psi(c))$

Dabei wird Ψ durch $\Psi(c)$ bzw. $\Psi \in pot_sec$ durch $\Psi(c) \in ex(pot_sec)$ ersetzt.

Bemerkung:
Die expandierte Menge $ex(pot_sec)$ kann unendlich sein. Leider gibt es bis jetzt keinen sicher terminierenden Zensor, der in der Lage ist, eine unendliche Menge $ex(pot_sec)$ zu prüfen. Wir müssen einen alternativen Zensor entwerfen, der terminiert.
Jan-Hendrik Lochner hat in seiner Publikation [Loc09] einen alternativen Zensor für den statischen Modus aufgestellt, der mit der unendlichen Menge pot_sec arbeiten kann. Dabei bezeichnet $\chi[A_i]$ die Instantiierung des Attributes A_i in χ.
z.B. $\Psi(X_f) \equiv (\exists X_b)R(a, X_b, X_f)$, dann $\Psi(X_f)[A_1] = a$ und $\Psi(X_f)[A_2] = X_b$.

$censor^{f,alt}_{stat}(pot_sec, \Phi) :=$ (es existiert $\Psi(V) \in pot_sec$, so dass für alle $A \in \mathcal{U}$:
wenn $\Psi(V)[A] \in Const$, dann $\Phi[A] = \Psi(V)[A]$, und
wenn $\Psi(V)[A]$ freie Variable, dann $\Phi[A] \in Const$)

15.1.3 Minimierung der Menge *pot_sec* mittels Prover9

Laut `http://www.cs.unm.edu/~mccune/mace4/` können alle Prover9-Versionen ab Juli 2006 mit freien Variablen umgehen. Prover9 unterscheidet zwischen Konstanten und freien Variablen dadurch, dass freie Variablen mit klein geschriebenen Buchstaben von “u“ bis “z“ anfangen.

Wie oben erwähnt, kann die Menge der potentielle Geheimnisse in sich, durch Einbetten von freien Variablen, redundante Informationen beherbergen. Die Minimalisierung der Menge pot_sec der potentiellen Geheimnisse zu einer Menge pot_sec' fr̈ die drei Zensorentypen kann aus [PG408] *(S. 177 - 179)* entnommen werden.

15.1.4 Problemdarstellung

Leider kann die Minimalität der Menge pot_sec mithilfe des Theorembeweisers Prover9 nicht erreicht werden.

Zum Beispiel enthält die Menge pot_sec bereits $\psi_1 \equiv \exists x.db(Sparkasse, y, z, x)$. Möchte der Admin das Geheimnis $\psi_2 \equiv \exists x.db(Sparkasse, Hans, 251, x)$ in die Men-

ge *pot_sec* einfügen, so wird es überprüft, ob das hinzuzufügende Geheimnis ψ_2 von dem bereits enthaltenen Geheimnis ψ_2 logisch impliziert wird.

Da die logische Folgerung mit freien Variablen nicht definiert ist, und Prover9 die Implikation von freien Variablen nach der Konvention löst, in der freie Variablen durch Allquantifizierte ersetzt werden, was wiederum bei einem unendlichen Universum zu einem Nicht-Terminieren führen kann, ist Prover9 auf unser Problem nicht anwendbar. Eine approximative Lösung wird im Folgenden besprochen.

15.2 Approximation freier Variablen

Die im vorangegangenen Kapitel angesprochene Approximation bildet der existentielle Abschluss, der wie folgt definiert sei:

Definition 15.2.1:
Für $\Psi[X] \in L_{ps}^{f}$ *mit freien Variablen* $X = (X_1, ..., X_n)$
sei $(\exists X)\Psi[X] \equiv (\exists X_1)...(\exists X_n)\Psi[X]$ *der existentielle Abschluss.*

In dem nachfolgenden Theorem 15.2.1 beschäftigen wir uns mit der Frage, ob alle Geheimnisse, die mit einem *pot_sec* mit freien Variablen geschützt werden, auch durch dessen existentiellen Abschluss geschützt werden.

Theorem 15.2.1:
Wenn $log \not\models (\exists X)\Psi[X]$
dann gilt für alle Konstantenzeichen $c_1, ..., c_n \in Const$
$log \not\models \Psi(c_1, ..., c_n)$.

Beweis durch Kontraposition Wenn Konstantenzeichen $c_1, ..., c_n \in Const$ existieren mit
$log \models \Psi(c_1, ..., c_n)$, dann gilt
$log \models (\exists X)\Psi[X]$.
Denn: Setze $c_1, ..., c_n$ als Belegung von $X = (X_1, ..., X_n)$ $\boxdot$

Somit gilt, dass alle Geheimnisse, die durch ein *pot_sec* mit freien Variablen geschützt werden, werden auch durch dessen Approximation geschützt.

15.2.1 Implementierung

Die Zeilen 4-12 wurden in die Methode **addToPot_sec(String loginKey, String expr)** in der Klasse Application zugefügt. In Zeile 4 wird die Methode **free2BoundVar(expr)** aus der Klasse Translater aufgerufen, wobei die Klasse Translater von dieser PG536 erstellt wurde und für die Übersetzung von Formeln zuständig ist. Dies führt dazu, dass falls das Geheimnis *expr* freie Variablen enthält, es durch seinen existentiellen Abschluss ersetzt und erst dann im *pot_sec* gespeichert wird.

```
1  public void addToPot_sec(String loginKey, String expr) throws
       Exception {
2      try {
3        /***********************************************/
4        String expr_free2bound = Translater.free2BoundVar(expr);
5        if ( ! expr_free2bound.equals(expr) ) {
6          /*
7           * expr contains free Variables
8           */
9          System.err.println("Warnung: eingegebenes Geheimnis
               enthaelt freie Variablen! " +
10         " Von nun an werden die Anfragen in dynamischem Modus
               aproximiert beantwortet.");
11         //TODO die Neue Zustand speichern
12       }//  @autor adalat,  10.12.2009
13       /***********************************************/
14
15     ...
16
17     } catch (Exception e) {
18       unlock();
19       throw new Exception(e.getMessage());
20     }
21 }
```

Wir zeigen einige Abschnitte aus der Methode **free2BoundVar(expr)**. Diese Methode erzeugt in Zeile 2 eine leere Hash Tabelle und gibt diese als Parameter zur Methode **free2BoundVar (SimpleNode psi_node, HashMap < String , Boolean > vars, StringBuffer sb_psi)**, welche den Syntaxbaum mit der Wurzel **psi_node** von der Wurzel aus rekursiv durchsucht. Dadurch werden alle Variablen der Formel expr in **vars** gespeichert, so dass man abfragen kann, ob eine Variable freie oder gebundene Variable ist. Falls die Formel freie Variablen enthält, muss die transformierte Formel diese Variablen mit dem Existenzquantor binden. Ausserdem wird ein StringBuffer **sb_psi** als Parameter weitergegeben, welcher nach dem Durchlauf des Syntax Baums die transformierte Formel enthalten wird. Falls die Bedingung in Zeile 24 wahr ist, heisst das, dass eine freie Variable gefunden wurde. Die gefundene Variable wird sowohl in einem StringBuffer **sb_psi** als auch in einer HashMap mit dem Präfix **FVAR** gespeichert. Der Präfix FVAR dient hier als Merker dafür, dass es sich hier um eine freie Variable handelt, da sonst diese Information verloren gehen würde, nachdem man die transformierte Formel in der Datenbank gespeichert hat. Falls die Formel eine gebundene Variable enthält, wird dies in Zeile 29 festgestellt und die gefundene gebundene Variable wird unverändert sowohl in **sb_psi** als auch in **vars** gespeichert. Die Konstanten werden in Zeile 32 entdeckt und nur im StringBuffer gespeichert.

```
1  public static String  free2BoundVar( String psi  ){
2      HashMap<String, Boolean> vars = new HashMap<String, Boolean>();
3      ...
4
5      free2BoundVar(psi_node,  vars , sb );
6
7      for(String s : vars.keySet())
8        if( vars.get(s) == FREE_VAR )
9          prefix += "exists "+ s+" ";
10     ...
11
12     return prefix.length() == 0 ? psi : prefix + sb.toString()+");"
           ;
13   }
14
15
16
17   private static void free2BoundVar(SimpleNode psi_node,
18                        HashMap<String, Boolean> vars,
```

```
19                      StringBuffer sb_psi ){
20
21  ...
22
23          else if ( ( PARAM=psi_node.jjtGetChild(i).jjtGetChild(0).
                toString()).startsWith("VAR:") ) {
24            if (  ! vars.containsKey( PARAM.substring(5) ) ) {//not
                  bounded
25              vars.put( "FVAR"+ PARAM.substring(5), FREE_VAR );
26              sb_psi.append("FVAR"+ PARAM.substring(5));
27            }
28            else
29              sb_psi.append(PARAM.substring(5));
30          }
31          else if ( ( PARAM=psi_node.jjtGetChild(i).jjtGetChild(0).
                toString()).startsWith("KONST:")  ){
32            sb_psi.append(PARAM.substring(7));
33          }
34        }
35  ...
36
37  }
```

15.3 Statischer Modus - Erzeugung von Klassifikationsinstanzen

Wie wir bereits wissen, benutzt die kontrollierte Anfrageauswertung im statischen Modus die Klassifikationsinstanzen 5.3.2 anstatt *pot_sec*. Da nun unsere Geheimnissprache um freie Variablen erweitert ist, müssen wir die Klassifikationsinstanzen auch entsprechend erweitern.

Definition 15.3.1 (Erweiterte Klassifikationsinstanzen):
Gegeben ist die Menge von potentiellen Geheimnissen pot_sec $\subseteq L^f_{ps}$*. Dann ist die erweiterte Klassifikationsinstanz s bezüglich pot_sec folgenderweise definiert. Für jedes* $\Psi \in$ *pot_sec wird ein Tupel* $S(v^*_1, \ldots, v^*_n)$ *in s eingefügt, wobei für die*

$S(v_1^*, \ldots, v_n^*)$ *gilt:*

$$v_i^* = \begin{cases} v_i & \textit{falls } v_i \textit{ in } \Psi \textit{ als Konstante vorkommt} \\ \sim & \textit{falls } v_i \textit{ in } \Psi \textit{ als freie Variable vorkommt} \\ \# & \textit{falls } v_i \textit{ in } \Psi \textit{ als gebundene Variable vorkommt} \end{cases}$$

15.4 Optimierung von Klassifikationsinstanzen

Nun untersuchen wir, wie man die erweiterten Klassifikationsinstanzen optimieren kann. Dazu definieren wir erstmal, was eigentlich Optimierung heißt. Durch das Einbetten von freien Variablen in der Geheimnissprache können in der Menge der potentiellen Geheimnisse *pot_sec* und damit auch in Klassifikationsinstanzen redundante Informationen entstehen. Es ist von Vorteil, dass man diese redundanten Informationen aus den Klassifikationsinstanzen entfernt. Wir können genau dann einen Eintrag aus der Klassifikationsinstanz entfernen, wenn dadurch der Statische Zensor nicht ein anderes Ergebnis ausliefert. Da wir die erweiterten Klassifikationsinstanzen optimieren wollen, aber der statische Zensor über *pot_sec* definiert ist, wollen wir hier den Zusammenhang zwischen den beiden mit Hilfe der folgenden Definition nochmal klären.

Definition 15.4.1 (Erlaubte Anfragen)**:**
Sei pot_sec die Menge von potentiellen Geheimnissen und s die erweiterte Klassifikationsinstanz bezüglich pot_sec. Eine Anfrage Φ ist erlaubt genau dann, wenn $allowed_query(s, \Phi) = true$ ist. Wobei für $allowed_query(s, \Phi)$ gilt:

$$allowed_query(s, \Phi) = \ not\ censor_{stat}^{f,alt}(pot_sec, \Phi)$$

Nun definieren wir, was optimierte erweiterte Klassifikationsinstanzen heißt.

Definition 15.4.2 (Optimierte erweiterte Klassifikationsinstanzen)**:**
*Sei pot_sec die Menge von potentiellen Geheimnissen, s die erweiterte Klassifikationsinstanz bezüglich pot_sec. Wir bezeichnen die **optimierte-erweiterte Klassifikationsinstanz** von s mit $\tilde{s}$. Für $\tilde{s}$ gelten die folgenden Eigenschaften:*

1. *$\tilde{s}$ ist **semantiktreu** zu s:* *Für alle Anfragen Φ gilt:*

$$allowed_query(\tilde{s}, \Phi) \;=\; allowed_query(s, \Phi)$$

2. $\tilde{s}$ *ist* ***minimal*** *:* *Für alle* $\xi \in \tilde{s}$ *:* $\tilde{s} \setminus \{\xi\}$ *ist nicht semantiktreu zu* $\tilde{s}$

Nun überlegen wir uns, wie man die optimierte erweiterte Klassifikationsinstanz erzeugen kann. Wir gehen davon aus, dass wir erstmal eine leere Geheimnismenge *pot_sec* haben. Wir erstellen dementsprechend ein leere erweiterte Klassifikationsinstanz $\tilde{s}$. Danach, wenn der Admin einen neuen Eintrag Ψ in *pot_sec* einfügt, erzeugen wir für Ψ einen optimierten erweiterten Klassifikationsinstanzeintrag ξ, also ein Tupel $S(v_1^*, \ldots, v_n^*)$ im Sinne der Definition 15.3.1, und überlegen uns, ob wir es in $\tilde{s}$ einfügen müssen.

15.4.1 Fallstudie

Wir haben am Anfang die folgende leere Menge *pot_sec*:

pot_sec	

und die entsprechende optimierte erweiterte Klassifikationsinstanz:

Instanz $\tilde{s}$	

Fall 1: Angenommen der Admin fügt ein neues Geheimnis $\Psi = R(c_1, c_4)$ zu *pot_sec* hinzu. Dann sieht *pot_sec* so aus:

pot_sec	
	$R(c_1, c_4)$

Der entsprechende erweiterte Klassifikationsinstanzeintrag von $\Psi = R(c_1, c_4)$ ist $S(c_1, c_4)$. Die erweiterte Klassifikationsinstanz s sieht dann so aus:

Instanz s	
	$S(c_1, c_4)$

Nun ist die Frage, wie $\tilde{s}$ aussieht. Wir wollen erstmal wissen, ob wir $S(c_1, c_4)$ zu $\tilde{s}$ hinzufügen müssen. Wir nehmen erstmal an, dass $\tilde{s}$ durch das Hinzufügen $S(c_1, c_4)$ nicht mehr minimal wird. Also ist $S(c_1, c_4)$ überflüssig bezüglich $\tilde{s}$ und wir fügen nichts zu $\tilde{s}$ hinzu. Um zu sehen, ob nun $\tilde{s}$ semantiktreu zu s ist, betrachten wir die folgende Situation, und achten darauf, wie der statische Zensor dabei reagiert: Angenommen der User stellt die folgende Anfrage:

$$\text{Anfrage } \Phi = R(c_1, c_4)$$

$allowed_query(s, \Phi) =$ "$false$" wegen dem ersten Eintrag in s, aber $allowed_query(\tilde{s}, \Phi) =$ "$true$", weil es keinen Eintrag in $\tilde{s}$ gibt, der die Antwort verhindern könnte. Die Anfrage Φ deckt somit den 1-ten Eintrag in pot_sec auf. Daher ist unsere Annahme falsch und $S(c_1, c_4)$ muss zu $\tilde{s}$ hinzugefügt werden. d.h.:

Instanz $\tilde{s}$	
	$S(c_1, c_4)$

Nun taucht eine andere Frage auf. Ob nämlich $\tilde{s}$ irgendeinen Eintrag ξ hat, so dass durch das Hinzufügen von $S(c_1, c_4)$ der Eintrag ξ überflüssig geworden ist. In diesem Fall sollte ξ aus $\tilde{s}$ entfernt werden, denn sonst wäre $\tilde{s}$ nicht mehr minimal. Da $S(c_1, c_4)$ der einzige Eintrag in $\tilde{s}$ ist, ist dieser Fall hier ausgeschlossen.

Fall 2: Hinzuzufügen ist Eintrag $S(c_1, c_4)$. Da der $1-te$ Eintrag von $\tilde{s}$ gleich diesem Eintrag ist, darf $S(c_1, c_4)$ zu $\tilde{s}$ nicht hinzugefügt werden. Somit bleibt $\tilde{s}$ gleich.

Fall 3: Angenommen der Admin fügt ein neues Geheimnis $\Psi = R(Y, c_4)$ zu pot_sec hinzu, wobei Y eine freie Variable ist.

pot_sec	
	$R(c_1, c_4)$
	$R(Y, c_4)$

Der entsprechende erweiterte Klassifikationsinstanzeintrag von $\Psi = R(Y, c_4)$ ist $S(\sim, c_4)$:

Instanz s	
	$S(c_1, c_4)$
	$S(\sim, c_4)$

Analog zu den Überlegungen in Fall 1 gehen wir erstmal davon aus, dass $S(\sim, c_4)$ überflüssig ist, und aktualisieren $\tilde{s}$ nicht. Stellen wir uns folgende Situation vor: Angenommen der User stellt die folgende Anfrage:

$$\Phi = R(c_5, c_4)$$

$allowed_query(s, \Phi) = "false"$ wegen dem 2-ten Eintrag in s, aber $allowed_query(\tilde{s}, \Phi) = "true"$, weil $\tilde{s}$ keinen Eintrag enthält, der das verhindern kann. Deswegen muss $S(\sim, c_4)$ zu $\tilde{s}$ hinzugefügt werden. Denn wäre $S(\sim, c_4)$ in der Tabelle $\tilde{s}$, dann hätte der Zensor die Anfrage Φ nicht beantwortet. Daher ist unsere Annahme wieder falsch und $S(\sim, c_4)$ muss zu $\tilde{s}$ hinzugefügt werden:

Instanz $\tilde{s}$	
	$S(c_1, c_4)$
	$S(\sim, c_4)$

Nun prüfen wir, ob $\tilde{s}$ irgendeinen Eintrag ξ hat, so dass durch das Hinzufügen von $S(\sim, c_4)$ der Eintrag ξ überflüssig geworden ist. Auffällig ist, dass der Eintrag $S(c_1, c_4)$ aus $\tilde{s}$ entfernt werden muss, um die Minimalitätseigenschaft von $\tilde{s}$ zu gewährleisten, da $allowed_query(\{S(c1, c4), S(\sim, c4)\}, \Phi) = allowed_query(\{S(\sim, c4)\}, \Phi) = "false"$ ist. Also ist $S(c_1, c_4)$ überflüssig bezüglich $\tilde{s}$ und muss aus $\tilde{s}$ entfernt werden:

Instanz $\tilde{s}$	
	$S(\sim, c_4)$

Fall 4: Der Admin fügt ein neues Geheimnis $\Psi = (\exists X)R(c_1, X)$ zu pot_sec hinzu:

pot_sec	
	$R(c_1, c_4)$
	$R(Y, c_4)$
	$(\exists X)R(c_1, X)$

Hinzuzufügen ist der Eintrag $S(c_1, \#)$:

Instanz s	
	$S(c_1, c_4)$
	$S(\sim, c_4)$
	$S(c_1, \#)$

Analog zu Fall 3 (mit Useranfrage $\Phi = (\exists X)R(c_1, X)$) muss $S(c_1, \#)$ in $\tilde{s}$ eingetragen werden, da sonst der 3-te Eintrag von *pot_sec* schutzlos wäre.

Instanz $\tilde{s}$	
	$S(\sim, c_4)$
	$S(c_1, \#)$

Fall 5: Der Admin fügt ein neues Geheimnis $\Psi = (\exists X)R(X, c_4)$ zu *pot_sec* hinzu:

pot_sec	
	$R(c_1, c_4)$
	$R(Y, c_4)$
	$(\exists X)R(c_1, X)$
	$(\exists X)R(X, c_4)$

Die entsprechende erweiterte Klassifikationsinstanzeintrag von Ψ ist $S(\#, c_4)$:

Instanz s	
	$S(c_1, c_4)$
	$S(\sim, c_4)$
	$S(c_1, \#)$
	$S(\#, c_4)$

Hinzuzufügen ist der Eintrag $S(\#, c_4)$. Analog zu den obigen Überlegungen aktualisieren wir $\tilde{s}$ erstmal nicht. Stellen wir uns wieder eine Situation vor:
Angenommen der User stellt die folgende Anfrage:

$$\text{Anfrage } \Phi_1 = (\exists X)R(X, c_4)$$

Wegen dem 4-ten Eintrag in *pot_sec* darf diese Anfrage nicht beantwortet werden. Aber $\tilde{s}$ enthält keinen Eintrag, der das verhindern könnte. Wäre $S(\#, c_4)$ in der

Tabelle $\tilde{s}$, hätte der Zensor die Anfrage Φ nicht beantwortet. Deswegen muss $S(\#, c_4)$ zu $\tilde{s}$ hinzugefügt werden:

Instanz $\tilde{s}$	
	$S(\sim, c_4)$
	$S(c_1, \#)$
	$S(\#, c_4)$

Nachdem wir $S(\#, c_4)$ zu $\tilde{s}$ hinzugefügt haben, suchen wir in $\tilde{s}$ nach überflüssigen Einträgen. Auffällig ist, dass der erste Eintrag in $\tilde{s}$ die gleichen Attributswerte wie in $S(\#, c_4)$, bis auf den ersten Parameter, hat. Könnte es sein, dass einer von beiden Einträgen überflüssig ist? Wir haben gerade bereits festgestellt, dass $S(\#, c_4)$ nicht überflüssig ist. Ist dann $S(\sim, c_4)$ überflüssig? Könnte es sein, dass es Anfragen gibt, bei denen $S(\sim, c_4)$ zu einer Inferenzmeldung des Zensors führt und $S(\#, c_4)$ nicht? Bisher haben wir festgestellt, dass $S(\sim, c_4)$ bei Beantwortung von Anfragen zu einer Inferenzmeldung des Zensors führt, wenn die Anfragen die folgende Form haben:

- $\Phi_1 = R(c, c_4)$, wobei $c \in Const$ ist.
- $\Phi_1 = R(Y, c_4)$, wobei Y freie Variable ist.
- Die Anfragen, die keine gebundenen Variablen enthalten.

$S(\#, c_4)$ führt zu einer Inferenzmeldung des Zensors, wenn die Anfragen folgendermaßen aussehen:

- $\Phi_1 = R(c, c_4)$ wobei $c \in Const$ ist.
- $\Phi_1 = (\exists X)R(X, c_4)$ wobei X eine gebundene Variable ist.

Die Frage, ob $S(\#, c_4)$ bei Anfragen mit freien Variablen Einfluß auf den Zensor hat, ist noch offen. Um das festzustellen, betrachten wir die folgende Situation:
Der User stellt die folgende Anfrage:

$$\text{Anfrage } \Phi = R(X, c_4)$$

Bisher hat sich $S(\sim, c_4)$ in $\tilde{s}$ um die Anfragen in dieser Form gekümmert. Kann man $S(\sim, c_4)$ durch $S(\#, c_4)$ ersetzen?

Die Antwort lautet ja. Wäre $S(\#, c_4) \in \tilde{s}$, hätte der statische Zensor diese Anfrage nicht beantwortet. Wenn man als Antwort zu Φ ein Tupel ausgegeben hätte, wäre auch klar, dass es mindestens ein Element gibt, dessen 2-ter Attributswert $c4$ ist. Daraus folgt, dass wenn $S(\#, c_4) \in \tilde{s}$ wäre, hätte der Zensor einen Inferenzmeldung gegeben, wenn auch die Anfragen eine freie Variable wie in $\Phi = R(X, c_4)$ enthalten.

Insgesamt muss $S(\#, c_4)$ in $\tilde{s}$ eingefügt und $S(\sim, c_4)$ entfernt werden:

Instanz $\tilde{s}$	
	$S(c_1, \#)$
	$S(\#, c_4)$

Die Ergebnisse der obigen Fallstudie können wir in folgendem Satz zusammenfassen.

Satz 15.4.1 (Redundanz)**:**
Sei ξ ein erweiterter Klassifikationsinstanzeintrag und $\tilde{s}$ eine optimierte Klassifikationsinstanz. $\xi(A_i)$ bezeichnet den Wert des Attributs A_i in ξ. $\tilde{s}$ ist semantiktreu zu $\tilde{s} \cup \{\xi\}$ genau dann, wenn es ein $\xi^* \in \tilde{s}$ gibt, so dass für jedes A_i gilt:

$$\xi^*(A_i) = \begin{cases} \# \mid\sim\mid \xi(A_i) & \textit{falls } \xi(A_i) \in Const \\ \# \mid\sim & \xi(A_i) = \sim \\ \# & \xi(A_i) = \# \end{cases}$$

Dann sagen wir: ξ ist redundant bezüglich $\tilde{s}$ wegen ξ^*.

Beweis:
Wie wir sehen, wird die Redundanz anhand des paarweisen Vergleichs der Attributwerte von zwei erweiterten Klassifikationsinstanzeinträgen ξ, ξ^* festgelegt. Da es unendlich viele Konstanten gibt, kann es natürlich unendlich viele Wertepaare geben. Aber in unserem Fall wollen wir wissen, falls $\xi(A_i)$, $\xi^*(A_i) \in Const$ ist, ob sie die gleichen Werte haben oder nicht. Da nur zwei Attributwerte paarweise verglichen werden und die Attributwerte drei verschiedene Arten von Werten, nämlich Konstante, "∼" und "#" annehmen können, gibt es $3^2 + 1 = 9 + 1$ mögliche Wertepaare.

Dabei steht +1 für den Fall, dass $\xi(A_i)$ und $\xi^*(A_i)$ unterschiedliche konstante Werte haben.

1. $\xi(A_i) = c_1 \quad \xi^*(A_i) = c_2$ wobei $c_1, c_2 \in Const$ und $c_1 \neq c_2$
2. $\xi(A_i) = c_1 \quad \xi^*(A_i) = c_1$ wobei $c_1 \in Const$
3. $\xi(A_i) = c_1 \quad \xi^*(A_i) = \sim$ wobei $c_1 \in Const$
4. $\xi(A_i) = c_1 \quad \xi^*(A_i) = \#$ wobei $c_1 \in Const$
5. $\xi(A_i) = \sim \quad \xi^*(A_i) = c_1$ wobei $c_1 \in Const$
6. $\xi(A_i) = \sim \quad \xi^*(A_i) = \sim$
7. $\xi(A_i) = \sim \quad \xi^*(A_i) = \#$
8. $\xi(A_i) = \# \quad \xi^*(A_i) = c_1$ wobei $c_1 \in Const$
9. $\xi(A_i) = \# \quad \xi^*(A_i) = \sim$
10. $\xi(A_i) = \# \quad \xi^*(A_i) = \#$

Die Fallstudie hat die Fälle 1, 2, 3, 4 und 7 betrachtet. Durch den Rollentausch von ξ und ξ^* sind die restlichen Fälle analog zu den bereits betrachteten Fällen:

- Fall 5 ist analog zu Fall 3.
- Fall 6 ist analog zu Fall 2.
- Fall 8 ist analog zu Fall 4.
- Fall 9 ist analog zu Fall 7.
- Fall 10 ist analog zu Fall 2.

Wenn wir ein neues Element ξ zur erweiterten Klassifikationsinstanz $\tilde{s}$ hinzufügen wollen, müssen folgende zwei Fälle behandelt werden:

Minimierungsfall 1:
ξ ist redundant bezüglich $\tilde{s}$ und braucht nicht hinzugefügt zu werden.

Minimierungsfall 2:
ξ ist nicht redundant bezüglich $\tilde{s}$. In diesem Fall muss es hinzugefügt werden. Falls dadurch $\tilde{s}$ nicht mehr minimal ist, müssen die überflüssigen Einträge aus $\tilde{s}$ entfernt werden.

Basierend auf den obigen Überlegungen sollen die erweiterten Klassifikationsinstanzen mit Hilfe des folgenden Algorithmus erzeugt und minimiert werden:

15.5 Algorithmus: Clino (CLassification INstance Optimizer)

Eingabe: Geheimnismenge *pot_sec*;
Ausgabe: Optimierte erweiterte Klassifikationsinstanz $\tilde{s}$;

Schritt 1: Erzeuge eine leere erweiterte Klassifikationsinstanz $\tilde{s}$;

Schritt 2:

(a) Erzeuge für jedes $\Psi = (\exists X_1)\dots(\exists X_m)R(v_1,\dots,v_n)$ mit $\Psi \in pot_sec$ ein Tupel $\xi = S(v_1^*,\dots,v_n^*)$ im Sinne der Definition 15.3.1;

(b) Führe obige zwei Fallunterscheidungen folgendermaßen durch:

```
1 if ( Minimierungsfall 1 ) {
2     //Tue nichts, da ξ redundant bez\"uglich s̃ ist.
3         return;
4 }
5 else { //Minimierungsfall 2
6         s_red = { ξ_red | ξ_red ∈ s̃ und
7         ξ_red ist redundant bez\"uglich {ξ} };
8         s̃ = s̃ \ s_red ∪ {ξ};
9 }
```

15.6 Korrektheitsbeweis für Clino

Nun wollen wir beweisen, dass die durch Clino berechnete optimierte erweiterte Klassifikationsinstanz korrekt ist. Dafür brauchen wir erstmal ein paar Hilfssätze.

Satz 15.6.1 (EqRel)**:**
Die Semantiktreue (S) ist eine Äquivalenzrelation, d.h. für sie gelten die folgenden Eigenschaften:

- *S ist reflexiv.*
- *S ist symmetrisch.*
- *S ist transitiv.*

Beweis:
1. S ist reflexiv: Für jede erweiterte Klassifikationsinstanz s gilt:

$$s \text{ ist semantiktreu zu } s.$$

s **ist semantiktreu zu** s **heißt laut Definition 15.4.2:** Für jede Anfrage Φ gilt:

$$allowed_query(s, \Phi) = allowed_query(s, \Phi). \textbf{ Klar!}$$

2. S ist symmetrisch: Für zwei erweiterte Klassifikationsinstanzen s_1 und s_2 gilt:

$$s_1 \text{ ist semantiktreu zu } s_2 \iff s_2 \text{ ist semantiktreu zu } s_1.$$

Formal :

$$\begin{aligned} & allowed_query(s_1, \Phi) = allowed_query(s_2, \Phi) \\ \iff & allowed_query(s_2, \Phi) = allowed_query(s_1, \Phi). \textbf{ Klar!} \end{aligned}$$

3. S ist transitiv: Für drei beliebige erweiterte Klassifikationsinstanzen s_1, s_2 und s_3 gilt:

$$(s_1 \text{ ist semantiktreu zu } s_2) \textit{ und } (s_2 \text{ ist semantiktreu zu } s_3) \implies s_1 \text{ ist semantiktreu zu } s_3$$

Formal : Für jede Anfrage Φ gilt:

$$\begin{gathered} allowed_query(s_1, \Phi) = allowed_query(s_2, \Phi) \\ \text{und } allowed_query(s_2, \Phi) = allowed_query(s_3, \Phi) \end{gathered}$$

$$\implies allowed_query(s_1, \Phi) = allowed_query(s_3, \Phi) \textbf{ Klar!}$$

■

Satz 15.6.2:
Seien s_1 und s_2 zwei erweiterte Klassifikationsinstanzen. Dann gilt für jede Anfrage Φ:

$$allowed_query(s_1 \cup s_2, \Phi) = allowed_query(s_1, \Phi) \quad \wedge \quad allowed_query(s_2, \Phi);$$

Beweis:
Seien pot_sec_1 und pot_sec_2 die ursprünglichen Geheimnismengen, von denen s_1 und s_2 erzeugt wurden.

Definition $allowed_query$:
$allowed_query(s_1 \cup s_2, \Phi) = \ not\ censor_{stat}^{f,alt}(pot_sec_1 \cup pot_sec_2, \Phi)$

Definition $censor_{stat}^{f,alt}$:

$$\begin{gathered} not\ censor_{stat}^{f,alt}(pot_sec_1 \cup pot_sec_2, \Phi) \ = \ not(\text{es existiert } \Psi(V) \in \\ pot_sec_1 \cup pot_sec_2 \text{ und für alle } A \in U: \quad \text{wenn} \\ \Psi(V)[A] \in Const \text{ dann } \Phi[A] = \Psi(V)[A] \text{ und} \\ \text{wenn } \Psi(V)[A] \text{ freie Variable ist, dann } \Phi[A] \in Const) \end{gathered}$$

$not\ censor_{stat}^{f,alt}(pot_sec_1 \cup pot_sec_2, \Phi) =$

$$\begin{gathered} not\{ \\ (\text{es existiert } (\Psi(V))(\Psi(V) \in pot_sec_1 \text{ und für alle } A \in U: \end{gathered}$$

$$\text{wenn } \Psi(V)[A] \in Const \text{ dann } \Phi[A] = \Psi(V)[A] \text{ und}$$
$$\text{wenn } \Psi(V)[A] \text{ freie Variable ist, dann } \Phi[A] \in Const)$$
$$\vee$$
$$(\text{es existiert } (\Psi(V))(\Psi(V) \in pot_sec_2 \text{ und für alle } A \in U:$$
$$\text{wenn } \Psi(V)[A] \in Const \text{ dann } \Phi[A] = \Psi(V)[A] \text{ und}$$
$$\text{wenn } \Psi(V)[A] \text{ freie Variable ist, dann } \Phi[A] \in Const)$$
$$\}$$

Laut De Morgansche Regel gilt:
$not\ censor^{f,alt}_{stat}(pot_sec_1 \cup pot_sec_2, \Phi) =$

$$not\{$$
$$(\text{es existiert } (\Psi(V))(\Psi(V) \in pot_sec_1 \text{ und für alle } A \in U:$$
$$\text{wenn } \Psi(V)[A] \in Const \text{ dann } \Phi[A] = \Psi(V)[A] \text{ und}$$
$$\text{wenn } \Psi(V)[A] \text{ freie Variable ist, dann } \Phi[A] \in Const)\ \}$$
$$\wedge$$
$$not\{$$
$$(\text{es existiert } (\Psi(V))(\Psi(V) \in pot_sec_2 \text{ und für alle } A \in U:$$
$$\text{wenn } \Psi(V)[A] \in Const \text{ dann } \Phi[A] = \Psi(V)[A] \text{ und}$$
$$\text{wenn } \Psi(V)[A] \text{ freie Variable ist, dann } \Phi[A] \in Const)\ \}$$

$not\ censor^{f,alt}_{stat}(pot_sec_1 \cup pot_sec_2, \Phi) =$

$$not\ censor^{f,alt}_{stat}(pot_sec_1, \Phi) \wedge\ not\ censor^{f,alt}_{stat}(pot_sec_2, \Phi)$$
$$= allowed_query(s_1, \Phi) \quad \wedge \quad allowed_query(s_2, \Phi)$$

■

Satz 15.6.3 (Vereinigung)**:**
Seien a, b und c drei beliebige erweiterte Klassifikationsinstanzen. Dann gilt:

$$a \text{ ist semantiktreu zu } b \implies a \cup c \text{ ist semantiktreu zu } b \cup c.$$

Beweis:
a ist semantiktreu zu b heißt, dass für jede Anfrage Φ gilt:

$$allowed_query(a, \Phi) = allowed_query(b, \Phi). \textbf{ (*)}$$

$a \cup c$ ist semantiktreu zu $b \cup c$ heißt, dass für jede Anfrage Φ gilt:

$$allowed_query(a \cup c, \Phi) = allowed_query(b \cup c, \Phi). \textbf{ (**)}$$

** umgeschrieben unter Berücksichtigung von Satz 15.6.3:

$$\begin{aligned} & allowed_query(a, \Phi) \;\wedge\; allowed_query(c, \Phi) \\ = \; & allowed_query(b, \Phi) \;\wedge\; allowed_query(c, \Phi) \textbf{(***)} \end{aligned}$$

*** umgeschrieben unter Berücksichtigung von *:

$$\begin{aligned} & allowed_query(b, \Phi) \;\wedge\; allowed_query(c, \Phi) \\ = \; & allowed_query(b, \Phi) \;\wedge\; allowed_query(c, \Phi) \end{aligned}$$

■

Satz 15.6.4 (Korrektheit von Clino)**:**
Sei pot_sec die Menge von potentiellen Geheimnissen, s die erweiterte Klassifikationsinstanz bezüglich pot_sec und $\tilde{s}$ die durch Clino berechnete optimierte erweiterte Klassifikationsinstanz. Dann ist $\tilde{s}$ korrekt im Sinne der Definition 15.4.2.

Beweis:
Im Folgenden bezeichnen wir mit s die erweiterte Klassifikationsinstanz bezüglich *pot_sec*. Damit es übersichtlicher wird, bezeichnen wir s und $\tilde{s}$ in der i-ten Iteration von Clino in Schritt 2 jeweils mit s_i und $\tilde{s}_i$. Zu beachten ist, dass *pot_sec* und s in der i-ten Iteration i Einträge haben.

Wir beweisen das mit Hilfe der vollständigen Induktion über $n = \mid pot_sec \mid$, also die Anzahl der Elemente aus *pot_sec*.

Induktionsanfang: $n = 1$: Für $n = 1$ erfüllt $\tilde{s}$ offensichtlich diese Eigenschaft, da $\tilde{s}_1 = s_1$ gilt.

Induktionsvoraussetzung: $\tilde{s}_n$ ist minimal und $\tilde{s}_n$ ist semantiktreu zu s_n.

Induktionsschluss: $n \rightarrow n+1$: Nun werden der $n+1$-te Eintrag Ψ in *pot_sec* und der zu Ψ gehörige erweiterte Klassifikationsinstanzeintrag ξ in s eingefügt. An dieser Stelle wird Schritt 2.b von Clino angewendet:

Minimierungsfall 1: $\tilde{s}_{n+1} := \tilde{s}_n$

Minimierungsfall 2: $\tilde{s}_{n+1} := \tilde{s}_n \setminus s_{red} \cup \{\xi\}$

Wir beweisen zuerst, dass $\tilde{s}_{n+1}$ minimal ist.

Minimierungsfall 1: $\tilde{s}_{n+1} := \tilde{s}_n$

Laut Induktionsannahme: $\tilde{s}_n$ ist minimal. Daraus folgt, dass $\tilde{s}_{n+1}$ auch minimal ist.

Minimierungsfall 2: $\tilde{s}_{n+1} := \tilde{s}_n \setminus s_{red} \cup \{\xi\}$

Laut Clino: Für alle $\xi^* \in \tilde{s}_n$ gilt :

$$\xi^* \text{ redundant bezüglich } \{\xi\} \Longrightarrow \xi^* \in s_{red} \textbf{ (*1)}$$

Beweis durch Widerspruch:
Angenommen $\tilde{s}_{n+1}$ ist nicht minimal:

$$\text{Es existiert } \xi^{red} \textit{ mit } \xi^{red} \in \tilde{s}_n \cup \{\xi\} \textit{ und } \xi^{red} \notin s_{red} \text{ so dass:}$$
$$\tilde{s}_n \cup \{\xi\} \setminus s_{red} \setminus \{\xi^{red}\} \text{ ist semantiktreu zu } \tilde{s}_n \cup \{\xi\} \setminus s_{red}$$

Wir wissen, dass in Minimierungsfall 2 $\xi \in \tilde{s}_{n+1}$ ist. D.h. $\xi^{red} \neq \xi$ **(*2)**

Aus *2 und $\xi^{red} \in \tilde{s}_n$ *und* $\xi^{red} \notin s_{red}$ folgt:

$$\xi^{red} \in \tilde{s}_n \setminus s_{red} \textbf{ (*3)}$$

Aus *1 und $\xi^{red} \notin s_{red}$ folgt:

$$\xi^{red} \text{ ist nicht redundant bezüglich } \{\xi\} \textbf{ (*4)}$$

Aus *3 und *4 folgt: $\exists \xi^* \in \tilde{s}_n \setminus s_{red} \setminus \{\xi^{red}\}$ folgt:

$$\xi^{red} \text{ ist redundant bezüglich } \{\xi^*\}$$

D.h. $\tilde{s}_n$ ist NICHT minimal. Das widerspricht aber der Induktionsannahme! Daraus folgt, dass $\tilde{s}_{n+1}$ auch in Minimierungsfall 2 minimal ist. Da Minimierungsfall 1 und Minimierungsfall 2 eine vollständige Fallunterscheidung bilden, ist $\tilde{s}$ minimal. ■

Nun beweisen wir, dass $\tilde{s}_{n+1}$ semantiktreu zu s ist.

Minimierungsfall 1: $\tilde{s}_{n+1} := \tilde{s}_n$ **(*1)**

Laut Induktionsannahme gilt: $\tilde{s}_n$ semantiktreu zu s_n **(*2)**

Laut *2 und Satz 15.6.3 gilt:

$$\tilde{s}_n \cup \{\xi\} \text{ semantiktreu zu } s_n \cup \{\xi\} \textbf{ (*3)}$$

ξ redundant bezüglich $\tilde{s}_n$ heißt:

$$\tilde{s}_n \text{ semantiktreu zu } \tilde{s}_n \cup \{\xi\} \textbf{ (*4)}$$

Aus *3 und *4 und Satz 15.6.1 folgt :

$$\tilde{s}_n \text{ semantiktreu zu } s_n \cup \{\xi\} \textbf{ (*5)}$$

Aus *1 und *5 folgt, dass $\tilde{s}_{n+1}$ ist semantiktreu zu $s_n \cup \{\xi\} = s_{n+1}$ in Minimierungsfall 1 ist. ■

Minimierungsfall 2: $\tilde{s}_{n+1} = \tilde{s}_n \setminus s_{red} \cup \{\xi\}$ **(*0)**
Aus Übersichtlichkeitsgründen werden wir im Folgenden den Ausdruck "a ist semantiktreu zu b" mit "$a \overset{\text{st}}{\leftrightarrow} b$" bezeichnen.

Sei $s_{red} = \{\xi_{red_1}, \xi_{red_2}, ..., \xi_{red_k}\}$. Folgende Aussage ist korrekt:

$s_{n+1} = \ s_n \cup \{\xi\}$ //klar

$\overset{\text{st}}{\leftrightarrow} \tilde{s}_n \cup \{\xi\}$ //nach Satz[Vereinigung] und Induktionsannahme

$= \ \tilde{s}_n \setminus \{\xi_{red_1}\} \cup \{\xi_{red_1}\} \cup \{\xi\}$ //umformuliert

(*1) $\stackrel{\text{st}}{\leftrightarrow} \tilde{s}_n \setminus \{\xi_{red_1}\} \cup \{\xi\}$ //Satz 15.6.3 und $\xi_{red_1} \in s_{red}$

$= \quad \tilde{s}_n \setminus \{\xi_{red_1}\} \setminus \{\xi_{red_2}\} \;\; \cup \{\xi_{red_2}\} \cup \{\xi\}$ //umformuliert

...

$\stackrel{\text{st}}{\leftrightarrow} \tilde{s}_n \setminus \{\xi_{red_1}\} \setminus \{\xi_{red_2}\} ... \setminus \{\xi_{red_k}\} \cup \{\xi\}$ //nach Satz 15.6.3

$= \tilde{s}_n \setminus s_{red} \cup \{\xi\}$ //umformuliert

$= \tilde{s}_{n+1}$ //laut *0

Wie man den Satz 15.6.3 für die Aussage *1 anwenden kann, erklären wir ausführlicher. Satz 15.6.3 besagt:

$$a \stackrel{\text{st}}{\leftrightarrow} b \implies a \cup c \stackrel{\text{st}}{\leftrightarrow} b \cup c.$$

Setze a, b, c in *1 folgenderweise ein:

- $a = \{\xi\}$
- $b = \{\xi_{red_1}\} \cup \{\xi\}$
- $c = \tilde{s}_n \setminus \{\xi_{red_1}\}$

Nun ist $a \stackrel{\text{st}}{\leftrightarrow} b$ gültig, weil $\xi_{red_1} \in s_{red}$ gilt. Somit ist *1 auch gültig.

Laut obiger Aussage und Satz 15.6.1(Transitivität) folgt dann: $\tilde{s}_{n+1} \stackrel{\text{st}}{\leftrightarrow} s_{n+1}$
Daraus folgt: $\tilde{s} \stackrel{\text{st}}{\leftrightarrow} s$ in Minimierungsfall 2

Da Minimierungsfall 1 und Minimierungsfall 2 eine vollständige Fallunterscheidung bilden, folgt, dass $\tilde{s}$ semantiktreu zu s ist. ■

Insgesamt haben wir gezeigt, dass das durch Clino berechnete $\tilde{s}$ minimal und semantiktreu zu s ist. Damit ist der Satz korrekt. ■

15.7 Einpassung in den Prototypen

Die Idee der Klassifikationsinstanzen wurde bereits von Sebastian Sonntag folgendermaßen implementiert:

Wenn der Admin ein neues Geheimnis zu *pot_sec* hinzufügen will, dann werden folgende Schritte durchgeführt:

1. Der Admin trägt ein neues Geheimnis in *pot_sec* ein;
2. *pot_sec* wird mit Hilfe von Prover9 minimalisiert;

Wenn der User eine neue Anfrage stellt:

1. Es wird eine neue Klassifikationsinstanz bezüglich *pot_sec* erstellt. (Siehe die Methode createPot_secRel(LinkedList⟨Formula⟩ pot_sec) in core.UserData.java, welche von cqeClosed(Formula f) in core.Application aufgerufen wird)
2. Es wird untersucht, wie die Anfrage beantwortet werden soll.

Auffällig ist, dass die Klassifikationsinstanz bei jeder Anfrage neu erstellt wird. Es wäre besser, wenn die Klassifikationsinstanz nicht jedesmal neu erstellt und nur dann aktualisiert würde, wenn der Admin ein neues Geheimnis zu *pot_sec* hinzufügt. Dann sollte aber für jeden User eine eigene Klassifikationsinstanz erstellt werden. Aus Zeitgründen wurde dieses nicht gemacht. Daher werden wir auch nur ein Klassifikationsinstanz für alle User bei jeder Anfrage erstellen.

15.8 Statischer Zensor für Oracle-DB

Wie wir oben bereits erwähnt haben, benutzt die kontrollierte Anfrageauswertung im statischen Modus die erweiterten Klassifikationsinstanzen. Nun überlegen wir uns, wie man feststellen kann, ob eine Anfrage eine Gefahr darstellt oder nicht. Wir betrachten folgendes Beispiel: Angenommen der User stellt die folgende Anfrage:

$$\Psi = (\exists X_1)\dots(\exists X_m)db(\underbrace{a_1,\dots,a_k}_{Attrib.A_1,\dots,A_k},\underbrace{X_1,\dots,X_m}_{Attrib.B_1,\dots,B_m})$$

Und angenommen die entsprechende erweiterte Klassifikationsinstanz-Datenbank heißt R_s. Dann können wir mit folgender SQL Anfrage [Loc09] herausfinden, ob R_s ein Element hat, das durch diese Anfrage aufgedeckt würde:

SELECT COUNT(*)
FROM R_s
WHERE (A_1 = 'a_1' OR A_1 = '#' OR A_1 = '∼')
AND ... AND (A_k = 'a_k' OR A_k = '#' OR A_k = '∼')
AND (B_1 = '#') AND (B_2 = '#') AND ... AND (B_m = '#')

15.9 Implementierung des Clino Algorithmus:

Nun zeigen wir die Implementierung der Verwaltung der Klassifikationsinstanzen. Die Methode translatePotsec in Klasse Clino erzeugt eine optimierte erweiterte Klassifikationsinstanz.

```
1  public static void translatePotSec(String user){
2   try {
3    Connection con = DatabaseConf.getDBConnection();
4    String query = "SELECT * FROM " +
5             SQLInteraction.schema + "." + user + "Potsec";
6    ResultSet rs = con.createStatement().executeQuery( query );
7
8    while (rs.next()){
9    /*
10    * read psi from pot_sec
11    */
12     String psi = rs.getString(1) + ";";
13     add( psi);
14   }
15   rs.close();
16  } catch (Exception e) {
17   System.out.println(e.getMessage());
18  }
19 }
```

Wie man sieht, liest diese Methode die Menge von potentiellen Geheimnissen(Zeile 3-12) , und ruft für jeden Eintrag “psi“ die Methode add(psi) auf. Die Methode add ist der Clino Algorithmus, welcher bereits beschrieben wurde.

```
public static void add(String psi){
 try {
  /*
   * translate psi to classification instance entity xi
   */
   ArrayList<String []> xi_list =
               Translater.extract_psi2xi( psi );

   for ( String [] xi : xi_list ) {
     if (   hasRelevantEntityFor( xi   )   ){
   /* minimization case 1
    * do nothing
    */
    }
    else {
      /*
     * minimization case 2
     */
    removeRedundantEntityWRT( xi );
    /*
     * add xi to classification Instance
     */
     addToClassificationInstance( xi );
     }
   }
 } catch (Exception e) {
  System.out.println("Clino.add():---> " +e.getMessage());
 }
}
```

Die Methode *add* benutzt vier Methoden. Die erste Methode *extract_psi2xi(psi)* aus der Klasse Translater dient dazu, dass der Eintrag *psi* extrahiert wird und in den Klassifikationsinstanzeintrag *xi* übersetzt wird. Dabei kann *psi* eine Formel sein, welche auch Junktoren enthält. In diesem Fall wäre die Übersetzung eine Liste von Klassifikationsinstanzeinträgen. Die Übersetzung entspricht der Definition 15.3.1.

Nachdem *psi* extrahiert wurde, wird für jeden entstandenen Klassifikationsinstanzeintrag xi zuerst festgestellt, ob er zur entsprechenden Klassifikationsinstanz hinzugefügt werden muss, wie der Algorithmus Clino beschreibt. Das geschieht mit Hilfe der Methode $hasRelevantEntityFor(xi)$. $hasRelevantEntityFor(xi)$ ist die Implementierung von Satz 15.4.1. xi ist hier ein String array, wobei $xi[0]$ der Tabellenname der entsprechenden Klassifikationsinstanz $\tilde{s}$ ist. xi ist im Satz mit ξ^* bezeichnet.

Beispiel:

Gegeben: Klassifikationsinstanzeintrag $ARMBRUCH_PS(hans)$

xi als Array: $xi = \{\ ARMBRUCH_PS,\quad hans\ \}$

Hier ist $xi[0] = ARMBRUCH_PS$ der Name der entsprechenden Klassifikationsinstanz.

```
1  public static boolean hasRelevantEntityFor(String [] xi ){
2   String query = "SELECT "+ option1 + " FROM "+
3        SQLInteraction.schema+"." + xi[0]  + " WHERE ";
4   String [] columns = LookUpTable.getTableColumns( xi[0]  );
5   try{
6    for(int k = 0; k < columns.length ; k++){
7     if( xi[ k+1 ] == "#" )
8      query += "("+columns[k]+" = '#' ) "
9            +( k < columns.length-1 ? "AND " : "" ) ;
10    else if( xi[ k+1 ] == "~" )
11     query += "("+columns[k]+" = '#' "
12           + " OR "+columns[k]+" = '~' ) "
13      +( k < columns.length-1 ? " AND " : "" ) ;
14    else
15     query += "("+columns[k]+" = '#' " + " OR "
16                 +columns[k]+" = '~' OR "
17                 +columns[k]+" = '"+ xi[ k+1 ] + "'  ) "
18                 +( k < columns.length-1 ? " AND " : "" ) ;
19   }
20   Statement stm = DatabaseConf.getDBConnection().createStatement();
21   ResultSet rs = stm.executeQuery( query );
22
23   if(rs.next()){
24    xi[0] + toString(rs, columns.length));
25    rs.close();
26    stm.close();
27    return true;
28   }
29   rs.close();
```

```
30    stm.close();
31  }catch(Exception e) {
32    System.out.println("HREF()---> {\n your_query" + query +" " +
33    e.getMessage() );
34  }
35  return false;
36 }
```

hasRelevantEntityFor(xi) gibt *true* aus, falls xi redundant bezüglich der entsprechenden Klassifikationsinstanz ist. In diesem Fall wird xi nicht hinzugefügt. Sonst muss xi zu der Klassifikationsinstanz hinzugefügt werden. Davor müssen noch die Einträge aus der Klassifikationsinstanz entfernt werden, die durch Hinzufügen von xi überflüssig sein könnten, die also redundant bezüglich $\{xi\}$ sind. Dafür ist die Methode *removeRedundantEntityWRT*(xi) zuständig.

```
1  private static void  removeRedundantEntityWRT( String [] xi  ){
2   String [] columns = LookUpTable.getTableColumns( xi[0]  );
3   String query = "DELETE  FROM "+SQLInteraction.schema+"." + xi[0];
4   try{
5    boolean remove = false;
6    String wherepart ="";
7    for(int k = 0; k < columns.length ; k++){
8     if( xi [ k+1 ] !=  "#" && xi [ k+1 ] !=  "~" ) // const
9      wherepart += "("+columns[k]+" = '"+ xi [ k+1 ] + "' ) "
10                  +( k < columns.length-1 ? " AND " : "" ) ;
11    else if( xi [ k+1 ] == "~" ){
12      wherepart += "NOT ("+columns[k]+" = '#' ) "
13                  +( k < columns.length-1 ? " AND " : "" ) ;
14      remove = true;
15    }
16    else {
17    remove = true;
18    // in case of '#' does not play the value
19      // of column[k] any role
20    }
21   }
22   if ( ! remove )
23    return;
24   query += wherepart.length()==0 ? "" : " WHERE "+ wherepart;
25   ResultSet rs = DatabaseConf.getDBConnection()
26                 .createStatement()
27                 .executeQuery( query );
```

```
28    rs.close();
29   }catch(Exception e) {
30     System.out.println( e.getMessage() );
31   }
32 }
```

removeRedundantEntityWRT(xi) ist nicht anders als die Implementierung von Satz 15.4.1. Im Satz setzen wir $\tilde{s} = \{x_i\}$. xi ist ein String array, wobei $xi[0]$ der Tabellenname der Klassifikationsinstanz ist, in die wir xi hinzufügen wollen. Es wird für jeden Eintrag aus dieser Klassifikationsinstanz geprüft, ob er redundant bezüglich xi ist.

Nach dieser Prozedur wird xi zur entsprechenden Klassifikationsinstanz hinzugefügt. Dies ist in der Methode *addToClassificationInstance*(xi) implementiert. Wie man aus der Implementierung sieht, ist xi ein Stringarray. $xi[0]$ ist der Name der Klassifikationsinstanz, in die wir xi hinzufügen wollen. Zeile 3-7 wandelt xi in eine gültige SQL Anfrage als String um. Anschließend wird in Zeile 9 diese SQL Anfrage gestellt.

```
1  private static void  addToClassificationInstance( String xi [] ) {
2   try {
3    String sqlInsert = "INSERT INTO " + xi[0]  + VALUES + "(";
4    for (int i = 1; i < xi.length; i++)
5     sqlInsert += ( i>1 ? ", '" : "'" ) +xi [i] +"'";
6
7    sqlInsert += ")";
8    Statement s = DatabaseConf.getDBConnection().createStatement();
9    s.executeUpdate(sqlInsert);
10   s.close();
11  }catch (SQLException e) {
12    e.printStackTrace();
13  }
14 }
```

15.10 Testen von Clino

Da die Methode minimizePotSec() des Prototyps einen Fehler enthält(s. ToDos), können wir leider nicht alle Fälle aus der Fallstudie mit Hilfe des Prototyps testen. Daher testen wir die Implementierung erstmal außerhalb des Prototyps.

Wir haben die folgende Datenbankinstanz:

REHA	NAME	ORT	DAUER	KRANKHEIT
	⋮	⋮	⋮	⋮

und eine leere Geheimnismenge:

$PotSec\{\}$

Fall 1: Der Admin fügt den Eintrag psi=reha(fred, dortmund, lange, beinbruch) zu *PotSec* hinzu. Die Methode $extract_psi2xi(psi)$ aus der Klasse Translater übersetzt *psi* und erzeugt den folgenden Klassifikationsinstanzeintrag *xi*:

$$xi = REHA_PS(fred, dortmund, lange, beinbruch)$$

Clino fügt diesen zur Tabelle REHA_PS hin. Die Tabelle REHA_PS sieht nun **wie gewünscht** folgendermaßen aus:

REHA_PS	NAME	ORT	DAUER	KRANKHEIT
	fred	dortmund	lange	beinbruch

Fall 2:
psi=reha(fred, dortmund, lange, beinbruch)
xi=REHA_PS(fred, dortmund, lange, beinbruch)
Ergebnis **wie gewünscht**:

REHA_PS	NAME	ORT	DAUER	KRANKHEIT
	fred	dortmund	lange	beinbruch

Fall 3:
psi=reha(X, dortmund, lange, beinbruch)
xi=REHA_PS(∼, dortmund, lange, beinbruch)
Ergebnis **wie gewünscht**:

REHA_PS	NAME	ORT	DAUER	KRANKHEIT
	∼	dortmund	lange	beinbruch

Fall 4:
psi=exists X reha(hans, bochum, X, beinbruch);
xi=REHA_PS(hans, bochum, #, beinbruch)
Ergebnis **wie gewünscht**:

REHA_PS	NAME	ORT	DAUER	KRANKHEIT
	∼	dortmund	lange	beinbruch
	hans	bochum	#	beinbruch

Fall 5:
psi=exists X reha(X, dortmund, lange, beinbruch)
xi=REHA_PS(#, dortmund, lange, beinbruch)
Ergebnis **wie gewünscht**:

REHA_PS	NAME	ORT	DAUER	KRANKHEIT
	hans	bochum	#	beinbruch
	#	dortmund	lange	beinbruch

15.11 Testen von Clino innerhalb des Prototyps

Die Tests wurde in vier Testszenen mit der Tabelle *armbruch* durchgeführt und dokumentiert, wobei die Tabelle folgendermaßen aussieht:

ARMBRUCH	NAME
	dumpfbacke
	gustav
	hans
	jude
	lena
	ndula
	peter
	susan
	wolfgang

Wir haben die folgenden Fälle ins Betracht gezogen:

Testszene 1: DB und PotSec haben gleichen Eintrag.

a) Anfrage Φ ist weder Geheimnis noch wahr in DB.

b) Anfrage Φ ist sowohl Geheimnis als auch wahr in DB.

c) Anfrage Φ enthält eine gebundene Variable.

Testszene 2: DB und PotSec haben keinen gleichen Eintrag.

a) Anfrage Φ ist ein Geheimnis, aber nicht wahr in DB.

b) Anfrage Φ ist kein Geheimnis, aber wahr in DB.

c) Anfrage Φ enthält eine gebundene Variable.

Testszene 3: PotSec hat einen Eintrag mit gebundener Variable.

a) Anfrage Φ ist eine Formel, die nur Konstanten enthält und nicht wahr in der DB ist.

b) Anfrage Φ ist eine Formel, die nur Konstanten enthält und wahr in der DB ist.

c) Anfrage Φ enthält gebundene Variable.

Testszene 4: PotSec hat einen Eintrag mit freien Variablen.

a) Anfrage Φ ist eine Formel, die nur Konstanten enthält und nicht wahr in der DB ist.

b) Anfrage Φ ist eine Formel, die nur Konstante enthält und wahr in der DB ist.

c) Anfrage Φ enthält eine gebundene Variable.

Auffällig ist, dass wir die offenen Anfragen nicht betrachtet haben. Der Grund dafür ist, dass man offene Anfragen durch eine Sequenz von geschlossenen Anfragen ersetzen kann.

Ausserdem wird in Testszene 4 die Approximation von freien Variablen im Dynamischem Modus getestet.

15.11.1 Testszene 1

a) user: adalat_jcqe
schema: jcqe_normal
potsec={armbruch(hans)}
Klassifikationsinstanz ARMBRUCH_PS={ hans }
log={}
anfrage=armbruch(aa);

Gewünschtes Ergebnis: (not armbruch(aa))
Ergebnis: (not armbruch(aa))
log=(not armbruch(aa))

b) user: adalat_jcqe
schema: jcqe_normal
potsec={armbruch(hans)}
Klassifikationsinstanz ARMBRUCH_PS={ hans }
log=(not armbruch(aa))
anfrage=armbruch(hans);

Gewünschtes Ergebnis: MUM
Ergebnis: MUM
log=(not armbruch(aa))

c) user: adalat_jcqe
schema: jcqe_normal
potsec={armbruch(hans)}
Klassifikationsinstanz ARMBRUCH_PS={ hans }
log=(not armbruch(aa))
anfrage=exists X armbruch(X);

Gewünschtes Ergebnis: exists X armbruch(X)
Ergebnis: exists X armbruch(X)
log={ (not armbruch(aa)), exists X armbruch(X) }

15.11.2 Testszene 2

a) user: adalat_jcqe
schema: jcqe_normal
potsec={armbruch(rudi)}
Klassifikationsinstanz ARMBRUCH_PS={ rudi }
log={}

anfrage=armbruch(rudi);
Gewünschtes Ergebnis: MUM
Ergebnis: MUM
log={}

b) user: adalat_jcqe
schema: jcqe_normal
potsec={armbruch(rudi)}
Klassifikationsinstanz ARMBRUCH_PS={ rudi }
log={}

anfrage=armbruch(hans);
Gewünschtes Ergebnis: armbruch(hans)
Ergebnis: armbruch(hans)
log={ armbruch(hans) }

c) user: adalat_jcqe
schema: jcqe_normal
potsec={armbruch(rudi)}
Klassifikationsinstanz ARMBRUCH_PS={ rudi }
log={ armbruch(hans) }

anfrage=exists X armbruch(X);
Gewünschtes Ergebnis: exists X armbruch(X)
Ergebnis: exists X armbruch(X)
log={ armbruch(hans)
exists X armbruch(X) }

15.11.3 Testszene 3

a) user: adalat_jcqe
schema: jcqe_normal
potsec={ exists X armbruch(X) }
Klassifikationsinstanz ARMBRUCH_PS={ # }
log={}

anfrage=armbruch(rudi);
Gewünschtes Ergebnis: MUM
Ergebnis: MUM
log={}

b) user: adalat_jcqe
schema: jcqe_normal
potsec={ exists X armbruch(X) }
Klassifikationsinstanz ARMBRUCH_PS={ # }

log={}

anfrage=armbruch(hans);
Gewünschtes Ergebnis MUM
Ergebnis: MUM
log={}

c) user: adalat_jcqe
schema: jcqe_normal
potsec={ exists X armbruch(X) }
Klassifikationsinstanz ARMBRUCH_PS={ # }
log={}

anfrage=exists X armbruch(X);
Gewünschtes Ergebnis: MUM
Ergebnis: MUM
log={}

15.11.4 Testszene 4

Der Admin fügt ein neues Geheimnis psi=armbruch(X) zu adalatPotsec hinzu. Für diesen Fall haben wir die Approximation implementiert, die im Satz der Approximation beschrieben ist. Da das Geheimnis *psi* eine freie Variable enthält, sollte *psi* zuerst durch seinen existentiellen Abschluss ersetzt werden. Dafür haben wir in Klasse **Translater** die Methode **free2BoundVar(String psi)** implementiert. Diese Methode ersetzt die freien Variablen aus *psi* durch gebundene Variablen, wobei die übersetzten Variablen nun noch einen Präfix **FVAR** dazu bekommen. Das ist eine **Konvention** dafür, dass die Information über freie Variablen nicht verloren geht. Das übersetzte *psi*=exists FVARX armbruch(FVARX) wird in PotSec zugefügt. Für Clino übersetzt Translater die neue Formel als *xi*=ARMBRUCH_PS(∼).

a) user: adalat_jcqe
schema: jcqe_normal
potsec={ exists FVARX armbruch(FVARX) }

Klassifikationsinstanz ARMBRUCH_PS={ ∼ }
log={}

anfrage=armbruch(rudi);
Gewünschtes Ergebnis: MUM
Ergebnis: MUM
log={}

b) user: adalat_jcqe
schema: jcqe_normal
potsec={ exists FVARX armbruch(FVARX) }
Klassifikationsinstanz ARMBRUCH_PS={ ∼ }
log={}

anfrage=armbruch(hans);
Gewünschtes Ergebnis: MUM
Ergebnis: MUM
log={}

c) user: adalat_jcqe
schema: jcqe_normal
potsec={ exists FVARX armbruch(FVARX) }
Klassifikationsinstanz ARMBRUCH_PS={ ∼ }
log={}

anfrage=exists X armbruch(X);
Gewünschtes Ergebnis: exists X armbruch(X)
Ergebnis: exists X armbruch(X)
log={exists X armbruch(X)}

15.12 ToDos:

1. Aus folgendem Grund konnten wir den Testfall 5 mit dem Prototyp nicht testen. Die minimizePotSec() Methode des Prototyps enthält den folgenden Fehler:

   - *bluemer_Potsec* = { exists X reha (X, b, c, d) }
   - füge psi= 'exists Y reha (Y, b, c, d)' ein:
   - Ergebnis des Redundanztests von PG495 mit Hilfe Prover9:
     bluemer_Potsec = { }

   Das führt dazu, dass **"füge psi2=reha (X, b, c, d) ein"** auch nicht richtig funktioniert, da wir psi2 wegen der Approximation durch **exists FVARX reha(FVARX, b, c, d)** ersetzen.

2. Falls die Mengen der potentiellen Geheimnisse freie Variablen enthalten, wird im dynamischem Modus eine Approximation angewendet, so dass die freien Variablen durch gebundene Variablen ersetzt werden. In diesem Fall sollte der User benachrichtigt werden, dass seine Anfragen approximiert behandelt werden.

Die minimizePotSec() Methode des Prototyps enthält den folgenden Fehler:

bluemer_Potsec = { exists X reha (X, b, c, d) }

füge psi= 'exists Y reha (Y, b, c, d)' ein:

Ergebnis des Redundanztests von PG495 mit Hilfe Prover9:

bluemer_Potsec = { }

Das führt dazu, dass **"füge psi2=reha (X, b, c, d) ein"** auch nicht richtig funktioniert, da wir psi2 wegen der Approximation durch **exists FVARX reha(FVARX, b, c, d)** ersetzen.

Kapitel 16

Problemliste für die alten Prototypen

16.1 Überblick über Probleme der alten Prototypen

Während des Testens der beiden Prototypen - Prototyp der PG495 und Prototyp von Sebastian Sonntag- sind uns einige Probleme aufgefallen. Die Fehler wurden von uns dokumentiert und behoben.

Bei den Prototypen wurde die Benutzeroberfläche, die kontrollierte Anfrageauswertung und die Funktionalität der verschiedenen Zensoren überprüft . Das erste Problem, welches uns aufgefallen war, war die Tatsache, dass die beide Prototypen auf einem "Einbenutzersystem" basierten. Dies führt zu dem Problem, dass nur ein Benutzer, und zwar nur Bluemer, sich an die Prototypen anmelden konnte. Wenn mehrere Benutzer sich anmelden würden, würden sie sich dasselbe Benutzerwissen(*Log*) und dieselbe potentielle Geheimnisse(*pot_sec*) teilen. D.h, wenn immer einer dieser Benutzer etwas aus dem gemeinsamen Log(des Benutzers Bluemer) löschte bzw. einfügte wurde dies die Arbeit anderer angemeldeter Benutzer beeinflusst, da die Anderen benutzen doch dasselbe Log. Diese PG hat den alten Prototypen um ein "Mehrbenutzersystem" erweitert und somit das Benutzerwaltungsproblem gelöst.

Ein zweites Problem war bei dem Kursor von ResultSet-Objekten. Ein ResultSet-Objekt enthält alle Zeilen der Ergebnistabelle einer SQL-Anfrage und wird von einer executeQuery-Anweisung erzeugt, falls eine Ausgabe geliefert wird. Es besitzt einen Cursor, der auf die aktuelle Zeile verweist und anfänglich vor der ersten Zeile posi-

tioniert ist. Mit der Methode *next*() wird der Cursor um eine Zeile weitergesetzt und liefert die Fehlermeldung (wörtlich) "Maximal Anzahl offener Cursor überschritten" zurück, wenn er sich schließlich hinter der letzten Zeile befindet. Nach dieser Fehlermeldung, würde das Log (auf dem eigenen Rechner) geleert und es wären keine weiteren Anfragen mehr möglich. Durch einen Neustart würde das Problem vorübergehend behoben. Dieses Problem ist jetzt behoben worden und diese Fehlermeldung taucht nicht mehr auf.
Bei dem Testen beider Prototypen haben wir festgestellt, dass manche Zensoren nicht richtig funktionieren. Alle diese Fehler wurden dokumentiert und behoben.

Das folgende Bild stellt dar, was für Probleme bei der Anmeldung an die Prototypen auftauchten:

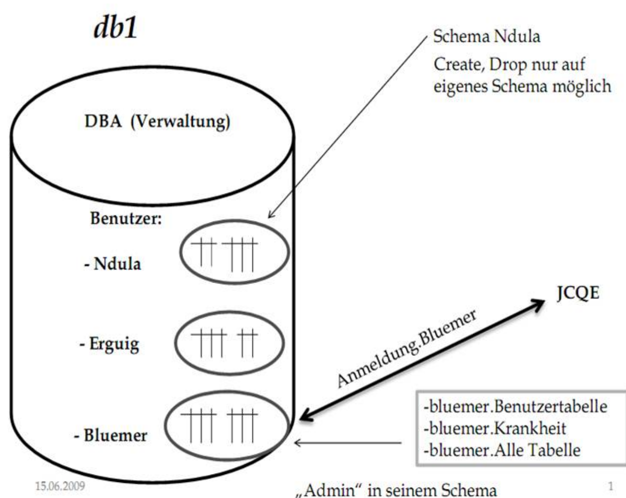


Abbildung 16.1: Benutzerverwaltungsproblem

16.1.1 Wie wurden diese Fehler behoben?

Um das Anmeldungsproblem zu beseitigen haben wir zwei neue Schemata angelegt: jcqe_normal, für Benutzer mit entweder dem Ablehnungs- oder dem kombinierten Zensor und *jcqe_lying* für Benutzer mit dem Lügenzensor. Sobald ein neuer Benutzer angelegt wird, wird ein Oracle-Account für ihn erstellt und die Relationen $< Benutzername >Log$ und $< Benutzername > Pot_sec$ werden je nach seinem

Zensor in dem Schema(*Benutzername_normal*/*Benutzername_lying*) in der Datenbank angelegt. Diese Tabellen dienen der Speicherung des Benutzerwissens bzw. der potentiellen Geheimnisse des neuen Benutzers. Da wir jetzt ein "Mehrbenutzersystem" haben, kann das Benutzverwaltungsproblem nicht mehr auftauchen, weil jeder Benutzer sein eigenes Schema und seine eigenen Tabellen für *log* und *pot_sec* hat.

Die PG495 hat in manchen Methoden im Quellcode vergessen die Cursor wieder zu schließen. Wir haben deshalb im Quellcode gesucht und die geöffnete Cursor mit dem Befehl *rs.closed*() geschlossen. Dies hat das Problem behoben.

16.2 Testen

In dieser Testserie testen wir die Funktionalität der verschiedenen Zensoren. Wir wollen hier testen, ob die gestellten Anfragen Geheimnisse implizieren. Die Datenbankeinträge für diesen Testverlauf sind die von der PG495 erstellten Tabellen.

1. Tabelle Beinbruch:
   CREATE TABLE "BLUEMER"."BEINBRUCH"("NAME"VARCHAR(30));

2. BluemerLog:
   CREATE TABLE "BLUEMER"."BLUEMERLOG"("NAME"VARCHAR(4000));

BluemerLog	Formel
	.
	.
	.
	.

   Jeder Benutzer hat sein eigenes Benutzerlog. Im Benutzerlog können Formeln wie z.B. armbruch(lena) oder beinbruch(hans) stehen.

3. Tabelle ⟨Benutzer⟩POTSEC:

   CREATE TABLE "BLUEMER"."BLUEMERPOTSEC"("NAME"VARCHAR2(4000));

BluemerPotsec	**Formel**
	.
	.
	.
	.

Jeder Benutzer hat seine eigene Sicherheitspolitik.

4. Tabelle Armbruch:
   CREATE TABLE "BLUEMER".“ARMBRUCH"("NAME"VARCHAR(30));
   mit folgendem Inhalt:

Armbruch	**Name**
	alfred
	bjoern
	christian
	claudia
	daniel
	gabriele
	hans
	jan
	joachim
	lena
	manuela
	marcel
	martin
	matthias
	michael
	nils
	ralf
	sandra
	sebastian
	thomas
	torbenb
	yu

Nach dem Wechsel des Zensors haben wir immer mit einem leeren Log angefangen.

16.2.1 Prototyp der PG495

Test des Lügenzensors

Benutzer: Bluemer
Rolle: Administrator

Wir testen, ob der Prototyp syntaktische Fehler richtig erkennt und ob die Zensoren einwandfrei funktionieren. Folgende Tests wurden durchgeführt:

Testfall 1a:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : armbruch(hans)
log : -
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 1b:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : -
Ausgabe : ((not armbruch(hans)) or (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 1c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : armbruch(hans)
log : armbruch(lena)
Ausgabe : (not armbruch(hans))
Bemerkung: Ausgabe korrekt.

Testfall 1d:
Schemadesign: ONF

pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : armbruch(lena)
Ausgabe : ((not armbruch(hans)) or (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 1e:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : (not armbruch(hans))
Ausgabe : ((not armbruch(hans)) or (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 1f:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (not(not armbruch(hans)))
log : -
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 2a:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : armbruch(lena)
log : -
Ausgabe : (not armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testfall 2b:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : (armbruch(hans) or armbruch(lena))
log : -
Ausgabe : ((not armbruch(hans)) and (not armbruch(lena)))

Bemerkung: Ausgabe korrekt.

Testfall 2c:
Schemadesign: ONF
pot_sec: armbruch(hans),armbruch(lena)
query : armbruch(hans)
log : (armbruch(hans) or armbruch(lena))
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Wir testen mit Existenzquantoren. Vor dem Testen mit Existenzquantoren leeren wir zuerst das Log.

Testfall 3a:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X (armbruch(X) and armbruch(hans))
log : -
Ausgabe : forall X ((not armbruch(X)) or (not armbruch(hans)))
Bemerkung: Ausgabe korrekt.

Testfall 3b:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X armbruch(X)
log : -
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3c:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X (armbruch(X) and armbruch(lena))
log : -
Ausgabe : exists X (armbruch(X) and armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testfall 3d:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X (armbruch(X) and armbruch(hans))
log : armbruch(lena)
Ausgabe : forall X ((not armbruch(X)) or (not armbruch(hans)))
Bemerkung: Ausgabe korrekt.

Wir testen hier, ob der Prototyp Doppel-Negationen fehlerfrei behandelt. Vor diesem Test wurde das Log erstmal geleert.

Testfall 4:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(hans));
log : -
Ausgabe : (not armbruch(hans))
Bemerkung: Ausgabe korrekt.

Testergebnis:
Der Lügenzensor funktionierte einwandfrei. Es wurden keine Geheimnisse verraten und auch Anfragen mit Existenzquantoren und Doppel-Negationen wurden richtig behandelt.

Test des Ablehnungszensors

Benutzer: Bluemer
Rolle: Administrator
Wir testen, ob die Anfragen Geheimnisse implizieren. Wir fangen mit einem leeren Log an.

Testfall 1a:
Schemadesign: ONF
pot_sec: armbruch(hans)

query : armbruch(hans)
log : -
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 1b:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : armbruch(hans)
log : -
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 1c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : armbruch(hans)
log : armbruch(lena)
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 1d:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : armbruch(lena)
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 1e:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : -
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 2a:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : armbruch(lena)
log : -
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 2b:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : (armbruch(hans) or armbruch(lena))
log : -
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 2c:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : (not armbruch(hans))
log : -
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Wir testen mit Existenzquantoren. Vor diesem Test wurde das Log geleert.

Testfall 3a:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : exists X armbruch(X)
log : -
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3b:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : exists X (armbruch(X) and armbruch(lena))
log : -
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 3c:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : exists X (armbruch(X) and armbruch(peter))
log : -
Ausgabe : forall X ((not armbruch(X)) or (not armbruch(peter)))
Bemerkung1: *Peter* steht nicht in der Armbruchtabelle.
Bemerkung2: Ausgabe korrekt.

Testfall 3d:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : exists X armbruch(X)
log : armbruch(jan)
Ausgabe : MUM
Bemerkung: Ausgabe nicht korrekt.
Gewünschte Ausgabe:
Ausgabe : exists X armbruch(X)

Testfall 3e:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)
log : -
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3f:
Schemadesign: ONF

pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X (armbruch(X) and armbruch(lena))
log : armbruch(jan)
Ausgabe : exists X (armbruch(X) and armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testfall 3g:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X (armbruch(X) and armbruch(lena))
log : armbruch(hans)
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 3h:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X armbruch(X)
log : armbruch(lena)
Ausgabe : MUM
Bemerkung: Ausgabe nicht korrekt.
Gewünschte Ausgabe:
Ausgabe : exists X armbruch(X)

Testfall 3i:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X armbruch(X)
log : exists X armbruch(X)
Ausgabe : MUM
Bemerkung: Ausgabe nicht korrekt.
Gewünschte Ausgabe:
Ausgabe : exists X armbruch(X)

Testfall 3j:
Schemadesign: ONF
pot_sec: armbruch(hans),armbruch(lena),armbruch(alfred)

query : exists X ((not armbruch(X)) and armbruch(hans))
log : (armbruch(hans) or armbruch(lena))
Ausgabe : "Gebundene Variable ist nicht range restricted"**Bemerkung:** Negationen sind nicht in Existenzquantoren erlaubt. Diese wurde von der PG495 implementiert und die PG536 hat diese Idee übernommen. Die Ausgabe ist deshalb nicht fehlerhaft.

Testfall 3k:
Schemadesign: ONF
pot_sec: armbruch(hans),armbruch(lena),armbruch(alfred)
query : exists X krankheit(X,husten)
log : -
Ausgabe : forall X (not krankheit(X,husten))
Bermerkung: die Konstante, „husten", existiert nicht in unserer Datenbank.
Bemerkung: Ausgabe korrekt.

Testfall 3l:
Schemadesign: ONF
pot_sec: armbruch(hans),armbruch(lena),armbruch(alfred)
query : exists X krankheit(X,cold)
log : -
Ausgabe : exists X krankheit(X,cold)
Bemerkung: Ausgabe korrekt.

Wir testen hier, ob der Prototyp Doppel-Negationen fehlerfrei behandeln kann. Vor dem Test mit Doppel-Negationen wurde das log geleert.

Testfall 4a:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (not(not armbruch(hans)))
log : -
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 4b:

Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(hans)))
log : -
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 4c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (not(not armbruch(hans)))
log : armbruch(hans)
Ausgabe : MUM
Bemerkung: Ausgabe nicht korrekt.
Gewünschte Ausgabe:
Ausgabe : armbruch(hans)

Testfall 4d:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(lena)))
log : armbruch(lena)
Ausgabe : MUM
Bemerkung: Ausgabe nicht korrekt.
Gewünschte Ausgabe:
Ausgabe : armbruch(lena)

Testfall 4e:
Schemadesign: ONF
pot_sec: -
query : armbruch(lena)
log : armbruch(lena)
Ausgabe : MUM
Bemerkung: Ausgabe nicht korrekt.
Gewünschte Ausgabe:
Ausgabe : armbruch(lena)

Testfall 4f:
Schemadesign: ONF
pot_sec: armbruch(lena),armbruch(hans),armbruch(alfred)
query : armbruch(lena)
log : (armbruch(lena) or armbruch(hans))
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testergebnis:
Es wurden keine Geheimnisse verraten und auch Anfragen mit Existenzquantoren und Doppel-Negationen wurden richtig behandelt. Auf dem ersten Blick scheinen die Fälle 3d, 3h, 3i, 4c, 4d und 4e fehlerhaft zu sein. Dies liegt an der Arbeitsweise des normalen Ablehnungszensors. Dieser arbeitet wie folgt:
if((Log $\cup \{eval^*(\phi)(db)\} \models \psi$) oder (Log $\cup \{\neg(eval^*(\phi)(db)\}) \models \psi$)) für ein $\psi \in$ *pot_sec*, then **return MUM**
else $\{eval^*(\phi)(db)\}$.
Es gibt aber einen speziellen Fall, wobei der normale Ablehnungszensor verweigert, weil das Log das Ergebnis der Anfrage impliziert (die Fälle 3d, 3h, 3i, 4c, 4d und 4e). D.h. Log $\models eval^*(\phi)(db)$.

Wie kann diese Situation vermieden werden? Die PG495 hat den Improved Refusal Censor nicht implementiert. Dieser Zensor arbeitet anders als der normale Ablehnungszensor, nämlich: Wenn das Ergebnis einer Anfrage das Log impliziert, soll das Ergebnis ausgegeben werden, andernfalls wird der normale Ablehnungszensor gewählt.
Der verbesserte Ablehnungszensor wurde von der PG536 (s.u. 16.3) implementiert, um diese Situation zu vermeiden.

Test des kombinierten Zensors

Benutzer: Bluemer
Rolle: Administrator

Wir testen, ob diese Anfragen Geheimnisse implizieren. Wir fangen mit einem leeren Log an.

Testfall 1a:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : armbruch(lena)
log : -
Ausgabe : (not armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testfall 1b:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : (armbruch(hans) or armbruch(lena))
log : -
Ausgabe : ((not armbruch(hans)) and (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 1c:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : -
Ausgabe : ((not armbruch(hans)) or (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 2a:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : armbruch(lena)
log : armbruch(hans)
Ausgabe : (not armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testfall 2b:

Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
pquery : (not armbruch(hans))
log : armbruch(hans),(not armbruch(lena))
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 2c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : -
Ausgabe : ((not armbruch(hans)) or (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Wir testen mit Existenzquantoren. Wir fangen mit einem leeren Log an .

Testfall 3a:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)
log : -
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3b:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)
log : armbruch(jan)
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))

query : exists X armbruch(X)
log : armbruch(hans)
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3d:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : exists X (armbruch(X) and armbruch(lena))
log : -
Ausgabe : forall X ((not armbruch(X)) or (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 3e:
Schemadesign: ONF
pot_sec: armbruch(hans),armbruch(lena),armbruch(alfred)
query : exists X ((not armbruch(X)) and armbruch(hans))
log : (armbruch(hans) or armbruch(lena))
Ausgabe : "Gebundene Variable ist nicht range restricted"
Bemerkung: Ausgabe is korrekt, da Negationen in Anfragen mit Existenzquantoren nicht erlaubt sind.

Testfall 3f:
Schemadesign: ONF
pot_sec: armbruch(hans),armbruch(lena),armbruch(alfred)
query : exists X krankheit(X,husten)
log : (armbruch(hans) or armbruch(lena))
Ausgabe : forall X (not krankheit(X,husten))
Bemerkung: Ausgabe korrekt.

Testfall 3g:
Schemadesign: ONF
pot_sec: armbruch(hans),armbruch(lena),armbruch(alfred)
query : exists X krankheit(X,husten)
log : -
Ausgabe : forall X (not krankheit(X,husten))
Bermerkung: die Konstante "husten" existiert nicht in unserer Datenbank.

Bemerkung: Ausgabe korrekt.

Testfall 3h:
Schemadesign: ONF
pot_sec: armbruch(hans),armbruch(lena),armbruch(alfred)
query : exists X krankheit(X,cold)
log : -
Ausgabe : exists X krankheit(X,cold)
Bemerkung: Ausgabe korrekt.

Wir testen hier, ob der Prototyp Doppel-Negationen fehlerfrei behandelt. Wir fangen mit einem leeren Log an .

Testfall 4a:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (not(not armbruch(hans)))
log : -
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 4b:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : (not(not armbruch(hans)))
log : -
Ausgabe : (not armbruch(hans))
Bemerkung: Ausgabe korrekt.

Testfall 4c:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(hans)))
log : -
Ausgabe : (not armbruch(hans))
Bemerkung: Ausgabe korrekt.

Testfall 4d:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(lena)))
log : armbruch(lena)
Ausgabe : armbruch(lena)
Bemerkung: Ausgabe korrekt.

Testfall 4e:
Schemadesign: ONF
pot_sec: -
query : (not(not armbruch(lena)))
log : armbruch(lena)
Ausgabe : armbruch(lena)
Bemerkung: Ausgabe korrekt.

Testfall 4f:
Schemadesign: ONF
pot_sec: armbruch(hans), armbruch(lena)
query : armbruch(hans)))
log : (armbruch(hans) or armbruch(lena))
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testen mit offenen Anfragen.

Testfall 5a:
Schemadesign: ONF
pot_sec: armbruch(alfred), armbruch(yu)
query : armbruch(X)
log : -
Ausgabe : „Ein Wort ist nicht im Verzeichnis vorhanden“
Bemerkung: *Offene Anfragen waren mit der Relation „Armbruch“ nicht möglich. Die PG536 hat aber diesen Fehler teilweise gelöst. Im aktuellen Prototypen (PG536) wurde die optimierte offene Anfrage implementiert. Ferner sind jetzt optimierte offene Anfrage nicht nur mit dem kombinierten Zensor zu stellen sondern auch mit*

dem Verweigerungszensor möglich. Gibt es allerdings noch keinen Verweigerungszensor für offene Anfragen, die nicht optimiert behandelt werden künnen. Nach der Implementierung haben wir den Prototyp erneut getestet. Die Fälle(5a und 5b), in denen die Testergebnisse einen Fehler aufdeckten, sind nun richtig.

Testfall 5b:
Schemadesign: ONF
pot_sec: armbruch(alfred), armbruch(yu)
query : armbruch(X)
log : armbruch(hans)
Ausgabe : „Ein Wort ist nicht im Verzeichnis vorhanden“
Ausgabe: armbruch(hans)

Testfall 5c:
Schemadesign: ONF
pot_sec: beinbruch(alfred), beinbruch(yu)
query : beinbruch(X)
log : -
Ausgabe : beinbruch(gerd), (not beinbruch(alfred)), forall X (equals(X,alfred) or (equals(X,gerd) or (not beinbruch(X))))
Bemerkung: Ausgabe korrekt.

Testfall 5d
Schemadesign: ONF
pot_sec: armbruch(hans),armbruch(lena),armbruch(alfred)
query : krankheit(X,cold)
log : exists X krankheit(X,cold)
Ausgabe : krankheit(yu,cold), forall X (equals(X,alfred) or (equals(X,gerd) or (equals(X,hans) or (equals(X,frank) or (equals(X,yu) or (not krankheit(X,cold))))))), krankheit(alfred,cold), (notkrankheit(gerd,cold)), (not krankheit(hans,cold)), (not krankheit(frank,cold))
Bemerkung: Ausgabe korrekt.

Testergebnisse nach der Implementierung:

Testfall 5a:
Schemadesign: ONF
pot_sec: armbruch(alfred), armbruch(yu)
query : armbruch(X)
log : -
Ausgabe: armbruch(dumpfbacke), armbruch(gustav), armbruch(hans), armbruch(jude), armbruch(lena), armbruch(ndula), armbruch(peter), armbruch(susan), armbruch(wolfgang).

textbfTestfall 5b:
Schemadesign: ONF
pot_sec: armbruch(alfred), armbruch(yu)
query : armbruch(X)
log : armbruch(hans)
Ausgabe: armbruch(hans)

Testergebnis:
Der kombinierte Zensor ist in Ordnung. Es werden keine Geheimnisse verraten und auch offene Anfragen mit Existenzquantoren und Doppel-Negationen wurden richtig behandelt.

16.2.2 Sonntag_JCQE

Die folgenden Informationen werden von dem Prototyp von Sonntag bei der Ausgabe geliefert:

- Schemadesign: ONF
  Zeigt, ob das Datenbankschema in ONF vorliegt oder nicht. Bisher muss der Benutzer dies bei der Anmeldung allerdings noch selbst angeben, weil der Prototyp es nicht selbst erkennen kann.

- Case : 3 -(stat. IK)
  Es gibt elf Optimierungsfälle (cases). Die ersten Fünf arbeiten mit statischer Inferenzkontrolle und benötigen kein Benutzerwissen, während die restlichen sechs Fälle mit der dynamischen Inferenzkontrolle arbeiten und das Benutzerwissen benötigen.
- isFactPotsecSet: true
  Gibt true aus, falls die Geheimnismenge nur Fakten enthält.
- PotsecSetLanguage : L_ps_nDS
  Die von der PG495 festgelegte Sprache der Sicherheitspolitik.
- QueryLanguage : Lstar_q_KI
  Die von der PG495 festgelegte Sprache der Anfragesprache.

Die Eingaben Pot_sec, Log, Query und Ausgabe sind von uns eingegeben.

- *Pot_sec* bezeichnet die Menge der potenziellen Geheimnisse, die geschützt werden sollen.
- *Query* ist die gestellte Anfrage und
- *Log* ist das Benutzerwissen.

Die gewählten Testfälle reichen aus, um die Funktionalität der Zensoren zu testen, und sollen die möglichen Schwachstellen der Prototypen aufdecken zu können.

Test des Lügenzensors

Benutzer: Bluemer
Rolle: Administrator

Wir testen, ob die Anfragen Geheimnisse implizieren. Wir fangen mit einem leeren Log an .

Testfall 1a:
Schemadesign: ONF
pot_sec : (armbruch(hans) and armbruch(lena))
query : armbruch(hans)
log : -
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 1b:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : -
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : ((not armbruch(hans)) or (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 1c:
Schemadesign: ONF
pot_sec : (armbruch(hans) and armbruch(lena))
query : armbruch(hans)

log : armbruch(lena)
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : (not armbruch(hans))
Bemerkung: Ausgabe korrekt.

Testfall 1d:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : armbruch(lena)
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : ((not armbruch(hans)) or (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 1e:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : (not armbruch(hans))
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : ((not armbruch(hans)) or (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 2a:
Schemadesign: ONF
pot_sec : (armbruch(hans) or armbruch(lena))

query : armbruch(lena)
log : -
Case : 3-(stat. IK)
isFactPotsecSet: true
PotsecSetLanguage : Lstar_ps_nDS
QueryLanguage : Lstar_q_KI
Ausgabe : (not armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testfall 2b:
Schemadesign: ONF
pot_sec : (armbruch(hans) or armbruch(lena))
query : (armbruch(hans) or armbruch(lena))
log : -
Case : 7 -(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : Lstar_ps_nDS
QueryLanguage : L_q_PG
Ausgabe : ((not armbruch(hans)) and (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Wir testen mit Existenzquantoren. Wir fangen mit einem leeren Log an .

Testfall 3a:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X (armbruch(X) and armbruch(hans))
log : -
Case : NOOPT-(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_nDS
QueryLanguage : Lstar_q_nDS
Ausgabe : forall X ((not armbruch(X)) or (not armbruch(hans)))
Bemerkung: Ausgabe korrekt.

Testfall 3b:

Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X armbruch(X)
log : -
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_nDS
QueryLanguage : Lstar_q_nDS
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3c:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X (armbruch(X) and armbruch(lena))
log : -
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_nDS
QueryLanguage : Lstar_q_nDS
Ausgabe : exists X (armbruch(X) and armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testfall 3d:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X (armbruch(X) and armbruch(hans))
log : armbruch(lena)
Case : NOOPT -(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_nDS
Ausgabe : forall X ((not armbruch(X)) or (not armbruch(hans)))
Bemerkung: Ausgabe korrekt.

Wir testen hier, ob der Prototyp Doppel-Negationen fehlerfrei behandelt. Wir fangen mit einem leeren Log an .

Testfall 4a:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(hans));
log : -
Case : NOOPT-(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_nDS
QueryLanguage : L_q_PG
Ausgabe : (not armbruch(hans))
Bemerkung: Ausgabe korrekt.

Testfall 4b:
Schemadesign: ONF
pot_sec: -
query : armbruch(hans);
log : -
Case : 3-(stat. IK)
isFactPotsecSet: true
PotsecSetLanguage : -
QueryLanguage : L_q_KI
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 4c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (not(not armbruch(hans)))
log : -
Case : 7 -(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : L_q_PG

Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testergebnis:
Der Lügenzensor funktioniert einwandfrei und keine Geheimnisse wurden verraten.

Test des Ablehnungszensors

Benutzer: Bluemer
Rolle: Administrator

Wir testen, ob die Anfragen Geheimnisse implizieren. Wir fangen mit einem leeren Log an .

Testfall 1a:
Schemadesign: ONF
pot_sec : armbruch(hans)
query : armbruch(hans)
log : -
Case : 3-(stat. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 1b:
Schemadesign: ONF
pot_sec : (armbruch(hans) and armbruch(lena))
query : armbruch(hans)
log : -
Case : NOOPT- (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG

QueryLanguage : Lstar_q_KI
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 1c:
Schemadesign: ONF
pot_sec : (armbruch(hans) and armbruch(lena))
query : armbruch(hans)
log : armbruch(lena)
Case : NOOPT- (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 1d:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : armbruch(lena)
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 1e:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : -
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG

QueryLanguage : Lstar_q_KI
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 2a:
Schemadesign: ONF
pot_sec : (armbruch(hans) or armbruch(lena))
query : armbruch(lena)
log : -
Case : 4-(stat. IK)
isFactPotsecSet: true
PotsecSetLanguage : Lstar_ps_nDS
QueryLanguage : Lstar_q_KI
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 2b:
Schemadesign: ONF
pot_sec : (armbruch(hans) or armbruch(lena))
query : (armbruch(hans) or armbruch(lena))
log : -
Case : 7 -(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : Lstar_ps_nDS
QueryLanguage : L_q_PG
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Wir testen mit Existenzquantoren. Wir fangen mit einem leeren Log an .

Testfall 3a:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)

log : -
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3b:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)
log : armbruch(jan)
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_nDS
Ausgabe : MUM
Bemerkung: Ausgabe nicht korrekt.
Gewünschte Ausgabe:
Ausgabe : exists X armbruch(X)

Testfall 3c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)
log : armbruch(hans)
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_nDS
Ausgabe : MUM
Bemerkung: Ausgabe nicht korrekt.
Gewünschte Ausgabe:
Ausgabe : exists X armbruch(X)

Testfall 3d:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X armbruch(X)
log : armbruch(lena)
Case : 3-(stat. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_nDS
QueryLanguage : Lstar_q_nDS
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Wir testen hier, ob der Prototyp Doppel-Negationen fehlerfrei behandelt. Wir fangen mit einem leeren Log an .

Testfall 4a:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(hans)))
log : -
Case : 6-(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_nDS
QueryLanguage : L_q_PG
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 4b:
Schemadesign: ONF
pot_sec : armbruch(hans), armbruch(lena), armbruch(alfred)
query : armbruch(lena)
log : -
Case : 3-(stat. IK)
isFactPotsecSet: true
PotsecSetLanguage : Lstar_ps_nDS

QueryLanguage : Lstar_q_KI
Ausgabe : MUM
Bemerkung: Ausgabe korrekt.

Testfall 4c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (not(not armbruch(hans)))
log : armbruch(hans)
Case : NOOPT-(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : L_q_PG
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Testfall 4d:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(lena)))
log : armbruch(lena)
Case : 7-(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_nDS
QueryLanguage : L_q_PG
Ausgabe : MUM
Bemerkung: Ausgabe nicht korrekt.
Gewünschte Ausgabe:
Ausgabe : armbruch(lena)

Testfall 4e:
Schemadesign: ONF
pot_sec: -
query : (not(not armbruch(lena)))
log : armbruch(lena)
Case : 3-(stat. IK)

isFactPotsecSet: true
PotsecSetLanguage : L_ps_nDS
QueryLanguage : L_q_PG
Ausgabe : armbruch(lena)
Bemerkung: Ausgabe korrekt.

Testergebnis: Siehe Testergebnis des Ablenungszensors der PG495

Testergebnisse nach der Änderung

Nachdem wir die neue Klasse ImprovedRefusalCensor angelegt haben, haben wir den Prototyp erneut getestet. Die Fälle, in denen die Testergebnisse falsch aufgewiesen wurden, sind nun richtig.

Testfall 3b:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)
log : armbruch(jan)
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_nDS
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)
log : armbruch(hans)
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG

QueryLanguage : Lstar_q_nDS
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 4d:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(lena)))
log : armbruch(lena)
Case : 7-(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_nDS
QueryLanguage : L_q_PG
Ausgabe : armbruch(lena)
Bemerkung: Ausgabe korrekt.

Test des kombinierten Zensors

Benutzer: Bluemer
Rolle: Administrator

Wir testen, ob die Anfragen Geheimnisse implizieren. Wir fangen mit einem leeren Log an.

Testfall 1a:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : armbruch(lena)
log : -
Case : NOOPT-(dynam. IK)
isFactPotsecSet: true

PotsecSetLanguage : Lstar_ps_nDS
QueryLanguage : Lstar_q_KI
Ausgabe : (not armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testfall 1b:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : (armbruch(hans) or armbruch(lena))
log : -
Case : 7-(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : Lstar_ps_nDS
QueryLanguage : Lstar_q_PG
Ausgabe : ((not armbruch(hans)) and (not armbruch(lena)))
Bemerkung: Ausgabe korrekt.

Testfall 1c:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : (armbruch(hans) and armbruch(lena))
log : -
Case : 4-(stat. IK)
isFactPotsecSet: true
PotsecSetLanguage : Lstar_ps_nDS
QueryLanguage : Lstar_q_KI
Ausgabe : Mum
Bemerkung: Ausgabe korrekt.

Testfall 2a:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : armbruch(lena)
log : armbruch(hans)
Case : NOOPT-(dynam. IK)

isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage: L_q_KI
Ausgabe : (not armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testfall 2b:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (not armbruch(hans))
log : armbruch(hans),(not armbruch(lena))
Case : NOOPT-(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage: Lstar_q_KI
Ausgabe : armbruch(hans)
Bemerkung: Ausgabe korrekt.

Wir testen mit Existenzquantoren. Wir fangen mit einem leeren Log an .

Testfall 3a:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)
log : -
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_nDS
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3b:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)

log : armbruch(jan)
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_nDS
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X armbruch(X)
log : armbruch(hans)
Case : NOOPT - (dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage : Lstar_q_KI
Ausgabe : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.

Testfall 3d:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : exists X (armbruch(X) and armbruch(lena))
log : -
Case : 4-(stat. IK)
isFactPotsecSet: true
PotsecSetLanguage : Lstar_ps_nDS
QueryLanguage: Lstar_q_nDS
Ausgabe : exists X armbruch(X), armbruch(lena) ==> MUM
Bemerkung: Ausgabe korrekt.

Testfall 3e:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : exists X (armbruch(X) and armbruch(lena))

log : -
Case : NOOPT-(dynam. IK)
isFactPotsecSet: true
PotsecSetLanguage : L_ps_PG
QueryLanguage: Lstar_q_nDS
Ausgabe : exists X (armbruch(X) and armbruch(lena))
Bemerkung: Ausgabe korrekt.

Testergebnis:
Der kombinierte Zensor funktioniert richtig. Aber in den Testfällen 1c und 3d, arbeitete der kombinierte Zensor im statischen Modus. Es darf jedoch nur der Ablehnungszensor im statischen Modus arbeiten. Dieser Fehler wurde von der PG536 behoben. Nach erneuten Testen arbeitet der kombinierte Zensor wieder im dynamischen Modus.

16.3 Behebung der Fehler in den Prototypen

16.3.1 Optimierter Ablehnungszensor

Da der Ablehnungszensor nach den Testen 16.2 nicht richtig funktioniert, wird nun der optimierte Ablenungszensor “Improved Refusal Censor“ implementiert, der die Aufgabe hat

```
1     if (log ⊨ {eval*(φ)(db)})
2         then {eval*(φ)(db)};
3     else normal_refusal_sensor};
```

Die ImprovedRefusalCensor-Klasse ist eine Erweiterung der RefusalCensor-Klasse. Dabei wird die Methode “CType classify(LinkedList<Formula> log, Formula query, LinkedList<Formula> potSec)“ überschrieben, um die logische Implikation $log \models \{eval^{*}(\phi)(db)\}$ zuerst zu überprüfen. Falls die Implikation “false“ liefert, dann wird auf die Methoden des normalen Ablehnungszensors zugegriffen. Ansonsten wird “UNCRITICAL“ zurückgegeben.

```
1 public class ImprovedRefusalCensor extends RefusalCensor {
2   public CType classify(LinkedList<Formula> log, Formula query,
        LinkedList<Formula> potSec) throws Exception {
```

```
3     boolean ergebnis = false;
4     tpInterface = new ProverNine();
5     LinkedList<Formula> mod = new LinkedList<Formula>();
6     mod.add(query);
7     ergebnis = tpInterface.isImplied(log, null, mod);
8     if (ergebnis) return CType.UNCRITICAL;
9       else
10      return super.classify(log, query, potSec);
11  }
12 }
```

16.3.2 Testergebnisse nach der Änderung

Nachdem wir die neue Klasse ImprovedRefusalCensor angelegt haben, haben wir den Prototyp erneut getestet. Die Fälle, in denen die Testergebnisse einen Fehler aufdeckten wurden, sind nun richtig.

Testfall 3d:
Schemadesign: ONF
pot_sec: (armbruch(hans) or armbruch(lena))
query : exists X armbruch(X)
log : armbruch(jan)
Bemerkung: Ausgabe korrekt.
Ausgabe : exists X armbruch(X)

Testfall 3h:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X armbruch(X)
log : armbruch(lena)
Bemerkung: Ausgabe korrekt.
Ausgabe : exists X armbruch(X)

Testfall 3i:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : exists X armbruch(X)

log : exists X armbruch(X)
Bemerkung: Ausgabe korrekt.
Ausgabe : exists X armbruch(X)

Testfall 4c:
Schemadesign: ONF
pot_sec: (armbruch(hans) and armbruch(lena))
query : (not(not armbruch(hans)))
log : armbruch(hans)
Bemerkung: Ausgabe korrekt.
Ausgabe : armbruch(hans)

Testfall 4d:
Schemadesign: ONF
pot_sec: armbruch(hans)
query : (not(not armbruch(lena)))
log : armbruch(lena)
Bemerkung: Ausgabe korrekt.
Ausgabe : armbruch(lena)

Testfall 4e:
Schemadesign: ONF
pot_sec: -
query : armbruch(lena)
log : armbruch(lena)
Bemerkung: Ausgabe korrekt.
Ausgabe : armbruch(lena)

Kapitel 17

Neues Datenbankschema

Für ein besseres Testen der Funktionalitäten von unserem Prototypen haben wir uns entschieden, eine neue Datenbank zu entwickeln, die im Gegensatz zu der alten auch mehrstellige Prädikate unterstützt. Eine Entity-Relationship (ER)-Modellierung ist ein wichtiger Schritt beim Informationssystem-Design und Software-Engineering. Er stellt die reale Welt als eine Reihe von grundlegenden Objekten (Entitäten) und Beziehungen zwischen diesen Objekten dar.

17.1 Entity-Relationship-Modell

In unserem Entity-Relationship-Modell (oder Gegenstands-Beziehungs-Modell), das ein konzeptuelles Modell darstellt, besteht ein Krankenhaus aus mehreren Stationen. Auf einer Station befinden sich mehrere Zimmer. Dabei kann es vorkommen, dass nicht immer alle Zimmer mit Patienten belegt sind. Im Laufe der Krankengeschichte eines Patienten kann es natürlich auch vorkommen, dass ein Patient auf unterschiedlichen Zimmern lag.

Zu beachten ist auch, dass ein Patient nicht unbedingt unter einer Krankheit leidet und auf einem Zimmer liegen muss.

Auf den Stationen des Krankenhauses arbeiten Ärzte, wobei ein Arzt auch mehreren Stationen zugeordnet sein kann. Die Ärzte behandeln die Patienten. Dabei kann

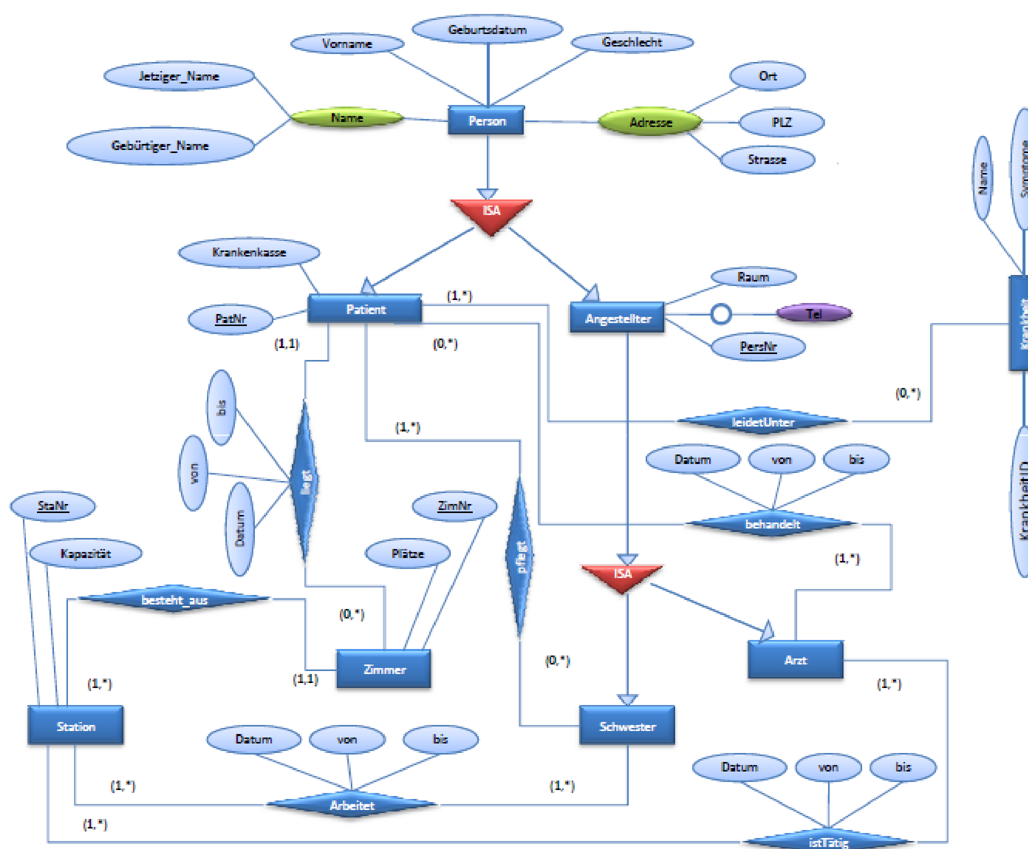


Abbildung 17.1: Entity-Relationship-Modell *Krankenhaus*

es auch vorkommen, dass ein Patient von mehreren Ärzten gleichzeitig behandelt wird oder noch gar nicht. Neben den Ärzten arbeiten auch Krankenschwestern auf den Stationen. Die Schwestern arbeiten nicht immer nur auf einer Station, sondern können die Stationen auch wechseln. Auf einer Station ist mindestens eine Schwester tätig, die sich um die Patienten kümmert. Patienten können auch gleichzeitig von mehreren Schwestern versorgt werden.

- **Generalisierung**: Hierbei werden Eigenschaften von ähnlichen Entity-Typen einem gemeinsamen Obertyp zugeordnet. Somit stellt der Untertyp eine Spezialisierung des Obertyps dar.

  In unserem Entity-Relationship-Modell *Krankenhaus* sind z.B. *Angestellter* und *Patient* jeweils Spezialisierungen von *Person* und stehen daher zu diesem

Entity-Typ in einer ISA Beziehung.

- **Optionales Attribut** (Kreis in der Attribut-Linie): Attributwert muss nicht für jede Entität vorhanden sein.
- **Strukturiertes Attribut** (Attribute eines Attributes): Ein Attribut wird durch weitere Attribute beschrieben.

17.2 Datenbanktabellen

Nun sind wir endlich bereit, die Datenbank zu erstellen. Dafür nutzen wir den SQL Developer. Das ist ein freies grafisches Tool für die Datenbankentwicklung. Damit können SQL-Anweisungen und SQL-Skripte bearbeitet und ausgeführt werden. SQL Developer steigert die Produktivität und vereinfacht Datenbankentwicklungsaufgaben.

Nachdem eine Verbindung zur Datenbank hergestellt wurde, hat man mit Hilfe des SQL Developers zwei Möglichkeiten zum Anlegen der Tabellen. Entweder man verwendet die grafische Oberfläche, die die nötigen SQL-Anweisungen generiert, oder man gibt die CREATE TABLE Anweisung im SQL-Worksheet selbst ein. Beim Erstellen von Tabellen muss man den Namen der Tabelle, Spaltenname(n) und den Datentyp für jede Spalte eingeben.

Um uns an dem Programm anmelden und um Benutzer verwalten zu können, hat der Oracle-Administrator zwei Benutzer angelegt (krankenhaus_normal und krankenhaus_lying). Damit haben wir die anderen Benutzer hinzugefügt.

Zu beachten ist es auch, dass In dem Schema Krankenhaus_lying alle Constraints deaktiviert sind. Damit werden die Integritätsbedingungen nur innerhalb das Schema krankenhaus_normal aktiv.

Tipp: Der Benutzer mit dem Schema, in dem die Datenbank angelegt wird, ist immer (Oracle) Administrator und damit kann er Berechtigungen an andere Benutzer erteilen.

- Die Tabelle Patient beinhaltet die persönlichen Daten der Patienten. GDatum

steht für Geburtsdatum, KKasse für Krankenkasse, JName für Jetziger Name und GName für Gebürtiger Name.

Patient

PatNr	Vorname	Gdatum	Geschlecht	KKasse	JName	Gname	Ort	PLZ	Straße
einseins	Patrik	einssechzig	männlich	AOK	Derarme	Derarme	Hagen	viervierdrei	Lewackerstr
zweizwei	Martin	einsechzig	männlich	DKV	Derkranke	Derkranke	Wattenscheid	vierviereins	Bochumerweg
dreidrei	Karin	einsechzig	weiblich	DEWS	Derarme	Biermann	Essen	viervierzwei	Murhardstr

- Die Tabelle Schwester enthält die persönlichen Daten der Krankenschwestern im Krankenhaus.

Schwester

PersNr	Vorname	Gdatum	Geschlecht	Raum	TelNr	JName	GName	Ort	PLZ	Straße
dreizwei	Miriam	siebeneins	weiblich	vier	einsnull	Dick	Dick	Essen	zweizwei	Balzstr
viersechs	Katherina	siebendrei	weiblich	vier	einsein	Bremer	Bremer	Bochum	zweieins	Laerstr
siebenacht	Nadja	siebenzwei	weiblich	drei	einszwei	Fritz	Fritz	Dortmund	zweidrei	Hühnerstr

- Die Tabelle Arzt beinhaltet die persönlichen Daten, der im Krankenhaus tätigen Ärzte.

Arzt

PersNr	Vorname	Gdatum	Geschlecht	Raum	TelNr	JName	GName	Ort	PLZ	Straße
einszwei	Mathias	einssieben	männlich	zehn	einszwei	Bäumer	Bäumer	Bochum	viervieracht	Markstr
dreizwei	Marko	viersieben	männlich	einseins	einsdrei	Fröhlich	Fröhlich	Dortmund	siebensieben	Essenerstr
vierfunf	Sabine	zweiieben	männlich	eins	einsdrei	Freier	Freier	Münster	einszweidrei	Kortumstr

- Die Tabelle Krankheit enthält eine Sammlung der Krankheiten, die in dem Krankhaus behandelt werden können.

Krankheit

KrankheitsId	Name	Symptome
eins	Schweinegrippe	Fieber, Knie und Gelekschmerzen
zwei	Blähungen	Der Bauch ist vorgewölbt und fühlt sich gespannt an
drei	Blasenentzündung	Schmerzhaften Gängen auf die Toilette verbunden

- Die Tabelle Station gibt für jede Station eine Nummer und Kapazität an.

Station

StaNr	Kapazität
eins	zweitausend
zwei	dreitausend
drei	zweitausend
vier	zweitausend

- Die Tabelle Zimmer weist jedem Zimmer die entsprechende Anzahl der Plätze zu.

Zimmer

ZimNr	Plätze
eins	drei
zwei	funf
drei	zehn
vier	acht
funf	sechs
sechs	zwei

- Die Zwischentabelle LeidetUnter enthält zwei Fremdschlüssel, nämlich PatNr aus der Tabelle Patient und KrankheitsId aus der Tabelle Krankheit

LeidetUnter

PatNr	KrankheitsId
einseinseins	eins
zweizweizwei	zwei
dreidreidrei	zwei

- Die Zwischentabelle BestehtAus enthält zwei Fremdschlüssel, nämlich StaNr aus der Tabelle Station und ZimNr aus der Tabelle Zimmer.

BestehtAus

StaNr	ZimNr
eins	eins
eins	zwei
zwei	drei
zwei	vier
drei	funf
vier	sechs

- Die Zwischentabelle Pflegt enthält zwei Fremdschlüssel, nämlich PersNr aus der Tabelle Patient und PatNr aus der Tabelle Schwester.

Pflegt

PersNr	PatNr
dreizwei	einseins
viersechs	zweizwei
siebenacht	dreidrei

- Die Zwischentabelle gibt an, welcher Patient zu welchem Zeitpunkt auf welchem Zimmer gelegen hat.

Liegt

Datum	Von	Bis	PatNr	ZimNr
nullneun	zehn	einsvier	einseins	eins
zweizweinullsechs	acht	zweinull	zweizwei	zwei
siebendreizehn	sieben	sechs	dreidrei	drei

- Die Zwischentabelle Behandelt enthält zwei Fremdschlüssel, nämlich PersNr aus der Tabelle Arzt und PatNr aus der Tabelle Patient.

Behandelt

Datum	Von	Bis	PersNr	PatNr
zweizweinullacht	einsvier	einsfunf	einszwei	einseins
zweizweinullacht	neun	neundrei	dreizwei	zweizwei
zweizweinullacht	zehn	elf	vierfunf	dreidrei

- Die Zwischentabelle Arbeitet enthält zwei Fremdschlüssel, nämlich PersNr aus der Tabelle Arzt und PatNr aus der Tabelle Patient.

Arbeitet

Datum	**Von**	**Bis**	**PersNr**	**StaNr**
zweizweinullacht	sechs	einsvier	dreizwei	zwei
zweizweinullacht	einsvier	zweizwei	viersechs	zwei
zweizweinullacht	zweizwei	sechs	siebenacht	zwei

- In dieser Tabelle ist angegeben, welcher Arzt wo und wann arbeitet.

IstTätig

Datum	**Von**	**Bis**	**PersNr**	**StaNr**
einseinsnullsieben	sechs	einsvier	einszwei	eins
einseinsnullsieben	einsvier	zweizwei	dreizwei	eins
einseinsnullsieben	zweizwei	sechs	vierfunf	eins

17.3 Testfälle

Diese Tests dienen dem Test der Gesamtfunktionalität der implementierten Funktionen mit der neuen Datenbank und wurden im Prototypen selbst durchgeführt. Und in allen hier aufgeführten und getesteten Fällen, stimmen gewünschtes und tatsächliches Ergebnis überein.

Damit ein Test erfolgreich durchgeführt werden kann, müssen die folgenden Nebenbedingungen erfüllt sein:

1. Jedes Relationensymbol muss eine Relation beschreiben, die auch tatsächlich in der aktuellen DB-Instanz mit der entsprechenden Stelligkeit existiert.
2. Alle vorkommenden Variablen müssen unter ANF auch gebunden worden sein.

Eine dieser BNF ungenügende Anfrage wäre z.B. : **zimmer(1,3);**

Deswegen haben wir bei der Implementierung der neuen Datenbank nur den Datentyp *Varchar2* genommen.

- **Testfall 1:** Optimierter offener Anfragen(Fünfstellige Relation, zwei freie Variablen)

Zensor: Verweigerungszensor

pot_sec:

Liegt				
Datum	**Von**	**Bis**	**PatNr**	**ZimNr**
nullneun	zehn	einsvier	einseinseins	eins

log: -

query: liegt(nullneun,X,einsvier,einseinseins,Y);

==============================

Gewünschte Ausgabe: liegt(nullneun,zehn,einsvier,einseinseins,drei)

Ausgabe: liegt(nullneun,zehn,einsvier,einseinseins,drei)

- **Testfall 2:** Optimierter offener Anfragen(Fünfstellige Relation, zwei freie Variablen, Existenzquantor)

  Zensor: Verweigerungszensor

  pot_sec: exists X(exists Y liegt(nullneun,zehn,X,einseinseins,Y));

  log: -

  query: liegt(nullneun,zehn,einsvier,einseinseins,eins);

  Gewünschte Ausgabe: MUM

  Ausgabe: MUM

- **Testfall 3:** Optimierter offener Anfragen(nothing_pos)

  Zensor: Verweigerungszensor

  Vorbedingungen: openOpt=true, boundVar=false, opt_possible=true

  log: -

  pot_sec: -

  query: (zimmer(eins,drei) or zimmer(drei,zehn);

  ==============================

  Gewünschte Ausgabe: (zimmer(eins,drei) or zimmer(drei,zehn)

Gewünschte Nachbedingungen: openOpt=true, boundVar=false, opt_possible=false

Ausgabe: (zimmer(eins,drei) or zimmer(drei,zehn))

Nachbedingungen: openOpt=true, boundVar=false, opt_possible=false

- **Testfall 4:**

  Zensor: Lügenzensor

  PeterLog:

PeterLog
Formel
(not zimmer(eins,drei))

  BinhLog:

BinhLog
Formel
(not zimmer(eins,drei))

  BinhPotSec:

BinhPotSec
Formel
zimmer(eins,drei)

  LorenzoLog: -

  Admin-Update-Anfrage: zimmer(eins,drei);

  ==============================

  Gewünschte Ausgabe:

  – PeterLog: zimmer(eins,drei)

  – BinhLog: (not zimmer(eins,drei))

  – LorenzoLog: -

  Ausgabe: wie gewünscht

- **Testfall 5.a:** Sicht-Änderung (Wert NICHT enthalten und neues Log impliziert EIN Geheimnis)

  Zensor: Lügenzensor

  pot_sec:

PotSec
Formel
(not zimmer(eins,zehn))

Log:

Log
Formel
Station(zwei,zweitausend)

DB:

Station	
StaNr	**Kapazität**
zwei	zweitausend

Constraints:

Constraints
Formel
Station(zwei,zweitausend)

Admin-Update-Anfrage: zimmer(eins,zehn);

================================

Gewünschtes Ergebnis:

1. **log**:

Log
Formel
station(zwei,zweitausend)
zimmer(eins,zehn)

2. **Ausgabe**: Wert bereits in der Datenbank enthalten.

Tatsächliches Ergebnis: wie gewünscht

- **Testfall 5.b:** Sicht-Änderung (Wert NICHT enthalten und neues Log impliziert EIN Geheimnis)

Zensor: Lügenzensor

pot_sec:

PotSec
Formel
zimmer(eins,zehn)

Log:

Log
Formel
station(zwei,zweitausend)

DB:

Station	
StaNr	**Kapazität**
zwei	zweitausend

Zimmer	
ZimNr	**plätze**
eins	zehn

Constraints:

Constraints
Formel
station(zwei,zweitausend)

Admin-Update-Anfrage: (not zimmer(eins,zehn));

==============================

Gewünschtes Ergebnis:

1. **log**:

Log
Formel
station(zwei,zweitausend)
(not zimmer(eins,zehn))

2. **Ausgabe**: Wert bereits in der Datenbank enthalten.

Tatsächliches Ergebnis: wie gewünscht

- **Testfall 6.a:** Sicht-Änderung (Konsistenz nicht verletzt)

Zensor: Lügenzensor

pot_sec:

PotSec
Formel
zimmer(eins,zehn)

Log:

Log
Formel
station(zwei,zweitausend)
zimmer(zwei,vierzehn)

DB:

Station	
StaNr	**Kapazität**
zwei	zweitausend

Zimmer	
ZimNr	**plätze**
zwei	vierzehn

Constraints:

Constraints
Formel
station(zwei,zweitausend)

Admin-Update-Anfrage: (not zimmer(zwei,vierzehn));

==============================

Gewünschtes Ergebnis:

1. **log**:

Log
Formel
station(zwei,zweitausend)
(not zimmer(zwei,vierzehn))

2. **DB**:

Station	
StaNr	Kapazität
zwei	zweitausend

Zimmer	
ZimNr	plätze

3. **Ausgabe**: Änderungs-Operation durchgeführt.

Tatsächliches Ergebnis:

1. **log**: wie gewünscht
2. **DB**: wie gewünscht
3. **Ausgabe**:Wert wurde eingefuegt.

- **Testfall 6.b:** Sicht-Änderung (Konsistenz nicht verletzt)

Zensor: Lügenzensor

pot_sec:

PotSec
Formel
zimmer(eins,zehn)

Log:

Log
Formel
station(zwei,zweitausend)
(not zimmer(zwei,vierzehn))

DB:

Station	
StaNr	Kapazität
zwei	zweitausend

Constraints:

Constraints
Formel
station(zwei,zweitausend)

Admin-Update-Anfrage: zimmer(zwei,vierzehn);

==============================

Gewünschtes Ergebnis:

1. **log**:

Log
Formel
station(zwei,zweitausend)
zimmer(zwei,vierzehn)

2. **DB**:

Station	
StaNr	**Kapazität**
zwei	zweitausend

Zimmer	
ZimNr	**plätze**
zwei	vierzehn

3. **Ausgabe**: Änderungs-Operation durchgeführt.

Tatsächliches Ergebnis:

1. **log**: wie gewünscht
2. **DB**: wie gewünscht
3. **Ausgabe**:Wert wurde eingefügt.

Kapitel 18

ToDos

- Wenn man einen neuen Benutzer hinzufügen möchte, tauchen ab und zu Fehler (Exceptions) auf. Die Ursache(n) sollte gefunden und behoben werden. Es könnte an der Privilegienvergabe auf Oracle-Ebene liegen, muss aber nicht.
- Die GUI zur Benutzer Editierung sollte verbessert werden. Um die Funktionalität zu gewährleisten wurde kurzfristig eine veränderte GUI erstellt. Swing ermöglicht auf die schnelle keine Erstellung einer ansprechenden GUI, daher werden die Buttons auf FenstergröÃŸe skaliert. Es muss ein neuer (komplizierterer) LayoutManager Verwendung finden.
- Automatische Anmeldung im zugehörigen Schema gemäß Zensor. Falls der Lügenzensor ausgewählt wurde, findet die Anmeldung im Schema jcqe_lying statt. Ansonsten, also bei Wahl des Verweigerungs- oder des kombinierten Zensors, wird der Nutzer im Schema jcqe_normal angemeldet. Beachte den Umgang mit den Administratoren jcqe_normal und jcqe_lying.
- Auslesen der $Oracle - MetaData$ in SQLInteraction effizienter gestalten. Jede Kommunikation mit der Datenbank kostet viel Zeit und daher auch jeder Zugriff auf Metadaten (Spaltennamen usw.) einer Tabelle. Deswegen der Vorschlag beim Programmstart schon einige $MetaData$ viel verwendeter Tabellen auszulesen und in geeigneten Datenstrukturen (wie z.B. HashTables) zwischenzuspeichern. Zusätzlich muss eine effiziente Methode gefunden werden, die $MetaData$ der Tabellen auszulesen. Zur Zeit geschieht das über eine Anfra-

ge $SELECT * ...$ über welche dann die MetaData ausgelesen werden können. Es ist bis jetzt keine bessere Lösung bekannt, sich aber alle Tupel ausgeben zu lassen, nur um an die Spaltennamen zu gelangen, ist sehr ineffizient.

- Änderungsoperationen auf mehrspaltigen Relationen wurde nicht implementiert. Das Auslesen der Benutzereingabe wurde bis jetzt nur für einspaltige Relationen implementiert. Damit ein ViewUpdate auf allen atomaren Formeln vollzogen werden kann, muss das Parsen des Wertes bezüglich des Syntaxbaumes in Konstanten und Relation angepasst werden. (vgl. Abschnitt 12.2.11)
- Konzept der Constraints überarbeiten. Zur Zeit werden die Constraints in einer Tabelle als Formeln (bzw. Strings) gehalten und jeder Benutzer hat ein Flag in der Benutzertabelle, ob die Constraints schon im initialen Log stehen. Falls nein, werden diese einmalig hinzugefügt.
- Überlegen, ob eine Umstellung auf „PreparedStatements" für SQL-Anfragen in SQLInteraction in Hinblick auf Performance und Sicherheit sinnvoll ist.
- Ein weiteres Problem stellt zur Zeit die Definition der $< user >$ -Geheimnisse und des -Vorwissens dar. Sinnvoll wäre, einen Benutzer erst nach Erstellung von Geheimnissen und Vorwissen durch den Admin freischalten zu können. Wenn ein Admin einen Benutzer hinzufügt, dann sollte die Option implementiert werden, dass der Oracle-Benutzer erst deaktiviert ist und noch freigeschaltet werden muss. Direkt nach der Erstellung eines neuen Benutzers per GUI, müsste dem Admin die Frage per Java-Interaktionsfenster „JOptionPane" gestellt werden, ob er die Geheimnisse jetzt oder zu einem späteren Zeitpunkt definieren möchte. Wenn er die Geheimnisse direkt anlegen möchte, dann müsste ihm der „UserEditFrame" eingeblendet werden, in dem er die Geheimnisse und das Vorwissen anlegen kann. Beim „Speichern" der Daten müsste der Benutzer auf Oracle-Ebene aktiviert werden. Falls der Admin sich dazu entscheidet die Geheimnisse zu einem späteren Zeitpunkt anzulegen, dann muss der Status des Benutzers in der Benutzertabelle festgehalten werden und eine Möglichkeit gegeben sein, dass der Admin den Benutzer nachträglich aktivieren kann. Ob in diesem Fall Geheimnisse angelegt worden sind, obliegt dem Wissen des Admins.
- Die Besitzer der Schemata (also zwei der Administratoren) erhalten die Rechte um Oracle-Benutzer anlegen zu können. Allen anderen Administratoren wird

laut der GUI des Prototypen erlaubt, Benutzer hinzuzufügen. Dieses löst aber mangels Privilegien eine Exception aus. Es muss über ein geeignetes „Separation of Duty"-Konzept nachgedacht werden.

- In der Klasse SQLInteraction müssen die Zugriffe auf die Datenbanktabellen mittels *Lock*() (*Row* – *Level* – *Locking*) abgesichert werden. Die Methode *Lock*() ist schon implementiert, sie muss allerdings noch in allen anderen Methoden eingebunden werden, damit bei einem Mehrbenutzersystem keine Inkonsistenz entsteht, weil zwei Prozesse des Prototypen gleichzeitig auf eine Tabelle „schreibend oder lesend" zugreifen. (Transaktionsverwaltung)

- Da wir es zeitlich nur geschafft haben den theoretischen Ansatz von Gogolin für Transaktionen (siehe hierzu [Gog08] Kapitel 7) zu implementieren, ist dies noch ein offener Punkt. Dessen Korrektheitsbeweis findet man in „Requirements and protocols for inference-proof interactions in information systems " [JBW08] von Biskup, Gogolin, Seiler, Weibert zur Kompatibilitätsprüfung der Algorithmen von Seiler zur Sichtänderung und Gogolin zur Sicht-Erneuerung.

- Durch die von uns implementierte Trennung von Administrator Update und Benutzer Update mittels Rolle und GUI, sind bereits beide Algorithmen voneinander getrennt und parallel einsetzbar. Meldet man sich als einfacher Benutzer an, kann man nur Benutzer Updates durchführen und als Administrator nur die Administrator Updates. Die Aktualisierung der Benutzerlogs funktioniert bei beiden Implementierungen gleich.

- Auf Grund eines Missverständnisses glaubten wir Literale wären atomare Formeln mit nur einem Attribut, d.h. es können nur Tabellen mit einer Spalte korrekt bearbeitet werden. Diesen Fehler gilt es noch zu beheben.

- In dem Zustandsdiagramm 14.3 ist der Zustand *open Opt_pos* aufgeführt. In diesem Zustand können aufgrund des Problems mit den Existenzquantoren, das den Schutz von Geheimnissen gefährdet (siehe Abschnitt 14.2) keine Anfragen mehr gebundene Variablen enthalten. Es wäre aber möglich gebundene Variablen in geschlossenen Anfragen zuzulassen, wenn man:

  1. Diese Anfragen dynamisch bearbeiten würde.

2. Das neu hinzugekommene Benutzerwissen, durch vorangegangene offene Anfragen, im Benutzerlog speichern würde.

Der Fehler tritt nämlich in dem Fall einer offenen Anfrage gefolgt von einer Anfrage mit mind. einer gebundenen Variablen nur auf, wenn bei der Anfrage mit gebundener Variable das Benutzerlog nicht berücksichtigt wird oder dieses nicht vollständig ist.
Wir haben die Methode cqeOpen(Formula) dahingehend modifiziert, dass geschlossene Anfragen mit Existenzquantoren, die nach einer offenen Anfrage erfolgen, ein Dialogfeld öffnen. Dieses bietet die Möglichkeiten, die Anfrage dynamisch bearbeiten zu lassen oder die Anfrage umzuformulieren.
Leider war es uns nicht möglich, das durch optimierte offene Anfragen gewonnene Benutzerwissen korrekt zu speichern. Es wird im Rahmen der Methode cqeOpenOpt(Formula) der Klasse Application lediglich ein String erzeugt, in dem die Formel gespeichert ist, die das neu hinzugekommene Benutzerwissen korrekt ausdrückt. Leider kommt es aber zu einer Fehlermeldung, bei dem Versuch diesen in der Oracle Datenbank zu speichern, so dass nach offen optimierten Anfragen kein neues Benutzerwissen erzeugt wird.
Deswegen haben wir die Methode cqeOpen(Formula) der Klasse Application dahingehend abgeändert, dass geschlossene Anfragen mit Existenzquantor per Fehlermeldung unterbunden werden. Offene Anfragen mit Existenzquantor müssen zur Zeit nicht berücksichtigt werden, da nur optimierte offene Anfragen behandelt werden, welche per Definition keine Existenzquantoren enthalten (BNF 14.3.2). Offene Anfragen, die nicht optimiert behandelt werden können, müssen nicht berücksichtigt werden, da diese nur mit einem anderen Zensortypen vorkommen können.
Das Unterbinden jeglicher geschlossener Anfrage mit Existenzquantor kann rückgängig gemacht werden, sobald das neu gewonnene Benutzerwissen einer optimierten offenen Anfrage korrekt gespeichert wird, weswegen dies eine wichtige zukünftige Aufgabe darstellt.

- Wie man dem Zustandsdiagramm 14.3 entnehmen kann, gilt das ganze Diagramm nur für den Verweigerungszensor, aus dem einfachen Grund, dass ein statischer Ansatz bei offenen sowie geschlossenen Anfragen bisher nur bei diesem Zensortypen möglich ist. Im aktuellen Prototypen (PG536) gibt es allerdings keinen Verweigerungszensor für offene Anfragen, die nicht optimiert

behandelt werden können.
Die Implementierung eines dynamischen Ansatzes mit Verweigerungszensor für offene Anfragen würde folgendes erreichen:

1. Anfragen mit freien Variablen, die die zusätzlichen Bedingungen einer optimierten Anfrageauswertung, die in Kapitel 14.2 beschrieben sind, nicht erfüllen, könnten bearbeitet werden.
2. Analog zu dem Unterkapitel 14.6.1 könnte man für die Zustände open Opt_imp, boundVar_pos1 und boundVar_pos2 offene Anfragen zulassen, diese müssten dann aber von dem noch zu implementierenden dynamischen Ansatz der offenen Anfrageauswertung bearbeitet werden. Das Benutzerwissen wird bei geschlossenen Anfragen in jedem Fall schon richtig gespeichert.

Wie man sieht, wird durch die Implementierung eines dynamischen Ansatzes mit Verweigerungszensor für offene Anfragen die Menge der auswertbaren Anfragen weiter steigen, sowie eine Symmetrie im Zustandsdiagramm hergestellt, was der Verständlichkeit zu Gute kommt.

- Im aktuellen Prototyp (PG536) werden im statischen Fall, sowohl bei offenen als auch bei geschlossenen Anfragen, Klassifikationsinstanzen wie folgt verwaltet:

1. Für jede Datentabelle wird bereits bei der Schemaerzeugung eine Klassifikationsinstanztabelle erzeugt.
2. Während der Laufzeit werden für jede Anfrage die entsprechenden Klassifikationsinstanztabellen bzgl. der Geheimnismenge des aktuellen Benutzers gefüllt und nach Ausgabe der Antwort wieder geleert.

Nun ist dieses Vorgehen nicht sonderlich effizient. Es wäre effizienter, wenn die Klassifikationsinstanztabellen eines Benutzers einmal gefüllt werden würden und nur noch angepasst würden, wenn sich die Geheimnismenge des Benutzers ändert.
Allerdings bräuchte man dann pro Datentabelle pro Benutzer eine entsprechende Klassifikationsinstanztabelle. Um das wiederum verwalten zu können, müssten die Klassifikationsinstanztabellen eines Benutzers bei der Erzeugung eines neuen Benutzers auf Oracleebene angelegt werden. Man kann aber nicht

einfach für jede Tabelle des entsprechenden Oracleschemas eine Klassifikationsinstanztabelle erzeugen, da es auch reine Verwaltungstabellen gibt, wie z.B. die Benutzertabelle, für die man keine Klassifikationsinstanz benötigt. Leider kann die Entscheidung, für welche Tabellen man eine Klassifikationsinstanz benötigt, noch nicht automatisiert getroffen werden, da dies bisher nur aus dem Namen der Tabelle ersichtlich ist.
Dieses Problem zu lösen ist ein erster Schritt, die Klassifikationsinstanzen persistent zu speichern und somit die statische Anfrageauswertung effizienter zu gestalten.

- Aus folgendem Grund konnten wir den Testfall 5 mit dem Prototypen nicht testen. Die Methode *minimizePotSec*() enthält folgenden Fehler:
  - bluemer Potsec = { exists X reha (X, b, c, d) }
  - füge psi= 'exists Y reha (Y, b, c, d)' ein:
  - Ergebnis des Redundanztests der PG495 mit Hilfe von Proover9:
    - bluemer Potsec = { }

    Das führt dazu, dass **"füge psi2=reha (X, b, c, d) ein"** auch nicht richtig funktioniert, da wir psi2 wegen der Approximation durch **exists FVARX reha(FVARX, b, c, d)** ersetzen.

- Falls die Mengen der potentiellen Geheimnisse freie Variablen enthalten, wird im dynamischem Modus eine Approximation angewendet, so dass die freien Variablen durch gebundene Variablen ersetzt werden. In diesem Fall sollte der User benachrichtigt werden, dass seine Anfragen approximiert behandelt werden.

- Die in den Anforderungen für die erste Iteration festgelegte Klasse von Formeln für Anfragen, kann durch folgende Backus-Naur-Form beschrieben werden:

  N = {ANF ,EXP ,R,VAR ,KONST ,RELSYMB ,PARAM ,JUNC ,KBUCHST ,GBUCHST}

T $= \{\wedge, \vee, \neg, \exists, (,), ; , a, ..., z, A, ..., Z\}$

ANF ::= (∃ VAR)* EXP;

EXP ::= R | (EXP JUNC EXP) | (¬EXP)

JUNC ::= ∨ | ∧

R ::= RELSYMB(PARAM(, PARAM)*)

PARAM ::= KONST | VAR

VAR ::= (GBUCHST)+

KONST ::= (KBUCHST)+

RELSYMB ::= (KBUCHST)+

KBUCHST ::= a | . . . | z

GBUCHST ::= A | . . . | Z

- Damit ein Test erfolgreich durchgeführt werden kann, müssen die folgenden Nebenbedingungen erfüllt sein:
  1. Jedes Relationensymbol muss eine Relation beschreiben, die auch tatsächlich in der aktuellen DB-Instanz mit der entsprechenden Stelligkeit existiert.
  2. Alle vorkommenden Variablen müssen unter ANF auch gebunden worden sein.
- Eine dieser BNF ungenügende Anfrage wäre z.B. : **zimmer(1,3);**
  Deswegen haben wir bei der Implementierung der neuen Datenbank nur den Datentyp *Varchar2* genommen.

Literaturverzeichnis

[Ack68] Wilhelm Ackermann. *Solvable Cases of the Decision Problem.* Studies in logic and the foundations of mathematics. North-Holland Publ. Co., 1968.

[BB07] Joachim Biskup and Piero Bonatti. Controlled Query Evaluation with Open Queries for a decidable relational Submodel. *Annals of Mathematics and Artificial Intelligence 50*, pages 39–77, 2007.

[Bec01] Bernhard Beckert. Automatisches Beweisen. `i12www.ira.uka.de/~beckert`, SS 2001. Vorlesungsskriptum.

[Ben09] Lorenzo Benet. Sichterneuerung, 2009. Seminarausarbeitung.

[Bisa] Joachim Biskup. Complete Systems - Dynamic Enforcement of Confidentiality under Query Sequences. Forlesungsfolien.

[Bisb] Joachim Biskup. Schutz von Information durch Kontrollierte Anfrageauswertung. Paper.

[Bis95] Joachim Biskup. *Grundlagen von Informationssystemen.* Vieweg-Lehrbuch Informatik. Vieweg, 1995.

[BKI08a] Christoph Beierle and Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme.* Studium. Vieweg + Teubner, 4 edition, 2008.

[BKI08b] Christoph Beierle and Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme.* Vieweg+Teubner, Wiesbaden, 2008.

[Bui09] Binh Bui. Automatisches Beweisen, 2009. Seminarausarbeitung.

[Gog08] Christian Gogolin. *Kontrollierte Anfrageauswertung und Sichterneuerung.* Diplomarbeit, TU Dortmund, Juli 2008.

[Gro05] Martin Grohe. Mathematische und logische Grundlagen der Informatik. `http://www2.informatik.hu-berlin.de/logik/lehre/WS04-05/`, WS 2004/05. Vorlesungsskriptum.

[HGF98] R. Cattell Hamilton G. and M. Fisher. *JDBC Datenbankzugriff mit Java*. Addison-Wesley, Bonn, 1998.

[HW07] H.Faeskorn-Woyke. *Datenbanksysteme*. Pearson Studium, 2007.

[Jab09] Adalat Jabrayilov. Optimierung, der statische Zensor, 2009. Seminarausarbeitung.

[JBL09] Jan-Hendrik Lochner Joachim Biskup, Sven Hartmann and Sebastian Link. Inference-proof access control for relational databases. page 12, 2009. Paper.

[JBW08] Jens Seiler Joachim Biskup, Christian Gogolin and Torben Weibert. Requirements and Protocols for Inference-proof Interactions in Information Systems. page 20, 2008. Paper.

[Jun02] Achim Jung. Logik. In Pomberger Rechenberg, editor, *Informatik-Handbuch*, chapter 1, pages 34–72. Hanser Verlag, 3 edition, 2002.

[KÖ8] Thomas Käufl. Automatisches Beweisen. `http://algo2.iti.uni-karlsruhe.de/kaeufl/`, SS 2008. Vorlesungsskriptum.

[Loc09] Jan-Hendrik Lochner. Optimization of the Controlled Evaluation of Closed Relational Queries. 2009.

[May09] Nils Maybaum. Oracle-Virtual Private Database, 2009. Seminarausarbeitung.

[M.S06] M.Skulschus. *Oracle 10g Das Programmierhandbuch*. Galileo Computing, 2006.

[Oraa] Oracle. Database 2 Day Developer's Guide 11g release 1. `http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28843/toc.htm`. Oracle Online Documentation.

[Orab] Oracle. Database Advanced Security Administrator's Guide 11g release 1. `http://download.oracle.com/docs/cd/B28359_01/network.111/b28530/toc.htm`. Oracle Online Documentation.

[Orac] Oracle. Database Security Guide 11g release 1. `http://download.oracle.com/docs/cd/B28359_01/network.111/b28531/toc.htm`. Oracle Online Documentation.

[PG408] Projektgruppe PG495. Kontrollierte Anfrageauswertung für relationale Datenbanken, WS 20007 - SS 2008. Endbericht.

[Sch09] Thomas Schwentick. Logik für Informatiker. `http://ls1-www-cs.uni-dortmund.de/cms/vorlesung-logik-für-informatiker-ws-08-09/`, WS 2008/09. Vorlesungsfolien.

[Sch09] Torsten Schlotmann. Sichtänderung, 2009. Seminarausarbeitung.

[Sei08] Jens Seiler. *Kontrollierte Anfrageauswertung und Sichtänderung*. Diplomarbeit, TU Dortmund, Juni 2008.

[Sin08] Carsten Sinz. Automatisches Beweisen. `http://www-sr.informatik.uni-tuebingen.de/~post/AB_SS06/skript.pdf`, WS 2007/08. Vorlesungsskriptum.

[Son08] Sebastian Sonntag. *Optimierung der kontrollierten Anfrageauswertung für relationale Datenbanken durch geeignete Anwendung statischer Inferenzkontrollmechanismen*. Diplomarbeit, TU Dortmund, Mai 2008.

[SS05] Vicorica Sofronie-Stokkermans. Automatisches Beweisen. `http://www.mpi-inf.mpg.de/~sofronie/teaching/autreas1-trier.html`, WS 2004/05. Vorlesungsskriptum.

[S.U05] S.Urman. *Oracle-Database-10g-PL-SQL-Programmierung*. Hanser, 2005.

[Xia08] Yuhong Xia. *Glaubwürdige Definition und optimierte Verwendung von Wertebereichsaufzählungen für die kontrollierte Auswertung offener Anfragen*. Diplomarbeit, TU Dortmund, Juli 2008.

[Zha08] Yu Zhang. *Algorithmen für kontrollierte Anfrageauswertung offener Anfragen: Weiterentwicklung, Implementierung und Vergleich*. Diplomarbeit, TU Dortmund, Juni 2008.