Kontrollierte Anfrageauswertung für relationale Datenbanken

PG 495

18. September 2007

Endbericht der Projektgruppe
Teilnehmer: Elena Bernchteine, Dennis Blümer, Kalin Boyanov, Peiman Dabidian, Irina Felko, Pedram Rasekhi, Sebastian Sonntag, Yuhong Xia, Yu Zhang

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I
Einführung

1 Beschreibung

Kann man tatsächlich mit effizienten informatorischen Mitteln inhaltliche Information vertraulich halten, oder muss man sich darauf beschränken, die Weitergabe von rohen Daten zu unterbinden? Die Projektgruppe versuchte, darauf eine positive Antwort zu geben: Es wurde ein System entworfen und implementiert, das den Schutz vertraulicher Information bei der Verwendung einer relationalen Datenbank auf effiziente Weise kontrolliert.
Die benutzten Methoden der kontrollierten Anfrageauswertung waren:

- Ein Benutzerbild mit dem (vermuteten) derzeitigen Wissen des Benutzers unterhalten.
- Jede neue Anfrage dahingehend überprüfen, ob eine korrekte Antwort die Vertraulichkeitspolitik verletzen würde?
- Gegebenenfalls eine geeignet veränderte Antwort in Form einer „Lüge“ oder einer „Verweigerung“ liefern, wobei eine Methode auch beide Formen kombinieren kann. (In der ersten Iteration wurde nur die „Verweigerungsmethode“ verwendet, in den zweiten und dritten Iteration wurden die weiteren Methoden verwendet.)

Die Abbildung 1 auf Seite 13 stellt eine „theoretische Architektur“ der kontrollierten Anfrageauswertung dar.

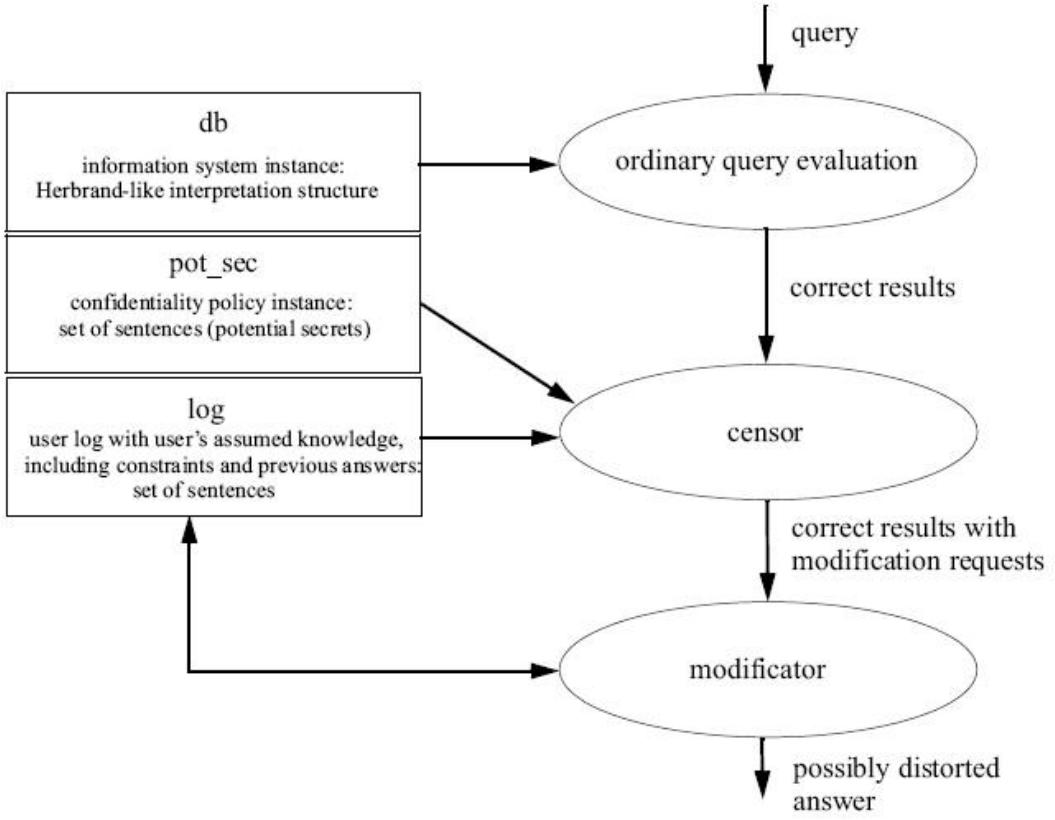

Abbildung 1: Eine „theoretische Architektur“ der kontrollierten Anfrageauswertung

2 Organisation und Vorgehensweise der Projektgruppe

1. Seminarfahrt (23.10.06 - 24.10.06)
   Die Seminarphase fand im Haus Ortlohn in Iserlohn statt. Alle PG-Teilnehmer hatten sich zuvor ein Thema ausgesucht und einen Vortrag ausgearbeitet, der in der Seminarphase den anderen Teilnehmern vorgetragen wurde. Es wurden dabei folgende Vortragsthemen von den PG-Teilnehmern vorgetragen:

   - Kontrollierte Anfrageauswertung – Sebastian Sonntag und Yuhong Xia
   - Relationale Datenbanken und Entscheidbarkeitstheorie – Yu Zhang und Elena Bernchteine
   - Theorembeweiser – Irina Felko und Dennis Blümer
   - Oracel DBMS – Pedram Rasekhi und Peiman Dabidian

2. Vorbereitungsphase (25.10.06 - 10.11.06)
   Nach der Seminarfahrt begann die Projektgruppe direkt mit der nächsten Phase, es wurden Vorbereitungen für die weitere Organisation getroffen. Dazu wurden grundlegende Dinge wie die Sitzungsleitung, die Protokollführung und die weitere Vorgehensweise besprochen. Die Projektgruppe teilte sich in zwei Gruppen ein, um sich mit den zwei Schwerpunkten unserer weiteren Arbeit zu beschäftigen, nämlich der SQL-Übersetzung von Anfragen und der Prüfung von Implikationen.

3. Projektentwicklungsphase (13.11.06 - 16.2.07)
   Um sich in Zukunft besser zu organisieren und strukturierter vorgehen zu können, hat sich die Projektgruppe in den Vortrag Projektmanagement eingelesen. Dabei wurden folgende Entscheidungen getroffen:

   - Das Projekt ist in mehreren Iterationen gemäß des Spiralmodells aufzubauen.
     Die Projektgruppe hat daraufhin den folgenden Ablauf für die erste Iteration festgelegt. Begonnen hat die erste Iteration am 13.11.06 mit der Analysephase bis zum 26.11.06. Die Hauptaufgabe war die kennengelernte Theorie aus der Seminarfahrt mit unserem Projektziel zu verbinden, z.B. die Zensorauswertung, die Syntax der zu verwendenden Formeln. Die zweite Phase setzte sich aus drei Wochen Entwurfsphase, vom 27.11.06 bis zum 17.12.06 zusammen. In dieser Phase wurden die folgenden UML-Diagramme für das zu erstellende Programm entworfen: Klassendiagramm, Anwendungsfalldiagramme, Aktivitätsdiagramme und Sequenzdiagramme. Am Ende der Phase wurden schließlich noch die entstandenen Methoden beschrieben. Ab dem 18.12.06 fing die Phase der Codierung an, welche insgesamt zwei Wochen gedauert hat. Dazu haben sich Gruppen von 1 bis 2 Personen zusammengesetzt, die anhand der entworfenen Diagramme aus der vorherigen Phase die Javaklassen codierten. Nach Abschluss dieser Phase folgte ab dem 15.01.2007 bis zum 21.01.2007 die Testphase. Es mussten hier alle erstellten Programmteile separat getestet und auf Fehler

überprüft werden. Dazu wurden die einzelnen Methoden mit JUnit getestet. Ab dem 22.01.07 bis zum 16.02.07 erstellte die Projektgruppe den Zwischenbericht für das Projekt. Dabei sollten alle Seminarausarbeitungen, das Anforderungsdokument, das Entwurfsdokument, die Testergebnisse, der Zeitplan für das nächste Semester und das Implementierungsdokument in dieses Dokument aufgenommen werden.

- Die zu verwendende Software besteht dabei aus:
  - Together J — für die Entwurfphase, zur Erstellung von Diagrammen für die Softwareentwicklung.
  - Java — für die Java-Programmierung.
  - SVN — für die gemeinsame Arbeit am Projekt.
  - Prover9 — als Theorembeweiser kommt Prover9 zum Einsatz.

Im Wintersemester 06/07 wurde somit die kontrollierte Anfrageauswertung durch geschlossene Anfragen implementiert, dabei war die Software nur für einen Benutzer ausgelegt, der seine Sicherheitspolitik und sein Vorwissen selber definiert und Anfragen an das System richten kann. Das Ergebnis der Anfrage wird mit einem ablehnendem Zensor zensiert, was zur Folge hat, dass geprüft werden muss, ob die entsprechende Antwort zusammen mit dem Benutzerwissen ein potentielles Geheimnis impliziert. Dieses soll als Grundlage und Untersuchungsobjekt für die spätere Weiterentwicklung in der zweiten Iteration dienen.
Die Entwicklungsarbeit der Projektgruppe wurde im Sommersemester 07 in zwei weitere Iterationen (Iteration 2 und 3) unterteilt.
Bei der zweiten Iteration wurden folgende Teile realisiert:

- Serialisierung der Verbindungsinformationen zur Datenbank
- Mehrbenutzerbetrieb
- Konsistenzprüfung beim Benutzerwissen
- Negation in Formeln
- Lügen und kombinierte Methode für geschlossene Anfragen
- Fehlerbehandlung
- Optimierung und Wahrung von Redundanzfreiheit

Bei der dritten und letzten Iteration wurden zwei Teile realisiert:

- Behandlung offener Anfragen
- Behandlung der DB-Implikation

Es wurden alle vorgenommenen Ziele im Projekt erreicht. Es gibt allerdings einige Kritikpunkte, die auf der Seite 306 erwähnt worden sind.

Teil II
Seminarphase

3 Kontrollierte Anfrageauswertung

3.1 Einführung in die Thematik

Die kontrollierte Anfrageauswertung setzt Sicherheitspolitiken in Informationssystemen ein, um diese vertraulich nutzbar zu machen. Zu jeder Anfrage wird die gegebene Antwort zunächst von einem Zensor beurteilt und falls nötig entsprechend modifiziert, um die Sicherheit in dem System zu wahren.
Hierzu werden zwei verschiedene Modelle für Sicherheitspolitiken untersucht, zum einen das Heimlichkeits-Modell und zum anderen das Modell der potentiellen Geheimnisse. Beim Heimlichkeits-Modell sollen Paare von komplementären Aussagen geschützt werden, so dass dem Benutzer sowohl die positive als auch die negative Alternative als möglich erscheint. Beim Modell der potentiellen Geheimnissen hingegen werden einzelne Aussagen geschützt, so dass dem Benutzer die zugehörige komplementäre Aussage als möglich erscheint. Ferner kann man unterscheiden, ob dem Benutzer die Sicherheitspolitik bekannt oder unbekannt ist. Auf offene Anfragen des Benutzers an das Informationssystem werden nur bejahende, verneinende oder abgelehnte Antworten vom System zurückgeliefert. Das System kann also auf solche Anfragen die Wahrheit sagen, lügen oder aber die Anfrage zurückweisen. Wie wir später sehen werden, können diese beiden Methoden des Lügens und Ablehnens ebenfalls kombiniert werden, so dass sich die Schwachstellen der einzelnen Methoden reduzieren lassen.

In der entstehenden Informationsgesellschaft sind viele Informationssysteme in offene Umgebungen eingebettet, die von verschiedenen Benutzern geteilt werden. Daher besteht ein großes Erfordernis eine Sicherheitspolitik zu spezifizieren, die festlegt, welcher Benutzer welche Information erhalten darf und diese Sicherheitspolitik effizient einzusetzen.
Ein wichtiges Ziel der Sicherheitspolitik eines Informationssystems ist es, die Vertraulichkeit bestimmter sensibler Informationen zu gewährleisten. Hierzu gibt es zwei Ansätze, den statischen und den dynamischen Ansatz. Bei dem statischen Ansatz setzt ein Administrator im Voraus adäquate Zugriffsrechte auf die Informationen, basierend auf einer vollständigen Analyse der Schutzanforderungen. Der Vorteil des Ansatzes ist, dass er relativ umkompliziert und schnell umgesetzt werden kann. Der Nachteil liegt darin, dass die entsprechenden Zugriffsrechte bereits zur Entwurfszeit festgelegt und dann nicht mehr geändert werden können. Letzteres gilt auch für den dynamischen Ansatz, für diesen wird die Bürde der Anforderungsanalyse auf das System umgelegt. D.h. das System entscheidet bei jeder Anfrage, wie darauf angemessen geantwortet wird, unter Berücksichtigung der vorherigen Anfragen und der zugehörigen Antworten des Systems. Das System kann entweder auf Anfragen lügen oder die Anfragen ablehnen. Dies geschieht allerdings nur, wenn der Benutzer ansonsten in der Lage wäre, ein Geheimnis abzuleiten. Dieser Ansatz ist jedoch sehr vorteilhaft, da eine vorherige Analyse der Schutzanforderungen entfällt und flexibel auf die gegenwärtige Situation reagiert werden kann.
Um etwas tiefer in die Thematik einsteigen zu können, müssen zunächst einige Begriffe erläutert bzw. definiert werden.
Was genau versteht man z.B. unter einer Anfrage? Eine Anfrage ist eine Aussage vom

Benutzer, die dem Informationssystem übermittelt wird. Anschließend analysiert dieses die Anfrage und antwortet darauf, möglicherweise wird die ursprüngliche Antwort dazu modifiziert. Das Informationssystem hat die Option auf eine Anfrage zu lügen oder diese abzulehnen. Beim Lügen ändert das Informationssystem die Antwort so ab, dass Ihre Negation zurückgegeben wird. Beim Ablehnen hingegen wird lediglich die Entscheidung preisgegeben, dass überhaupt nicht auf diese Anfrage geantwortet wird.
Sicherheitspolitiken können unter verschiedenen Modellen betrachten werden. Unter dem Modell der Heimlichkeiten und unter dem Modell der potentiellen Geheimnissen. Eine Heimlichkeit ist ein Paar, bestehend aus einer Aussage und der zugehörigen Negation. Unter einer Sicherheitspolitik, die das Modell der Heimlichkeiten benutzt, darf der Benutzer nicht die wahre Alternative ableiten können, daher werden beide vor ihm versteckt.
Unter dem Modell der potentiellen Geheimnisse wird nur eine Alternative der Aussage beschützt. Unter einer Sicherheitspolitik, die das Modell der potentiellen Geheimnisse benutzt, darf der Benutzer nicht die Alternative ableiten können, die das potentielle Geheimnis darstellt, die andere hingegen darf abgeleitet werden.
Ein Beispiel, bei dem zwischen Heimlichkeiten und potentiellen Geheimnissen unterschieden werden kann, ist z.B. der Satz: „Herr Müller ist verheiratet.“ als Sicherheitspolitik interpretiert. Unter der Sicherheitspolitik, die auf Heimlichkeiten basiert, ist es weder erlaubt mitzuteilen, dass Herr Müller verheiratet ist, noch, dass er es nicht ist. Unter der Sicherheitspolitik basierend auf potentiellen Geheimnissen hingegen, ist es nicht erlaubt mitzuteilen, dass Herr Müller verheiratet ist, aber es darf mitgeteilt werden, dass Herr Müller nicht verheiratet ist, sofern das in der aktuellen Instanz zutrifft. Betrachtet man hier den Spezialfall des Modells des Lügens mit dem potentiellen Geheimnis „Herr Müller ist verheiratet“, so wird stets mit „Herr Müller ist nicht verheiratet“ geantwortet.
Bei bekannten Sicherheitspolitiken kann davon ausgegangen werden, dass der Benutzer unter beiden Modellen (Heimlichkeiten / potentielle Geheimnisse) erkennen kann, was das Informationssystem verbergen möchte. M.a.W. dem Benutzer ist bekannt, dass eine spezielle Aussage geschützt wird, er kennt aber natürlich nicht die wahre Alternative davon.

3.2 Anfrageauswertung

In diesem Abschnitt betrachten wir die einfache und kontrollierte Anfrageauswertung. Anschließend wird die Sicherheitspolitik erörtert, die auf bekannten und unbekannten Heimlichkeiten basiert. Den Abschluss des Abschnitts bildet die Diskussion der Sicherheitspolitik basierend auf bekannten potentiellen Geheimnissen.

3.2.1 Einfache Anfrageauswertung

Ein Informationssystem umfasst das Datenbank-Schema DS, mit einer endlichen Menge von Aussageformen (d.h. der Menge der erlaubten Instanzen) und die Struk-

tur der Datenbank-Instanz *db*, die die Wahrheitswerte für alle Aussagen festlegt. Beispielsweise sei eine Datenbankinstanz *db* und eine Anfrage Φ gegeben, wobei Φ eine Aussage ist, die entweder wahr oder falsch in *db* ist. Stellt der Benutzer eine Anfrage an das Informationssystem, so liefert $eval(\Phi)$ die entsprechende Antwort unter der Instanz *db*. Man definiert $eval(\Phi)$ durch

$$eval(\Phi) : DS \rightarrow \{true, false\} \text{ with } eval(\Phi)(db) := db \texttt{ model\_of } \Phi.$$

Eine äquivalente Definition zu $eval(\Phi)$ ist $eval^*(\Phi)$ mit

$$eval^*(\Phi) : DS \rightarrow \{\Phi, \neg\Phi\} \text{ with}$$
$$eval^*(\Phi)(db) := \texttt{if } db \texttt{ model\_of } \Phi \texttt{ then } \Phi \texttt{ else } \neg\Phi.$$

Es wird entweder die Anfrage oder deren Negation zurückgeliefert. [BB01]

3.2.2 Kontrollierte Anfrageauswertung

Die kontrollierte Anfrageauswertung besteht aus zwei Schritten. Zunächst wird die korrekte Antwort von einem Zensor beurteilt und anschließend wird, abhängig von der Beurteilung, die Antwort modifiziert. Der Zensor hat die Aufgabe zu überprüfen, ob durch eine einfache Anfrageauswertung Geheimnisse gefolgert werden können. Um den Zensor zu unterstützen, führt das System einen Benutzerlog, genannt *log*, der aus einer Menge von Sätzen, dem Benutzerwissen, besteht. Hier werden alle vom System gegebenen Antworten auf Benutzeranfragen protokolliert.
Letztendlich definiert sich die kontrollierte Anfrageauswertung *control_eval* durch

$$control\_eval(Q, log_0)(db, policy) = \langle (ans_1, log_1), (ans_2, log_2), ..., (ans_i, log_i), ... \rangle ,$$

dabei stellen die Parameter Q eine Anfragesequenz der Form $Q = \langle \Phi_1, \Phi_2, ..., \Phi_i, ... \rangle$ und log_0 den initialen Benutzerlog dar. Die Eingabe für diese Funktion stellt das Paar $(db, policy)$ dar, mit *db* einer Datenbankinstanz und *policy* einer angemessenen Sicherheitspolitik.
Wir betrachten dazu das folgende Beispiel mit der Anfragesequenz $Q = \langle p \vee q, p, q \rangle$, dem initialen Benutzerlog $log_0 = \emptyset$ (keine Vorwissen), der Instanz $db = \{p, \neg q\}$ und der Sicherheitspolitik $policy = \emptyset$ (keine Schutz). Die kontrollierte Anfrageauswertung ergibt dann

$$control\_eval(\langle p \vee q, p, q \rangle, \emptyset)(\{p, \neg q\}, \emptyset)$$
$$= \langle (p \vee q, \{p \vee q\}), (p, \{p \vee q, p\}), (\neg q, \{p \vee q, p, \neg q\}) \rangle .$$

Jede einzelne kontrollierte Anfrageauswertung gibt somit die Wahrheit aus. Der Benutzerlog wird nach jeder Benutzeranfrage aktualisiert.

3.2.3 Sicherheitspolitik basierend auf (un)bekannten Heimlichkeiten

Wie schon zuvor definiert, besteht eine Heimlichkeit aus einem Paar von komplementären Sätzen. Eine Sicherheitspolitik ist eine Menge *secr* aus Heimlichkeiten mit den folgenden Bedingungen:

- Für jede Heimlichkeit aus *secr* soll der Benutzer nicht die wahre Alternative ableiten können.
- Für jede Folge von Anfragen müssen die Antworten konsistent mit jeder Annahme der wahren Werte sein.

Um diese Bedingungen unter unbekannten Heimlichkeiten für die Ablehnungsmethode einzuhalten, wird der Geheimnis-Zensor eingesetzt. Dieser erteilt den Befehl zur Ablehnung einer Anfrage, falls sonst zusammen mit dem Benutzerlog eine wahre Alternative abgeleitet werden könnte.
Für die Ablehnungsmethode unter bekannten Heimlichkeiten wird der stärkere Heimlichkeiten-Zensor verwendet.

$$censor^R_{sec}(\Phi, log, db, secr) := (\text{exists}\, S \in secr, \text{exists}\, \Psi \in S)$$
$$[log \cup \{eval^*(\Phi)(db)\} \models \Psi \text{ or } log \cup \{\neg eval^*(\Phi)(db)\} \models \Psi]$$

Man erkennt aus dieser Definition, dass der Zensor immer dann eine Anfrage ablehnen muss, wenn $eval^*(\Phi)$ mit dem bereits bekannten Wissen *log* einen geschützten Satz Ψ enthüllen würde. Werden aber keine weiteren Antworten mehr blockiert, so könnte der Benutzer den wahren Wert von einer abgelehnten Antwort Φ zusammen mit *log* ableiten. Aus diesem Grund blockiert der Zensor auch dann die Anfrage, wenn das Komplement der korrekten Antwort Ψ implizieren würde. Daher können Instanzen, in denen Φ wahr ist, nicht von denen unterschieden werden, in denen Φ falsch ist.
Die zugehörige Anfrageauswertung $control\_eval^R_{sec}(Q, log_0)(db, secr)$ wird definiert durch

$$ans_i := \texttt{if}\ \ censor^R_{sec}(\Phi_i, log_{i-1}, db, secr)\ \ \texttt{then mum else}\ \ eval^*(\Phi_i)(db)$$
$$log_i := \texttt{if}\ \ ans_i = \texttt{mum then}\ \ log_{i-1}\ \ \texttt{else}\ \ log_{i-1} \cup \{ans_i\}$$

Für die Lügenmethode unter unbekannten Heimlichkeiten ist ein Geheimnis-Disjuktions-Zensor zuständig, die drei oben genannten Bedingungen einzuhalten. Er erteilt den Befehl zum Lügen, falls die wahre Antwort die Disjunktion der wahren Alternativen implizieren würde.
Die Lügenmethode unter bekannten Heimlichkeiten ist nicht anwendbar.
Zur Beurteilung der Schutzmethoden unter unbekannten Heimlichkteiten muss berücksichtigt werden, dass der Geheimnis-Zensor schwächer als der Geheimnis-Disjunktions-Zensor ist, und bei Ablehnung müssen nicht alle Disjunktionen der Geheimnisse geschützt werden. Ablehnung informiert den Benutzer über die vorgenommenen Schutzaktivitäten, Lügen jedoch nicht. Es kann also gesagt werden, dass Lügen den Benutzer fehlleitet und Ablehnen ein Bit Information preisgibt.

3.2.4 Sicherheitspolitik basierend auf bekannten potentiellen Geheimnissen

Ein potentielles Geheimnis ist ein Satz Ψ. Eine Sicherheitspolitik der Menge *pot_sec* aus potentiellen Geheimnissen besitzt die folgenden Bedingungen:

- Für jedes potentielle Geheimnis $\Psi \in$ *pot_sec* soll der Benutzer nicht ableiten können, dass Ψ wahr ist.
- Antworten des Informationssystem müssen konsistent sein, so dass der Benutzer glaubt, jedes beliebige potentielle Geheimnis sei falsch.
- Diese Bedingungen sollen auch dann gelten, wenn der Benutzer bereits die Menge *pot_sec* der potentiellen Geheimnisse kennt.

Die erste Bedingung ist erfüllt, wenn das potentielle Geheimnis auch ein tatsächliches Geheimnis ist, d.h. für die aktuelle Instanz *db* gilt *db* `model_of` Ψ. Möglicherweise ist ein Satz aus *pot_sec* nicht wahr in der aktuellen Instanz. In diesem Fall darf der Benutzer wissen, dass die Negation des Satzes in db wahr ist (2. Bedingung).

Betrachten wir nun die formale Definition der Sicherheitsanforderungen.
Definition 1: Sei $control\_eval(Q, log_0)$ eine kontrollierte Anfrageauswertung für die Anfragesequenz $Q = \langle \Phi_1, \Phi_2, ..., \Phi_i, ... \rangle$ und für den initialen Benutzerlog log_0 und sei *pot_sec* die Menge der potentiellen Geheimnisse.

1. $control\_eval(Q, log_0)$ ist sicher in Bezug auf *pot_sec*, wenn für alle endlichen Präfixe Q' von Q die folgenden Bedingungen gelten: Für alle potentiellen Geheimnissen $\Psi \in$ *pot_sec*, so dass $log_0 \not\models \Psi$ und für alle adäquaten Argumente (db_1, pot_sec) existiert ein adäquates Argument (db_2, pot_sec), so dass:

   (a) [gleiche Antworten]:
   $control\_eval(Q', log_0)(db_1, pot\_sec) = control\_eval(Q', log_0)(db_2, pot\_sec)$
   (b) [das potentielle Geheimnis ist falsch]:
   $eval^*(\Psi)(db_2) = \neg\Psi$

2. Allgemeiner kann man sagen, dass $control\_eval(Q, log_0)$ sicher ist, wenn es sicher in Bezug auf alle adäquaten Mengen von potentiellen Geheimnissen ist.

Für jedes potentielle Geheimnis aus *pot_sec* und irgendeine endliche Teilfolge von Antworten existiert eine Instanz, die diese Antworten erzeugt und dem potentiellen Geheimnis den wahren Wert „falsch“ zuordnet. Es gilt daher für jede Instanz, es existiert eine andere Instanz (evtl. die gleiche), so dass beide Instanzen nicht unterscheidbar sind (1. Eigenschaft) und das potentielle Geheimnis in der zweiten Instanz falsch ist (2. Eigenschaft).

Für die Lügenmethode unter potentiellen Geheimnissen wird der Lügen-Zensor $censor^L_{ps}$ verwendet

$$censor^L_{ps}(\Phi, log, db, pot\_sec) := log \cup \{eval^*(\Phi)(db)\} \models pot\_sec\_disj,$$
where $pot\_sec\_disj := \bigvee_{\Psi \in pot\_sec} \Psi$

Der Lügen-Zensor lehnt eine Anfrage ab, wenn die Antwort sonst die Disjunktion der potentiellen Geheimnisse implizieren würde. Die entsprechende Anfrageauswertung hat die Form
$control\_eval^L_{ps}(Q, log_0)(db, pot\_sec) = \langle (ans_1, log_1), (ans_2, log_2), ..., (ans_i, log_i), ... \rangle$
und wird definiert durch

$$ans_i := \texttt{if } censor^L_{ps}(\Phi_i, log_{i-1}, db, pot\_sec) \texttt{ then } \neg eval^*(\Phi_i)(db) \texttt{ else } eval^*(\Phi_i)(db)$$
$$log_i := log_{i-1} \cup \{ans_i\}$$

Der Ablehnungs-Zensor unter potentiellen Geheimnissen besitzt in Bezug auf den entsprechenden Heimlichkeiten-Zensor keine nennenswerten Unterschiede.

3.3 Lügen unter bekannten potentiellen Geheimnissen

3.3.1 Ein Fall zum Lügen

Angenommen ein Benutzer fordert Informationen von einem gemeinsam genutzten Informationssystem an. Es kann nun sein, dass das Informationssystem umfangreicher ist als eigentlich benötigt. Die Hauptaufgabe des Systems besteht darin die Geheimnisse vor dem Benutzer zu verstecken, so dass er nicht merkt, dass etwas vor ihm versteckt wird. Fordert der Benutzer Informationen an, die er nicht einsehen darf, so soll das System angemessen reagieren ohne zu verraten, dass es sich der Gefahr bewusst ist. Damit diese Ziele verwirklicht werden können, darf das Informationssystem manchmal lügen. Die folgenden Anforderungen sind dazu dienlich:

- Der Benutzer kann auch dann seine Anwendung ausführen, wenn das Informationssystem lügt.
- Die Anwendung ist somit ausreichend unabhängig von den Anfragen auf die gelogen wird.
- Das System lügt nur, wenn es unabdingbar ist, d.h. Geheimnisse ansonsten nicht geschützt sind.
- Der Benutzer kann nicht feststellen, ob das Informationssystem lügt.
- Der Benutzer kann keine Geheimnisse von den Antworten seiner Anfragen ableiten.
- Auch wenn der Benutzer die Menge der potentiellen Geheimnisse kennt, sollen die obigen Punkte eingehalten werden.

Ein Benutzer, der die potentiellen Geheimnisse kennt, kann nicht alle Antworten des Systems glauben. Er weiß z.B. genau, dass das System keine tatsächlichen Geheimnisse

preisgibt. Er kann austesten, welche Antworten unglaubwürdig sind, d.h. ob es eine eine Instanz gibt, die Lügen zurückgibt. Z.B. könnte der Benutzer eine Anfrage auf beide Alternativen an das Informationssystem stellen, bekommt er auf beide Anfragen die gleiche Antwort, so ist diese Antwort unglaubwürdig.

3.3.2 Eine Gliederung zum Lügen

Die Lügenmethode versucht wie folgt Informationen zu verbergen:

1. Initialisierung des Benutzerlogs durch $log_0 = constraints$, wobei *constraints* die Menge der Sätze enthält, die der Benutzer für wahr in der aktuellen Instanz hält.
2. Nach der Anfrageauswertung wird der Benutzerlog aktualisiert durch $log_n = log_0 \cup \{ans_1, \ldots, ans_n\}$ ersetzt.
3. Der Benutzerlog darf nie ein potentielles Geheimnis implizieren.
4. Um nützliche Antworten zu liefern, soll das System korrekt antworten, so fern dies möglich ist.
5. Das System soll lügen, wenn der Benutzer sonst die Disjunktion aller potentiellen Geheimnisse ableiten könnte.
6. Für den initiale Benutzerlog log_0 muss $log_0 \not\models pot\_sec\_disj$ gelten, um die obige Bedingung nicht zu verletzen.

3.3.3 Unglaubwürdige Antworten

Angenommen es liegt die folgende Situation vor:

- Binäres Prädikat: *pred*
- Konstantensymbole: p(erson), v(erheiratet), u(nverheiratet)
- Semantische Bedingungen: $pred(p, v) \vee pred(p, u)$ und $\neg pred(p, v) \vee \neg pred(p, u)$

Es existieren somit zwei mögliche Instanzen:

- $db_1 := \{pred(p, v) \vee \neg pred(p, u)\}$
- $db_2 := \{\neg pred(p, v) \vee pred(p, u)\}$

Die Sicherheitspolitik sei gegeben durch $pot\_sec := \{pred(p, v)\}$, dann ist $pred(p, v)$ ein tatsächliches Geheimnis für db_1 aber nicht für db_2. Angenommen die Benutzeranfragen seien $pred(p, v)$, $pred(p, u)$. So wären die gelieferten Antworten dazu:

- Für db_1:
  $\neg pred(p, v)$, was eine Lüge ist, um das tatsächliche Geheimnis zu schützen.
  $pred(p, u)$, was auch eine Lüge ist, um die Konsistenz mit der semantischen Bedingung zu wahren.
- Für db_2:
  $\neg pred(p, v)$, was der Wahrheit entspricht, da die Information harmlos ist.
  $pred(p, u)$, was auch wahr ist.

Obwohl die Instanzen unterschiedlich sind, erhält man die gleichen Antworten.
Wie beurteilt ein fortgeschrittener Benutzer die erhaltenen Antworten? Er kann nur zu dem Ergebnis kommen, dass die Antworten nicht glaubwürdig sind.
1. Wenn ihm das potentielle Geheimnis $pot\_sec := \{pred(p, v)\}$ bekannt ist, wird er schlussfolgern können, dass das System immer $\neg pred(p, v)$ für die erste Anfrage zurückliefert. Daher ist die erste Antwort also unglaubwürdig.
2. Ist die erste Antwort unglaubwürdig, dann lässt sich folgendes folgern: Ist die erste Antwort gelogen, dann ist es auch die zweite Antwort, oder die erste Antwort ist korrekt, dann ist es auch die zweite Antwort. D.h. die zweite Antwort ist entweder gelogen oder wahr, also insgesamt unglaubwürdig.

3.4 Ablehnen unter bekannten potentiellen Geheimnissen

3.4.1 Ein Fall zum Ablehnen

Die Methode des Lügens und die der Ablehnung sind in vielen Beziehungen ähnlich, der Hauptunterschied besteht jedoch darin, dass durch Ablehnung einer Anfrage der Benutzer nicht fehlgeleitet werden kann. Die Anforderungen an ein Informationssystem, das auf die Ablehnungsmethode zurückgreift, unterscheiden sich nur in einem Punkt von denen der Lügenmethode.

- Der Benutzer kann auch dann seine Anwendung ausführen, wenn das Informationssystem die Anfrage ablehnt.
- Die Anwendung ist somit ausreichend unabhängig von den Anfragen, die abgelehnt werden.
- Das System lehnt nur ab, wenn es unabdingbar ist, d.h. ansonsten Geheimnisse nicht geschützt sind.
- **Der Benutzer kann feststellen, ob das Informationssystem ablehnt.**
- Der Benutzer kann keine Geheimnisse von den Antworten seiner Anfragen ableiten.
- Auch wenn der Benutzer die Menge der potentiellen Geheimnisse kennt, sollen die obigen Punkte eingehalten werden.

Das bedeutet für den Benutzer, dass er die Antworten, die er vom Informationssystem erhält, bedenkenlos glauben kann.

3.4.2 Eine Gliederung zum Ablehnen

Der Ablehnungs-Zensor veranlasst die Modifikation der Antwort der einfachen Anfrageauswertung, falls der Benutzer sonst ein potentielles Geheimnis ableiten könnte.

$$censor^R_{ps}(db, pot\_sec, log, \Phi) := (\text{ exist } \Psi)\{[\Psi \in pot\_sec \text{ and } [log \cup \{eval^*(\Phi)(db)\} \models \Psi \text{ or } log \cup \{\neg eval^*(\Phi)(db)\} \models \Psi]].$$

Die Auswertung des Zensors ist dabei unabhängig von der aktuellen Instanz db:

$$censor^R_{ps}(db, pot\_sec, log, \Phi) := (\text{ exist } \Psi)\{[\Psi \in pot\_sec \text{ and } [log \cup \{\Phi\} \models \Psi \text{ or } log \cup \{\neg\Phi\} \models \Psi]].$$

3.4.3 Errungenschaften

Theorem 1 *Die kontrollierte Anfrageauswertung* $control\_eval^R_{ps}(Q, log_0)$ *mit dem Ablehnungs-Zensor ist sicher gemäß Definition 1.*

Beweis: Q' sei ein endlicher Präfix von Q, n die Anzahl der Anfragen in Q', $\Psi \in pot\_sec$ mit $log_0 \not\models \Psi$ und (db_1, pot_sec) ein adäquates Argument mit db_1 `model_of` log_0.
(x) Weil nur Sätze, die wahr in db_1 sind, ausgegeben werden, gilt auch db_1 `model_of` log_n.
(y) Aus der Definition des Ablehnungs-Zensors lässt sich ableiten, dass $log_n \not\models \Psi$.
(z) Nach Definition von $\models$ gibt es eine Instanz db_2 mit db_2 `model_of` log_n **und** db_2 `model_of` $\neg\Psi$.

Wir zeigen nun, dass (db_2, pot_sec) die gewünschten Eigenschaften besitzt.
Die geforderte Eigenschaft b) ergibt sich direkt aus Punkt (z), somit gilt $eval^*(\Psi)(db_2) = \neg\Psi$.
Die geforderte Eigenschaft a) erhalten wir mit:
1.) Der Ablehnungs-Zensor entscheidet unabhängig von der Instanz.
2.) Die einfache Anfrageauswertung gibt die gleichen Werte für db_1 und db_2 zurück, siehe Punkt (x) und (z).
Für einen formaleren Beweis zeige man noch, dass $control\_eval^R_{pot_sec}(Q', log_0)(db_1, pot\_sec) = control\_eval^R_{ps}(Q', log_0)(db_2, pot\_sec)$ per Induktion über i der Anfrage Φ_i. [BB04b]

3.5 Funktionale Äquivalenz

In diesem Kapitel analysieren wir die funktionale Äquivalenz zwischen Lügen und Ablehnen. Es gibt zwei Voraussetzungen für die funktionale Äquivalenz:

1. Potentielle Geheimnisse sind unter Disjunktion abgeschlossen.
   $pot\_sec = \{\sigma_1, ..., \sigma_n\}$, und $\sum = \sigma_1 \vee ... \vee \sigma_n$ ist das schwächste potentielle Geheimnis.

2. Der initale Benutzerlog impliziert nicht das schwächste potentielle Geheimnis. Wir definieren hier $log_0 = constraints$ und $log_0 \not\models \sum$.

3.5.1 Potentielle Geheimnisse sind unter Disjunktion abgeschlossen

Gegeben sei $Q = \langle \sigma_1 \vee ... \vee \sigma_n \rangle$ unter Lügen und Ablehnen.

Für $C = \{R, L\}$ sei

$\langle ((ans_1^C(db), log_1^C(db)), (ans_2^C(db), log_2^C(db)), ...(ans_i^C(db), log_i^C(db)), ... \rangle$
$= control\_eval^C((Q, log_0))((db, pot\_sec))$.

Hier ist $log_0^C(db) = log_0$ und, $log^C(db) = \langle log_0^C(db), log_1^C(db), ...log_i^C(db)... \rangle$.

Zuerst benötigen wir noch einige Sätze:

Satz 1 *(Non-distroted answers) Sei $C \in \{R, L\}$. Wenn für alle $\Phi_i^* \in \{\Phi_i, \neg\Phi_i\}$, $log_{i-1}^C db \cup \{\Phi_i^*\} \not\models \sum$, dann gilt $ans_i^C(db) = eval^*(\Phi_i)(db)$.*

Satz 2 *(Possibly distorted answers) Wenn für ein $\Phi_i^* \in \{\Phi_i, \neg\Phi_i\}$, $log_{i-1}^C db \cup \{\Phi_i^*\} \models \sum$, dann gilt $ans_i^L(db) = \neg\Phi_i^*$.*

Satz 3 *(Refusals) Wenn für ein $\Phi_i^* \in \{\Phi_i, \neg\Phi_i\}, log_{i-1}^R(db) \cup \{\Phi_i^*\} \models \sum$, dann gilt $ans^{R_i}(db) = mum$.*

Lemma 1 *Für alle $i \geq 0$ und alle Sätze T,*

$$log_i^L(db) \cup T \models \sum \text{ iff } log_i^R(db) \cup T \models \sum.$$

Theorem 2 *Sei pot_sec eine Menge von potentiellen Geheimnissen, die abgeschlossen unter Disjunktion mit schwächstem Element $\sum$ ist, und log_0 sei initialiseirt, so dass $log_0 \not\models \sum$. Dann gilt, für alle Anfragesequenzen Q und für alle Instanzen db_1 und db_2 :*

$$log^R(db_1) = log^R(db_2) \text{ iff } log^R(db_1) = log^R(db_2).$$

Beweis:
Wir beweisen die Äquivalenz der Aussagen über:

$$log^R(db_1) \neq log^R(db_2) \text{ iff } log^L(db_1) \neq log^L(db_2).$$

($\Rightarrow$)Angenommen $log^R(db_1) \neq log^R(db_2)$ und sei i der kleinste Index, so dass gilt

$$log_{i-1}^R(db_1) = log_{i-1}^R(db_2), \tag{1}$$

$$log_i^R(db_1) \neq log_i^R(db_2). \tag{2}$$

Nach Def. von *log* folgt sofort

$$ans_i^R(db_1) \neq ans_i^R(db_2). \tag{3}$$

Jetzt gibt es zwei Möglichkeiten:

$$\text{Für ein } \Phi_i^* \in \{\Phi_i, \neg\Phi_i\}, log_{i-1}^R(db_1) \cup \{\Phi_i^*\} \models \sum \tag{4}$$

$$\text{Für alle } \Phi_i^* \in \{\Phi_i, \neg\Phi_i\}, log_{i-1}^R(db_1) \cup \{\Phi_i^*\} \not\models \sum \tag{5}$$

Trifft (1) zu, dann gilt auch (4):

$$\Phi_i^* \in \{\Phi_i, \neg\Phi_i\}, log_{i-1}^R(db_2) \cup \{\Phi_i^*\} \models \sum$$

Nach Satz 3 gilt $ans_i^R(db_1) = ans_i^R(db_2) = mum$, das steht im Widerspruch zu (3).

Somit muss nach (3) der Punkt (5) gelten:

$$\Phi_i^* \in \{\Phi_i, \neg\Phi_i\}, log_{i-1}^R(db_2) \cup \{\Phi_i^*\} \not\models \sum$$

Nach Lemma folgt, dass für alle

$$\Phi_i^* \in \{\Phi_i, \neg\Phi_i\}, log_{i-1}^L(db_1) \cup \{\Phi_i^*\} \models \sum \text{ (j=1,2) .}$$

Jetzt kann mit Satz 1 abgeleitet werden, dass

$$ans_i^L(db_1) = eval^*(\Phi_i)(db_j) \text{ (j=1,2) ,}$$

da $eval^*(\Phi_i)(db_1) \neq eval^*(\Phi_i)(db_2)$ gilt. Sonst gilt $ans_i^R(db_1) = ans_i^R(db_2)$, das steht im Widerspruch zu (3), so folgt

$$ans_i^L(db_1) = eval^*(\Phi_i)(db_1) \neq eval^*(\Phi_i)(db_2) = ans_i^R(db_2)$$

und $log^L(db_1) \neq log^L(db_2)$. □

3.6 Kombination von Lügen und Ablehnen

Bis jetzt haben wir schon Lügen und Ablehnen kennengelernt und darauf hingewiessen, dass die beiden Methoden jeweils Nachteile haben. Ablehnen muss die Antwort beschützen, auch wenn die falsche Antwort schädlich ist. Lügen muss die Disjunktion beschützen. Um die Nachteile zu vermeiden, kombinieren wir die beiden Methoden.

Wenn eine Anfrage Φ gegeben ist, gibt es drei Reaktionsmöglichkeiten. Die erste Reaktionsmöglichkeit ist Ablehnen. Wenn das aktuelle Log und die richtige oder falsche Antwort ein potentielles Geheimnis implizieren würden, wird abgelehnt.

Die zweite Reaktionsmöglichkeit ist Lügen. Gelogen wird, wenn mit dem aktuellen Log und der richtigen Antwort ein potentielles Geheimnis impliziert wird, aber mit dem aktuellen Log und der falschen Antwort kein potentielles Geheimnis impliziert wird.
Die dritte Reaktionsmöglichkeit ist das Ausgeben der richtigen Antwort. Diese wird ausgegeben, wenn das aktuelle Log und die richtige Antwort kein potentielles Geheimnis implizieren.

3.6.1 Kombination von Lügen und Ablehnen unter bekannten potentiellen Geheimnissen

Gegeben sei pot_sec, Q=$\langle \Phi_1, \Phi_2, ..., \Phi_i, ... \rangle$ und log_0, und wir wollen die folgende Invariante für den Benutzerlog log_i aufrechterhalten:

$$\text{Für alle } \Psi \in pot\_sec, log_i \not\models \Psi. \tag{6}$$

Für die Initialisierung (i=0) benötigen wir diese Eigenschaft als Vorbedingung. Die neue kombinierte Methode, d.h. die kontrollierte Anfrageauswertung kombiniert durch Ablehnen und Lügen, $control\_eval^C_{PS}(Q, log_0)$ ist definiert als

$$control\_eval^C_{PS}(Q, log_0)(db, pot\_sec) = \langle (ans_1, log_1), (ans_2, log_2), ...(ans_i, log_i)... \rangle .$$

mit i-ter Antwort ans_i und i-tes log_i gegeben durch

ans_i = **if** (exists ψ_1) [$\psi_1 \in pot\_sec$ and $log_{i-1} \cup \{eval^*(\Phi_i)(db)\} \models \psi_1$]
then
if
(exists ψ_2)
[$\psi_2 \in pot\_sec$ and $log_{i-1} \cup \{\neg eval^*(\Phi_i)(db)\} \models \psi_2$]
then mum
else $\neg eval^*(\Phi_i)(db)$
else $eval^*(\Phi_i)(db)$

und

log_i:= **if** ans_i = mum **then** log_{i-1} **else** $log_{i-1} \cup \{ans_i\}$.

Wir betrachten einige Beispiele, damit man die Theorie besser verstehen kann. In allen Beispielen ist $pot_sec = \{$s1,s2,s3$\}$, $log_0 = \{$p $\Rightarrow$ s1 $\vee$ s2, p $\wedge$ q $\Rightarrow$ s3$\}$ gegeben.

Beispiel 1

Sei $Q = \langle p, q \rangle$, und db={p,q,s1,s2,s3}.

Die Antwort unter Ablehnung ist $\langle p, mum \rangle$, während die Antwort unter Lügen $\langle \neg p, q \rangle$ ist. Unter der kombinierten Methode ist die Antwort $\langle p, \neg q \rangle$. In diesem Fall haben Ablehnung und die kombinierte Methode die richtige Antwort abgeleitet. Aber weil log^L_1 und q zusammen kein pot_sec_disj ableiten können, wird unter Lügen die richtige Antwort q zurückgeben.

Sei $Q = \langle p, q \rangle$, und db={p,q,s1,s2,s3}.

Die Antwort unter Ablehnung ist <q, mum>, während die Antwort unter Lügen $\langle q, \neg p \rangle$ ist. Unter der kombinierten Methode ist die Antwort $\langle q, \neg p \rangle$. In diesem Fall leiten alle die gleichen richtigen Antworten ab.

Beispiel 2

Sei $Q = \langle p, r \rangle$, und db={p,q,s1,s2,s3}.

Die Antwort unter Ablehnung ist $\langle p, \neg r \rangle$, während die Antwort unter Lügen $\langle \neg p, \neg r \rangle$ ist. Unter der kombinierten Methode ist die Antwort $\langle q, \neg r \rangle$. In disem Fall hat die Ablehnung mehr richtige Antworten abgeleitet.

Beispiel 3

Q = {q, ¬p}, db={q}.

Die Antwort unter Ablehnung ist $\langle q, mum \rangle$, die Antwort unter Lügen ist $\langle q, \neg p \rangle$, die Antwort unter der kombinierten Methode ist $\langle q, \neg p \rangle$. In diesem Fall hat das Lügen mehr richtige Antworten abgeleitet.

3.6.2 Sicherheitsanforderung der Kombination von Lügen und Ablehnen

Basierend auf den einführenden Erklärungen, können wir die geforderten Sicherheitseigenschaften für die kombinierte Methode erhalten.

Theorem 1: Unter bekannten potentiellen Geheimnissen ist die kontrollierte Anfrageauswertung durch die Kombination von Lügen und Ablehnen $control\_eval^C_{ps}(Q, log_0)$ sicher gemäß Def.1, wobei wir ein Argument als adäquat betrachten, wenn „Für alle $\psi \in pot\_sec, log_0 \not\models \psi$“ gilt. [BB04a]

3.6.3 Kombination von Lügen und Ablehnen unter bekannten Heimlichkeiten

Die Modifikation besteht aus Nutzung einer anderen Invarianten und Bedingung.
Sei gegeben: $secr$, $Q = \langle \Phi_1, \Phi_2, ..., \Phi_i, ... \rangle$ und log_0,

$$\text{Für alle } \Psi \in secr, log_i \not\models \Psi \text{ und } log_i \not\models \neg\Psi.$$

Wir haben einen neuen Operator secrecy, der jedes Ψ auf die entsprechende Menge $\{\Psi, \neg\Psi\}$ abbildet.

$$control\_eval^C_{sec}(Q, log_0)(db, secr) = \langle (ans_1, log_1), (ans_2, log_2), ...(ans_i, log_i)... \rangle .$$

mit i-ter Antwort ans_i und i-tes log_i gegeben durch

ans_i = **if** (exists Ψ_1) [$\texttt{secrecy}(\Psi_1) \in secr$ und $log_{i-1} \cup \{eval^*(\Phi_i)(db)\} \models \Psi_1$]
then
if
(exist Ψ_2)
[$\texttt{secrecy}(\Psi_1) \in secr$ and $log_{i-1} \cup \{\neg eval^*(\Phi_i)(db)\} \models \Psi_2$]
then mum
else $\neg eval^*(\Phi_i)(db)$
else $eval^*(\Phi_i)(db)$
und
log_i:= **if** ans_i = mum **then** log_{i-1} **else** $log_{i-1} \cup \{ans_i\}$.

Wir fügen einige Beispiele hinzu, damit man die Theorie besser verstehen kann. In allen Beispielen gilt $secr = \{\{s1, \neg s1\}, \{s2, \neg s2\}, \{s3, \neg s3\}\}$, log_0 ={p ⇒ s1∨ s2, p ∧ q ⇒ s3} und $Q = \langle p, q \rangle$.

Beispiel 4

Sei db={p,q,s3}.

Die Antwort unter Ablehnung ist <p, mum>, die Antwort der kombinierten Methode ist $\langle p, \neg q \rangle$.

Beispiel 5

Sei db={p, ¬q, s3}.

Die Antwort unter Ablehnung ist <p, mum>, unter der kombinierten Methode ist die Antwort $\langle p, \neg q \rangle$.

Beispiel 6

Sei db ={p,¬q, ¬s3}.

Die Antwort unter Ablehnung ist <p, mum>, unter der kombinierten Methode ist die Antwort $\langle p, \neg q \rangle$.

Bei den drei Beispielen ist darauf hinzuweisen, dass die kombinierte Methode genauso kooperative ist wie die Ablehnung.

Jetzt wollen wir noch einen kurzen Blick auf „Naive Reduction“ werfen. Diese befasst sich mit der Methode, sichere kontrollierte Anfrageauswertungen unter potentiellen Geheimnissen zu sicheren kontrollierten Anfrageauswertungen unter Heimlichkeiten umzuwandeln.

$$pot\_sec(secr) := \bigcup_{S \in secr} S$$

Sei $control\_eval_{ps}$ eine kontrollierte Anfrageauswertung unter potentiellen Geheimnissen. Wir definieren ***naive reduction*** unter $control\_eval_{ps}$

$naive\_red(control\_eval_p\, s)$
$:= \lambda Q, log_0, db, secr.\ control\_eval_{ps}(Q, log_0)\ (db, pot\_sec(secr))$.

Die Schreibweise $\lambda Q, log_0, db, secr.\ control\_eval_{ps}(Q, log_0)\ (db, pot\_sec(secr))$ soll bedeuten, dass der durch $control\_eval_{ps}(Q, log_0)\ (db, pot\_sec(secr))$ bezeichnete „Rumpf“ mit den angegebenen „Argumenten“ $Q, log_0, db, secr$ ausgewertet werden soll. Speziell für M = L, R,

$$control\_eval^M_{sec} = naive\_red(control\_eval^M_{PS}).$$

3.7 Kontrollierte Anfrageauswertung zur Durchsetzung von Vertraulichkeitspolitik in vollständigen IS

3.7.1 Einführung

Es gibt drei wichtige Aspekte der kontrollierten Anfrageauswertung:

1. Vertraulichkeitspolitik:

   - Heimlichkeiten (secrecies) $secr = \{\{\Psi_1, \neg\Psi_1\}, ...\}$:
     Der Benutzer darf nicht die wahre Alternative ableiten können, daher werden beiden vor ihm versteckt.
   - Potentielle Geheimnisse (potential secrets) $pot\_sec = \{\Psi_1....\}$:
     Wenn einer der Sätze wahr ist, darf der Benutzer diese Wahrheit nicht ableiten können.

2. Das Bewusstsein (awareness):
   Es geht darum, ob der Benutzer weiss, was das Informationssystem verbergen möchte.

3. Durchsetzungsmethode (enforcement method):
   Diese besteht aus drei Methoden: Ablehnen, Lügen und die Kombination von Ablehnen und Lügen.

Wir haben schon gelernt, dass es zwei Anfrageauswertungsweisen gibt.

- Eine ist die einfache Anfrageauswertung:
  $eval(\Phi) : DS \longrightarrow \{true, false\}$ mit $eval(\Phi)(db) := db$ `model_of` (Φ)
- Die Zweite ist die kontrollierte Anfrageauswertung:

$$control\_eval(Q, log_0)(db, policy) =$$
$$\langle((ans_1^C(db), log_1^C(db)), (ans_2^C(db), log_2^C(db))), ..., (ans_i^C(db), log_i^C(db)), ...\rangle$$

3.7.2 Kontrollierte Anfrageauswertung unter unbekannten potentiellen Geheimnissen

Bis jetzt haben wir nur die Methoden unter bekannten potentiellen Geheimnissen betrachtet. Hier werden wir einen Blick auf kontrollierte Anfrageauswertung unter unbekannten potentiellen Geheimnissen werfen.
Unter unbekannten potentiellen Geheimnissen verwenden wir eine weniger beschränkte Disjunktion.

Zunächst definieren wir den Zensor, der nur abhängig von der aktuellen Instanz ist, für uniformes Lügen als $censor^{ps,unknown,L}$ wie folgt:

$$log \cup \{eval^*(\Phi)(db)\} \models true\_pot\_sec\_disj,$$

mit

$$true\_pot\_sec\_disj := \bigvee_{\Psi \in pot\_sec \text{ und } db \texttt{ model\_of } \Psi} \Psi \qquad (7)$$

und $log_0 \not\models true\_pot\_sec\_disj$.

Theorem 3 *Uniformes Lügen unter unbekannten potentiellen Geheimnissen hält die Vertraulichkeitspolitik gemäß Def.1 von Seite 21 ein.*

Beweis:
Sei pot_sec_1 eine Instanz, Q' sei ein Präfix von Q der Länge n, und db_1 mit (db_1, pot_sec_1) erfülle die notwendige Bedingung $log_0 \not\models true\_pot\_sec\_disj_1$.
Wir behaupten zuerst

$$log_i \not\models true\_pot\_sec\_disj_1.$$

Da beim Lügen-Zensor:

$$log_{i-1} \cup \{eval^*(\Phi_i)(db)\} \models true\_pot\_sec\_disj_1, \tag{8}$$

und

$$log_{i-1} \cup \{\neg eval^*(\Phi_i)(db)\} \models true\_pot\_sec\_disj_1 \ gilt, \tag{9}$$

impliziert log_{i-1} in jedem Fall nach (8) und (9) $true\_pot\_sec\_disj_1$, dies führt zu einem Widerspruch zur Annahme.

Für $\Psi \in pot\_sec_1$ müssen wir (db_2, pot_sec_2) erzeugen mit den Eigenschaften aus Def.1, entweder mit

- $eval^*(\Phi_i)(db_1) = \neg\Psi$:
  Dann nehmen wir (db_2, pot_sec_2)=(db_1, pot_sec_1), oder sonst mit
- $eval^*(\Phi_i)(db_1) = \Psi$:
  Wir wählen db_2 als einen Zeugen für $log_n \not\models true\_pot\_sec\_disj_1$.

Wir definieren $pot\_sec_2 = \emptyset$. Offensichtlich hat (db_2, pot_sec_2) die Bedingungen erfüllt.

Außerdem gilt

$$db_2 \texttt{ model\_of } log_n \tag{10}$$

und

$$db_2 \texttt{ model\_of } \neg true\_pot\_sec\_disj_1, \tag{11}$$

somit ist

$$db_2 \texttt{ model\_of } \neg\Psi. \tag{12}$$

Dann resultiert die Eigenschaft (a) aus (10), die Eigenschaft (b) resultiert aus (12).□

Theorem 4 *Uniformes Ablehnen unter unbekannten potentiellen Geheimnissen hält die Vertraulichkeitspolitik gemäß Def.1 von Seite 21 ein.*

Uniformes Ablehnen unter unbekannten potentiellen Geheimnissen ist im Prinzip ähnlich wie uniformes Lügen.

3.8 Zusammenfassung

Wir fassen nun unsere Kernthemen noch einmal zusammen:

Im zweiten Kapitel wurde eine Einführung in die Thematik gegeben. Es wurden die Ziele der Sicherheitspolitik erörtert und die zwei zugehörigen Ansätze (statischer und dynamsicher Ansatz) vorgestellt. Des Weiteren wurden essentiell wichtige Begriffe eingeführt wie die Begriffe „Anfrage“, „Lügen“, „Ablehnen“, „Heimlichkeiten“, „potentielle Geheimnisse“.

Der dritte Abschnitt befasste sich mit der Anfrageauswertung. Für die einfache Anfrageauswertung wurden die Funktionen $eval(\Phi)$ und $eval^*(\Phi)$ definiert, die der Auswertung der Anfrage dienen. Für die kontrollierte Anfrageauswertung wurde schliesslich noch ein Zensor dazugeschaltet, der die Aufgabe hatte, die gegebene Antwort zu beurteilen. Anschliessend richteten wir unsere Aufmerksamkeit auf die Sicherheitspolitik basierend auf bekannten und unbekannten Heimlichkeiten sowie auf die Sicherheitspolitik basierend auf bekannten potentiellen Geheimnissen.

In Kapitel vier betrachteten wir die Methode des Lügens unter bekannten potentiellen Geheimnissen. Dazu wurden die Anforderungen untersucht, die zum Lügen sichergestellt werden müssen. Der Benutzer kann unglaubwürdige Antworten unter der Methode des Lügens ausfindig machen, dazu betrachteten wir schliesslich ein Beispiel.

Im folgenden fünften Kapitel betrachteten wir die Methode des Ablehnens unter bekannten potentiellen Geheimnissen. Es wurden Unterschiede und Gemeinsamkeiten zur vorherigen Methode aufgedeckt und anschliessend die Errungenschaften (Sicherheitsbeweis) vorgestellt.

Im sechsten Kapitel identifizierten wir die Bedingungen für das Lügen und Ablehnen, damit beide Methoden funktional äquivalent sind. Dabei darf der initale Log nicht das schwächste potentielle Geheimnis implizieren und die potentiellen Geheimnisse müssen unter Disjunktion abgeschlossen sein. Wegen dieser Forderung kann unter der Lügen- oder Ablehnungsmethode genau die gleiche Information an den Benutzer mitgeteilt werden.

In Abschnitt sieben kombinierten wir beide Methoden (Lügen und Ablehnen), um die jeweiligen Schwächen der einzelnen Methoden zu minimieren. Es wurde die Kombination von Lügen und Ablehnen unter bekannten potentiellen Geheimnissen und unter bekannten Heimlichkeiten vorgestellt. Zusätzlich wurde eine kurze Einführung in die „Naive Reduktion“ gegeben.

Das achte und letzte Kaptitel gab eine kurze Einführung in die Durchsetzung von Vertraulichkeitspolitik in vollständigen Informationssystemen unter kontrollierter An-

frageauswertung. Speziell gingen wir hier auf die kontrollierte Anfrageauswertung bei der Lügenmethode unter unbekannten potentiellen Geheimnissen ein.

4 Relationale Datenbanken und Entscheidbarkeit 1

4.1 Relationale Datenbanken

Eine relationale Datenbank ist eine Datenbank, die auf dem relationalen Datenbankmodell basiert, das von Edgar F. Codd 1970 erstmals vorgeschlagen wurde. Darin ist eine Relation ein im streng mathematischen Sinn wohldefinierter Begriff, der im Wesentlichen ein mathematisches Modell für eine Tabelle beschreibt.

4.1.1 Die Struktur eines relationales Modell

Folgend werden einige Definitionen gegeben. Die Definitionen sind Grundlagen für Relationale Datenbanken und wurden an [AHV95] und [Bis95] angelehnt.

Definition 1 *Mit $dom(A)$ bezeichnen wir die Domaine eines Attributs A.*

Definition 2 *Mit dom bezeichnen wir die Vereinigung aller Domainen von Attributen,*

$$dom = dom(A_1) \cup dom(A_2) \cup ... \cup dom(A_n).$$

Definition 3 *Eine Relation R ist ein Paar R=(s,v) mit*

- *einem Schema $sch(R) = \{A_1, ..., A_n\}$, das aus einer endlichen Menge von Attributen (genauer: Attributnamen) besteht und für jedes Attribut A_i eine Domaine $dom(A_i) \in \{D_1, ..., D_m\}$ festlegt, und*
- *einer Ausprägung (auch Wert genannt) $var(R) \subseteq dom(A_1) \times dom(A_2) \times ... \times dom(A_n)$ ist eine endliche Menge.*

Definition 4 *Ein Element der Menge $val(R)$ wird als Tupel bezeichnet, dessen Stelligkeit sich aus dem Relationenschema ergibt.*

Definition 5 *Wir bezeichnen mit $sort(R) = \{A_1, ..., A_n\}$ die Menge der Attribute.*

Definition 6 *arity einer Relation R bezeichnet den Grad von R, also die Anzahl der Attribute in R.*

Definition 7 *Ein Datenbankschema ist eine Menge von Relationenschemas.*

Die entsprechenden Beispiele für die Begriffe werden bezüglich die Abbildung 2 gegeben.

Movies	*Title*	*Director*	*Actor*
	The Trouble with Harry	Hitchcock	Gwenn
	The Trouble with Harry	Hitchcock	Forsythe
	The Trouble with Harry	Hitchcock	MacLaine
	The Trouble with Harry	Hitchcock	Hitchcock
	...	...	...
	Cries and Whispers	Bergman	Andersson
	Cries and Whispers	Bergman	Sylwan
	Cries and Whispers	Bergman	Thulin
	Cries and Whispers	Bergman	Ullman

Location	*Theater*	*Address*	*Phone Number*
	Gaumont Opéra	31 bd. des Italiens	47 42 60 33
	Saint André des Arts	30 rue Saint André des Arts	43 26 48 18
	Le Champo	51 rue des Ecoles	43 54 51 60
	...	...	...
	Georges V	144 av. des Champs-Elysées	45 62 41 46
	Les 7 Montparnassiens	98 bd. du Montparnasse	43 20 32 20

Pariscope	*Theater*	*Title*	*Schedule*
	Gaumont Opéra	Cries and Whispers	20:30
	Saint André des Arts	The Trouble with Harry	20:15
	Georges V	Cries and Whispers	22:15
	...	...	...
	Les 7 Montparnassiens	Cries and Whispers	20:45

Abbildung 2: Datenbankschema Kino

- $D_{Kino} = \{R_{Movies}, R_{Location}, R_{Pariscope}\}$
- sort(R_{Movies}) = {Title, Director, Actor}
- arity(R_{Movies}) = 3
- name(R_{Movies}) = Movies
- sort($R_{Location}$) = {Theater, Address, Phone Number}
- name($R_{Location}$) = Location
- sort($R_{Pariscope}$) = {Theater, Title, Schedule}
- name($R_{Pariscope}$) = Pariscope

4.1.2 Anfragesprache

Es gibt verschiedene Konzepte für formale Sprachen zur Formulierung einer Anfrage an ein relationales Datenbanksystem:

1. Relationenalgebra (prozedural): Verknüpft konstruktiv die vorhandenen Relationen durch Operatoren $\wedge, \vee, \sigma, \Pi, -, \times, \rho, \bowtie, \div$.
2. Relationenkalkül (deklarativ): Beschreibt Eigenschaften des gewünschten Ergebnisses mit Hilfe einer Formel der Prädikatenlogik 1. Stufe unter Verwendung von $\wedge, \vee, \neg, \exists, \forall$.
3. SQL (kommerziell): Stellt eine in Umgangssprache gegossene Mischung aus Relationenalgebra und Relationenkalkül dar.
4. Query by Example (für Analphabeten): Verlangt vom Anwender das Ausfüllen eines Gerüstes mit Beispiel-Einträgen.

4.1.3 Inhärente Integritätsbedingungen des Relationenmodells

Definition 8 *Sei R eine Relation. Eine Attributmenge $K \subseteq sch(R)$ heißt Schlüsselkandidat, wenn zu jedem Zeitpunkt für je zwei Tupel t_1, $t_2 \subseteq val(R)$ gelten muss:*

$$t_1.K = t_2.K \Rightarrow t_1 = t_2$$

und wenn es keine echte Teilmenge von K gibt, die diese Eigenschaft hat.

Bemerkung: Jede Relation hat mindestens einen Schlüsselkandidaten. Ein Attribut einer Relation R, das in mindestens einem Schlüsselkandidaten vorkommt, heißt *Schlüsselattribut*.
Der *Primärschlüssel* einer Relation ist ein Schlüsselkandidat, der explizit ausgewählt wird.
Eine Attributmenge $F \subseteq sch(R)$ einer Relation R ist ein *Fremdschlüssel* in R, wenn es eine Relation R gibt, in der F Primärschlüssel ist. Das heißt, dass zu jedem Wert eines Fremdschlüsselattributs einer Relation R_2 ein gleicher Wert des Primärschlüssels in irgenteinem Tupel von Relation R_1 vorhanden sein muss.
Ein Attribut A eines Tupels t hat einen *Nullwert*, wenn der Wert $t.A$ undefiniert oder unbekannt ist. (Gedanklich werden alle Domains um einen solchen Nullwert erweitert.)
Beispiele:

- Vorrat für Produkte, die nicht gelagert werden, sondern bei Bedarf extern gekauft werden.
- Note von Studenten, die ihre Prüfung noch nicht abgelegt haben.

- Informatikkenntnisse von Studenten zu Beginn des Studiums.

Schreibweise:
In einem Relationenschema wird der Primärschlüssel unterstrichen; weitere Schlüsselkandidaten werden oft gestrichelt unterstrichen. Das Relationenmodell garantiert die folgenden Integritätsbedingungen:
Primärschlüsselbedingung:
Für jede Relation muss ein Primärschlüssel festgelegt werden. Der Primärschlüssel eines Tupels darf nie den Nullwert annehmen.
Beispiel:
Kunden (KNr, Name, Stadt, Saldo, Rabatt)
Produkte (PNr, Bez, Gewicht, Preis, Lagerort, Vorrat)
Bestellungen (BestNr, Monat, Tag, KNr, PNr, Menge, Summe, Status)
Fremdschlüsselbedingung:
Für jeden Wert eines Fremdschlüssels in einer Relation R muss in den referenzierten Relationen jeweils ein Tupel mit demselben Wert als Primärschlüssel existieren, oder der Wert des Fremdschlüssels muss der Nullwert sein.
Beispiel für Nullwert als Fremdschlüssel:
Produkte (PNr, ..., Lagerort)
Lager (Lagerort, Adresse, Verwalter, ...).
Das Attribut *Lagerort* von Relation *Produkt* ist ein Fremdschlüssel, weil ein Primärschlüssel (Lagerort) mit demselben Wert in den referenzierten Relation (Lager) existiert. Sonst muss der Wert von Lagerort in Produkte Null sein.

4.2 Entscheidbarkeitstheorie

Die Informationen der nächsten Abschnitte stammen von [BB06].

4.2.1 Offene Anfrage

Eine offene Anfrage ist eine Formel, in der mindestens eine freie Variable vorkommen muss. Im Allgemeinen, die Antwort zu solchen offenen Anfragen wird gegeben durch eine Menge von geschlossenen Anfragen, die wir durch Substitutionen von Variablen in offenen Anfragen bekommen können.
Zum Beispiel, wir haben eine offene Anfrage $IstKrank(x, Fieber)$. Die offene Anfrage kann man durch die folgenden geschlossenen Anfragen
$< IstKrank(Pete, Fieber), IstKrank(Tom, Fieber), IstKrank(Lisa, Fieber), ... >$
ersetzen.

4.2.2 Abbruch der Auswertung offener Anfragen unter festgelegter unendlicher Herbrand Domain

Wir wissen schon, dass wir offene Anfragen durch geschlossene Anfragen ersetzen können. Aber solche Substitutionen können unendlich sein. Um das Problem zu ver-

meiden, beschränken wir die Anfragen auf sichere und Domain-unabhängige Anfragen. Sichere Anfragen liefern immer ein endliches Ergebnis. Domain-unabhängige Anfragen bringen das gleiche endliche Ergebnis für jede aktive Domain zurück. Ein Beispiel für sichere und Domain-unabhängige Anfrage ist $IstKrank(x, Fieber)$.
Außerdem müssen wir auch effektiv erkennen, wann das endliche Ergebnis vollständig aufgezählt worden ist und keine weiteren Substitutionen betrachtet werden müssen. Wir können diese Anforderung durch einen einfachen Vollständigkeitstest erzielen. Im Allgemeinen, dieser Test fragt gerade, ob alle weiteren Substitutionen die Anfrage-Formel falsch bilden werden lassen.

4.3 Entscheidbarkeitsproblem in relationale Datenbanken

In diesem Kapitel wird ein Entscheidbarkeitstheorem bewiesen. Dazu müssen einige Definitionen und Lemmas, die die Grundlagen von diesem Theorem sind, noch gegeben werden. Um die Definitionen und Lemmas zu lernen, vereinbaren wir die Symbole, die in Definitionen und Lemmas vorkommen werden. Die folgenden Definitionen wurden angelehnt an [BB06].

L:	Logische Sprache für Prädikatenlogik der 1. Stufe
$dom(I)$:	Domain einer Interpretation I
p^I:	Interpretation für Prädikatensymbol p
c^I:	Interpretation für Konstantensymbol c
σ:	Substitution
φ und ψ:	Logische Formel

Definition 1 *Eine Interpretation I für L ist eine DB-Interpretation (DB-Instanz) genau dann, wenn die folgenden Bedingungen erfüllt werden.*

1. $dom(I) = dom$.
2. p^I *ist endlich für alle Prädikatensymbole* p *in* L.
3. $c_i^I = c_i$ *für alle Kanstantensymbole* c_i *in* L.

Definition 2 *Für jede Formelmenge* Γ *und jede Formel* φ*:*

1. $\Gamma \models_{DB} \varphi$, *wenn für alle DB-Interpretationen* I*:*
   $I \models \Gamma$ *impliziert* $I \models \varphi$
2. $\Gamma \models_{fin} \varphi$, *wenn für alle endlichen Interpretationen* I*:*
   $I \models \Gamma$ *impliziert* $I \models \varphi$
3. $\Gamma \models_{gen} \varphi$, *wenn für alle (klassischen) Interpretationen* I*:*
   $I \models \Gamma$ *impliziert* $I \models \varphi$

Definition 3 *Die aktive Domain $active_\varphi(I)$ einer Interpretation I bezgl. einer Formel φ ist eine Menge von Konstanten d, so dass d in einem Tupel von p^I mit $p \in L$ vorkommt, und alle c^I mit $c \in const(\varphi)$. Die formale Darstellung ist:*

$$active_\varphi(I) = \left\{ d \,|\, Es\, gibt\, d_1, ..., d_n, i < n, \langle d_1, ..., d_i, d, d_{i+2}, ..., d_n \rangle \in p^I \right\} \cup \left\{ c^I | c \in const(\varphi) \right\}.$$

Definition 4 *$Out_\varphi(x)$ ist eine Formel, die aussagt, dass x nicht zur aktiven Domain bzgl. φ gehört. Die formale Darstellung ist:*

$$(\forall y_1)...(\forall y_n) \bigwedge_p \bigwedge_{0 \le k \le arity(p)} \neg p(y_1, ..., y_k, x, y_{k+2}, ..., y_{arity(p)}) \wedge \bigwedge_{c \in const(\varphi)} \neg x = c,$$

dabei ist n die maximale arity für ein Prädikat p.

Die Formel $(\exists x) Out_\varphi(x)$ heißt non-totally assumption für φ. Die Formel sagt, dass die Domain der erfüllenden Interpretation nicht total durch die Interpretationen der Prädikatensymbole und der Konstanten abgedeckt wird, die in φ auftreten.

Definition 5 *Für alle Formeln φ ist UNA_φ (die unique Name Assumption für φ) durch die Formel*

$$\bigwedge \{ c_i \neq c_j \mid \{c_i, c_j\} \subseteq const(\varphi), und \quad i < j \}$$

definiert.

Definition 6 *Sei φ eine Formel der Prädikatenlogik 1. Stufe. φ hat nominelle Gleichung, wenn für alle Gleichungen $t = u$ in φ gilt:*

1. *t ist eine Variable.*
2. *u ist eine Konstante.*

Lemma 2 *Die Formel φ hat nominelle Gleichung und I sei eine Interpretation. Dann gilt für alle $c \in dom(I) \backslash active_\varphi(I)$, alle Substitutionen σ, alle Variablen v mit $\sigma(v) \notin active_\varphi(I)$:*

$$I, \sigma \models \varphi \quad gdw \quad I, \sigma[v/c] \models \varphi.$$

Das Lemma 1 bzw. die folgenden Lemma 2 und 3 sind Vorbereitungen für Theorem 1. Das Ziel ist es, eine unendliche Domain einer Interpretation auf eine endliche Domain zu beschränken.
Beweis von Lemma 1:
Induktionsanfang:
Der erste Fall ist , dass φ ein Atom ist und φ keine Gleichung hat. Dann haben wir zwei Situationen.

1. Wenn v nicht in φ vorkommt, gilt das Lemma in diesem Fall trivial. Da v nicht in φ auftritt, ist die Formel $I, \sigma[v/c] \models \varphi$ äquivalent mit der Formel $I, \sigma \models \varphi$.
2. Wenn v in φ vorkommt, gelten wegen der Voraussetzung $\sigma \notin active_\varphi(I)$ sowohl $I, \sigma[v/c] \not\models \varphi$ als auch $I, \sigma \not\models \varphi$ gelten.

Es ist auch möglich, dass φ eine Gleichung $x = c_j$ hat. Hier unterscheiden wir auch zwei Situationen.

1. Falls $x \neq v$, so gilt das Lemma 1 trivial.
2. Falls $x = v$, so bekommt man nach der Substitution die Gleichung $c \neq c_j^I$, da $\sigma(v) = c \notin active_\varphi(I)$ und $c_j^I \in active_\varphi(I)$. Dann gelten die beiden Aussagen $I, \sigma[v/c] \not\models \varphi$ und $I, \sigma \not\models \varphi$. Das Lemma ist bisher richtig.

Induktionsschritt:
Für $\neg$, $\cap$ und $\cup$ kann man die Behauptung sofort von der Induktionshypothese ableiten. Angenommen, φ ist die Formel $(\exists x)\psi$. Wir unterscheiden wieder zwei Situationen.

1. Falls $x = v$, so gilt das Lemma trivial, da wegen der Voraussetzung $\sigma(v) \notin active_\varphi(I)$ die beide Formeln $I, \sigma[v/c] \not\models \varphi$ und $I, \sigma \not\models \varphi$ gelten.
2. Falls $x \neq v$ und $I, \sigma \models \varphi$, so können wir φ aufbreiten und in der Formel $I, \sigma \models (\exists x)\psi$ umwandeln. Dann gilt $I, \sigma \models \varphi$, wenn $I, \sigma[x/d] \models \psi$ gilt. Durch Verwendung von Induktionsvorausetzung gilt auch die Formel $I, \sigma[x/d][v/c] \models \psi$, die äquivalent mit $I, \sigma[v/c] \models \varphi$ ist.

□

Lemma 3 *φ hat nominelle Gleichung und I sei eine DB-Interpretation. Dann gilt für alle $c \in dom(I) \backslash active_\varphi(I)$ und Substitutionen σ:*

$$I, \sigma \models \varphi \quad gdw \quad J, \tau \models \varphi.$$

J ist eine Einschränkung von I für $active_\varphi(I) \cup \{\bar{c}\}$, und

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

Beweis:
Wir nehmen an, dass φ Atom mit freien Variablen x_1,...,x_n ist und φ keine Gleichung ist. Dann unterscheiden wir zwei Situationen.

1. alle $\sigma(x_i) \in active_\varphi(I)$:
   Nach der Voraussetzung kann man feststellen, dass $\tau(x) = \sigma(x)$. Das Lemma 2 gilt trivial.

2. nicht alle $\sigma(x_i) \in active_\varphi(I)$:
   Sei $\sigma(x_i) \notin active_\varphi(I)$. Dann gilt $\tau(x) = \sigma(x) = c' \notin active_\varphi(J) = active_\varphi(I)$. Daraus folgt $I, \sigma \not\models \varphi$ und $J, \tau \not\models \varphi$.

Der zweite Fall ist, dass φ eine Gleichung $x = c_j$ ist. Hier unterscheiden wir wieder zwei Situationen.

1. alle $\sigma(x_i) \in active_\varphi(I)$:
   Nach der Vorausetzung kann man $\tau(x_i) = \varphi(x_i)$ feststellen. Die beiden Aussagen $I, \sigma \models \varphi$ und $J, \tau \models \varphi$ sind äquivalent. Das Lemma gilt trivial.

2. nicht alle $\sigma(x_i) \in active_\varphi(I)$:
   Sei $\sigma(x_i) \notin active_\varphi(I)$. Dann gilt $\tau(x) = \sigma(x) = c' \notin active_\varphi(I)$. Wegen $c_j \in active_\varphi(I)$ folgt $c' \neq c_j$. Dann gelten $I, \sigma \not\models \varphi$ und $J, \tau \not\models \varphi$.

Induktionsschritt:
Für $\neg$, $\cap$ und $\cup$ kann man die Behauptung sofort von der Induktionshypothese ableiten. Angenommen, φ ist die Formel $(\exists x)\psi$.
Es gelte $I, \sigma \models \varphi$. Dann existiert eine Konstante d mit $I, \sigma[x/c_k] \models \psi$. Nach der Induktionsvorausetzung gilt $J, \tau[x/c'] \models \psi$ mit

$$c' = \begin{cases} c_k & \text{falls } c_k \in active_\varphi(I) \\ \bar{c} & \text{sonst.} \end{cases}$$

Daraus folgt $J, \tau \models \varphi$. Umgekehrt gelte $J, \tau \models \varphi$. Dann existiert eine Konstante $c_k \in active_\varphi(I) \cup \{\bar{c}\}$, so dass $J, \tau[x/c_k] \models \psi$ gilt. Nach der Induktionsvorausetzung gilt auch $J, \sigma[x/c_k] \models \psi$. Daraus folgt die Formel $I, \sigma \models \varphi$.

□

Lemma 4 *Wir definieren zuerst ein paar Symbole.*
φ: *eine nominelle Gleichung Formel.*
J: *eine endliche Interpretation, so dass* $J \models (\exists x) OUT_\varphi(x)$.
f: $dom(J) \to dom$, *ein totale, injektive Funktion, so dass für alle* $c_i \in const(\varphi)$ $f(c_i^J) = c_i$.
I: *eine DB-Interpretation, so dass für alle Prädikate* p $p^I = \{\langle f(d_1), ..., f(d_n)\rangle \mid \langle d_1, ..., d_n\rangle \in p^J\}$.
Dann gilt für alle Substitutionen σ*:*

$$J, \sigma \models \varphi \quad gdw \quad I, (f \circ \sigma) \models \varphi$$

mit $(f \circ \sigma)(x) = f(\sigma(x))$.

Beweis:
Induktionsanfang:
Sei σ ein Atom. x_1,...,x_n seien die Variablen in φ. Wenn $J, \sigma \models \varphi$ gilt, kann man

feststellen, dass $\sigma(x_1)$,..., $\sigma(x_n) \in active_\varphi(J)$. Nach den Definitionen von f und I gehören $f(\sigma(x_1))$,...,$f(\sigma(x_n))$ auch zu $active_\varphi(J)$. Dann gilt $I, (f \circ \sigma) \models \varphi$. Wenn $J, \sigma \not\models \varphi$ gilt, kann man feststellen, dass nicht alle $\sigma(x_1), ..., \sigma(x_n) \in active_\varphi(J)$. Sei $\sigma(x_i) \notin active_\varphi(J)$. Nach den Definitionen von f und I ist $f(\sigma(x_i)) \notin active_\varphi(I)$. Dann gilt $I, (f \circ \sigma) \not\models \varphi$. Umgekehrter Beweis ist ähnlich.
Induktionsschritt:
Für $\neg$, $\cap$ und $\cup$ kann man die Behauptung sofort von der Induktionshypothese ableiten.
Sei φ die Formel $(\exists y)\psi$. Wenn $J, \sigma \models \varphi$ gilt, dann existiert ein $d \in dom(J)$, so dass $J, \sigma \models (\exists y)\psi$ gilt. Die Formel ist auch mit $J, \sigma[y/d] \models \psi$ äquivalent. Nach Induktionsvoraussetzung gilt auch $I, (f \circ \sigma[y/d]) \models \psi$. Da $f \circ (\sigma[y/d]) = (f \circ \sigma)[y/f(d)]$, gilt $I, (f \circ \sigma)[y/d] \models \psi$. Dann gilt auch $I, (f \circ \sigma) \models \varphi$.
Wenn $J, \sigma \not\models \varphi$ gilt, dann gilt $J, \sigma[y/d] \not\models \psi$ für alle $d \in dom(J)$. Nach Induktionsvoraussetzung gilt

$$I, (f \circ \sigma[y/d]) \not\models \psi. \tag{13}$$

Das bedeutet, dass es keine $c_k \in dom$ außerhalb der Wertebereiche von f gibt, so dass $I, (f \circ \sigma)[y/c_k] \models \psi$ gilt. Wir nehmen an, dass eine solche c_k existiert. Wegen des Lemmas 1 gilt $I, (f \circ \sigma)[y/c'] \models \psi$ für alle $c' \in dom \backslash active_\varphi(I)$. Eines dieser c' muss zu dem Wertebereich von f gehört, weil $J \models (\exists x)OUT_\varphi(x)$ gilt. Es gibt Widerspruch mit (1).

□

Theorem 5 *Für alle nominelle Gleichung Formel φ sind die folgenden Aussagen äquivalent:*

1. *φ gilt in einer DB-Interpretation ,*
2. *φ gilt in einem endlichen Modell J mit $(\exists x)OUT_\varphi(x)$ und UNA_φ.*

Beweis:
"$\Rightarrow$"
Angenommen wird die erste Aussage. Wegen Lemma 2 gibt es eine endliche Interpretation J mit

$$dom(J) = active_\varphi(I) \cup \{c\}, \quad c \notin active_\varphi(I) = active_\varphi(J),$$

so dass φ in endlicher Interpretation J gilt. Davon folgt, dass J $(\exists x)OUT_\varphi(x)$ und UNA_φ erfüllt. Da J $(\exists x)OUT_\varphi(x)$ und UNA_φ erfüllt, ist die zweite Aussage wahr.
"$\Leftarrow$"
Angenommen wird die zweite Aussage. Wir wählen eine Funktion f, die genau die Voraussetzungen in Lemma 3 erfüllt. Wegen Lemma 3 kann die erste Aussage abgeleitet werden.

□

Mit Hilfe von Theorem 1 können wir Vollständigkeitstests durchführen. Weil wir die DB-Interpretation I auf eine endlichen Interpretation J beschränkt haben, ist der Vollständigkeitstest auch endlich.

R	A	B
	1	2
	0	1
	3	4

Tabelle 1: endliche Relation

5 Relationale Datenbanken und Entscheidbarkeit 2

5.1 Einführung

Bei der Anfrageauswertung sind wir am meisten an zwei Sachen interessiert, und zwar:

1. ein Ergebnis zu erhalten
2. das erhaltene Ergebnis soll endlich sein

Um ein Ergebnis zu erhalten, soll das Informationssystem *IS* über bestimmte Eigenschaften verfügen, damit das Implikationsproblem, das bei der Anfrageauswertung entsteht, entscheidbar ist. Um eine endliche Ergebnisrelation als Antwort auf unsere Anfrage zu erhalten, soll die Anfrage richtig eingeschränkt werden.

5.2 Das DB-Submodell für Informationssystem

Ein Modell besteht aus einem *Schema DS* und einer *DB-Instanz*, wobei das *Schema DS*, die endliche Menge der Prädikatensymbole
$\mathbf{P} = \{P1, P2, ..., Pn\}$, $n \in \mathbf{N}$, und die unendliche Menge $dom = \{c1, c2, ..., ci, ..\}$ von Konstanten erfasst.
Eine *DB-Instanz* ist eine Herbrand-Struktur (Interpretation), die die unendliche Menge *dom* als Herbrand-Universum interpretiert. Da wir eine funktionsfreie Prädikatenlogik haben, bedeuten Konstanten in dieser Herbrand-Struktur in der Menge *dom* sich selbst.
Ausserdem sind alle Unique-Axiome für die Menge *dom* erfüllt, das heisst, für je zwei unterschiedliche Konstanten *c, c'*, die in dieser Herbrand-Struktur vorkommen, gilt: $\neg c \neq c'$.
Ausserdem hat diese Herbrand-Struktur nur endlich viele positiv interpretierte Atome (dargestellt als eine endliche Relation):

Und als letzt wird ein unendliches Komplement durch eine „closed world assumption“ erfasst [BB06].

Nachdem wir die grundlegenden Eigenschaften von einem Informationssystem *IS* beschrieben haben, werden wir mit ein paar Fragen konfrontiert:

- Was für Implikationen haben wir?

R	**A**	**B**
	1	2
	0	1
	3	4
	.	.
	.	.
	.	.

Tabelle 2: unendliche Relation

- Ist das Implikationsproblem entscheidbar?
- Existiert eine passende Struktur, in der das Implikationsproblem entscheidbar ist?

Wir betrachten hier drei Arten von Implikationen:

1. Klassische Implikation, die besagt :
   $\Phi \models \Psi$ gdw
   für alle Interpretationen **I** gilt: wenn Φ unter **I** erfüllt ist, dann ist auch Ψ unter **I** erfüllt.

2. Endliche Implikation, die besagt :
   $\Phi \models_{fin} \Psi$ gdw
   für alle endlichen Interpretationen I_{fin} gilt: wenn Φ unter I_{fin} erfüllt ist, dann ist auch Ψ unter I_{fin} erfüllt.

3. DB-Implikation, die besagt :
   $\Phi \models_{DB} \Psi$ gdw
   für alle DB-Instanzen I_{DB} gilt: wenn Φ unter I_{DB} erfüllt ist, dann ist auch Ψ unter I_{DB} erfüllt.

Diese drei Implikationsprobleme fallen zusammen und sind entscheidbar für ein Fragment der Prädikatenlogik (auf der unser Informationssystem *IS* aufgebaut ist) aufgrund weiter eingeführten Definitionen. Als eine Voraussetzung benutzen wir eine schon bekannte Definition im Bezug auf endliche Modelltheorie und klassische Modelltheorie und eine häufig untersuchte gegenseitige Reduktion zwischen Implikationsproblem und Allgemeingültigkeitsproblem.

Definition.1 Sei **L** ein Fragment der Prädikatenlogik. Dann sagen wir, **L** hat *endliche Modelleigenschaft*, falls für alle Aussagen φ aus **L** gilt: wenn φ *endlich* universell (allgemeingültig) ist, dann ist φ universell, d.h.:

wenn für alle *endlichen* Interpretationen I_{fin} : $I_{fin} \models \varphi$,

dann für alle (endlichen oder unendlichen) Interpretationen **I**: $\mathbf{I} \models \varphi$.

Daraus folgt, dass wenn wir ein Fragment mit endlicher Modelleigenschaft haben, dann hat das Fragment ein entscheidbares Allgemeingültigkeitsproblem. Und zwar

durch die Vollständigkeitseigenschaften der Prädikatenlogik ist das Allgemeingültigkeitsproblem rekursiv aufzählbar und durch die endliche Modelleigenschaft ist das Komplement des Problems auch rekursiv aufzählbar (alle endliche Interpretationen untersuchen). Die gegenseitige Reduktion des Implikationsproblems und Allgemeingültigkeitsproblems gibt an, dass für eine beliebige Art der Implikation $\models$ ($\models_{DB}$, $\models_{fin}$ oder $\models$) eine Implikation $\Phi \models \Psi$ zwischen zwei Aussagen(geschlossener Formeln) Φ und Ψ äquivalent ist zu der entsprechenden Version der Allgemeingültigkeit der Formel $\neg\Phi \vee \Psi$ [BB06].

Das folgende Theorem [BB06] beweist Äquivalenz und Entscheidbarkeit von diesen drei Implikationsproblemen im Bezug auf die erhaltenen Eigenschaften.

Theorem 1. Sei **L** ein Fragment der Prädikatenlogik mit folgenden Eigenschaften:

- mit nomineller Gleichung, abgeschlossen unter $\vee$,
- mit endlicher Modelleigenschaft,
- und mit Fähigkeit die negierte non-totality assumption $\neg(\exists x)Out_{\varphi}(x)$ und die negierte unique-name assumption $\neg Una_{\varphi}$ für alle **L**-Formeln φ auszudrücken.

Dann sind die Einschränkungen zu $\neg\mathbf{L} \times \mathbf{L}$ von den Implikationsbeziehungen $\models$, $\models_{fin}$ und $\models_{DB}$ alle entscheidbar.

Ausserdem, für alle $\psi \in \neg\mathbf{L}$ und $\varphi \in \mathbf{L}$ sind die folgenden Eigenschaften äquivalent:

1. $\psi \models_{DB} \varphi$;
2. $\psi \wedge (\exists x)Out_{\varphi}(x) \wedge Una_{\varphi} \models \varphi$;
3. $\psi \wedge (\exists x)Out_{\varphi}(x) \wedge Una_{\varphi} \models_{fin} \varphi$.

Wir verwenden das Theorem für das Fragment mit nomineller Gleichung von der *Bernays-Schönfinkel Klasse*.

Definition 2. Jede Aussage in *Bernays-Schönfinkel Klasse* ist in Pränex-Normalform und hat den Präfix $\forall^*\exists^*$, der in dieser Reihenfolge steht.
Das Allgemeingültigkeitsproblem ist für Bernays-Schönfinkel Klasse entscheidbar, und zwar, sowie für klassische Allgemeingültigkeit als auch für endliche Allgemeingültigkeit. Wenn alle angemessenen Aussagen (d.h. geschlossene Formeln) haben Pränex-Normalform und entweder den Präfix $\exists^*$ oder den Präfix $\forall^*$, dann ist die Implikation $\psi \models \varphi$ zwischen diesen zwei Formeln äquivalent zu der Allgemeingültigkeit der Aussage $\neg\psi \vee \varphi$, die in der Bernays-Schönfinkel Klasse ist.

Definition 3. Pränex-Normalform ist eine geschlossene Formel (d.h. in der Formel dürfen nur Konstanten oder gebundene Variablen vorkommen) von der Form

$Q_1 y_1 Q_2 y_2 ... Q_k y_k F$

mit k $\geq$0 und $Q_1, ..., Q_k \in \{\forall, \exists\}$

und in F darf kein Quantor vorkommen. D.h. eine Aussage in der Prädikatenlogik erster Stufe befindet sich in Pränex-Normalform, wenn alle Quantoren ausserhalb bzw. vor der eigentlichen Formel stehen. In der Prädikatenlogik gibt es zu jeder Formel (mit Konstanten, gebundenen Variablen und frei vorkommenden Variablen) eine logisch äquivalente Formel in Pränex-Normalform. Frei vorkommenden Variablen werden schliesslich allquantifiziert.

Beispiel 1. Der folgende Ausdruck

$\forall x P(x) \wedge \forall y Q(y) \wedge \forall z R(z)$

kann in folgende Pränex-Normalform übersetzt werden:

$\forall x \forall y \forall z P(x) \wedge Q(y) \wedge R(z)$.

5.3 Anfragen

Eine Anfrage ist eine prädikatenlogische Formel über *Schema DS* und Konstantenmenge *dom*, die aus Konstanten besteht, die sich selbst bedeuten, mit *sicherer* und *domain-unabhängiger* Auswertung, d.h. die Ergebnisrelation der positiv ausgewerteten Aussagen

- ist endlich,
- enthält nur Konstanten aus „aktiver Domaine“, und
- unendliches Komplement wird als „closed world assumption“ behandelt.

Die folgenden Definitionen erläutern einige oben genannten Begriffe.

Definition 4. *Sichere* Anfragen sind solche Anfragen , die für jeden Datenbankzustand ein endliches Ergebnis liefern.
Seien R, R_1, R_2 Relationen.
Eine *sichere* Anfrage [Wei06] hat die Form
$\{t | t \in R \wedge \varphi(t)\}$
und jede existenzquantifizierte Teilformel von $\varphi(t)$ die Form
$\exists t_1 : t_1 \in R_1 \wedge \varphi_1(t_1)$
und jede allquantifizierte Teilformel von $\varphi(t)$ die Form
$\forall t_2 : t_2 \in R_2 \Rightarrow \varphi_2(t_2)$.

Beispiel 2. für nicht *sichere* Anfrage

$\{t | \neg(t \in \mathbf{R})\}$

-liefert eine unendliche Menge, wenn die Datenbank unendlich ist.

Beispiel 3. für *sichere* Anfragen

$\{t|t \in \mathbf{R} \wedge \exists s : (s \in \mathbf{R} \wedge s.A = t.A)\}$,
$\{t|t \in \mathbf{R} \wedge \forall s : (s \in \mathbf{R} \Rightarrow s = t)\}$.

Definition 5. Eine Anfrage q heisst *domain-unabhängig*, falls für jede DB-Instanz **DB** und jedes Paar **d, d'** $\subseteq$ *dom* mit aktiver Domain $active_q(\mathbf{DB}) \subseteq$ **d** und $active_q(\mathbf{DB}) \subseteq$ **d'** gilt :

$q_{\mathbf{d}}(\mathbf{DB}) = q_{\mathbf{d'}}(\mathbf{DB})$.

Daraus ergibt sich, dass eine Anfrage genau dann *domain-unabhängig* ist, wenn sie für jede mögliche Ausprägung von der DB-Instanz, aber nicht von den zugrunde liegenden Domains abhängt [AHV95].
Aufgrund oben genannter Einschränkungen für Anfragen haben wir die zweite Frage auch positiv geantwortet. Wir haben das gewünschte Ergebnis erhalten, und zwar eine endliche Ergebnisrelation als Antwort auf unsere Anfrage.

5.4 Anwendungen für das Fragment (Bernays-Schönfinkel Klasse)

Die Bernays-Schönfinkel Klasse hat die folgenden möglichen Anwendungen:

- positive oder negative Antworten auf geschlossene Anfragen des positiven existentiellen Relationskalküls, der auszudrücken ermöglicht:
  - Gleichheit(Selektion),
  - Konjunktion(Verbund),
  - Disjunktion(Vereinigung),
  - existentielle Quantifikation(Projektion);
- potentielle Geheimnisse der selben Art;
- funktionale Abhängigkeiten oder Verbundabhängigkeiten als semantische Bedingungen;
- beliebige Konjunktionen und Disjunktionen davon [BB06].

Definition 8. Eine positive existentielle Relationskalkülanfrage ist eine domain-unabhängige Kalkülanfrage
$q = \{e_1, ..., e_n|\varphi\}$ möglich mit Gleichheit, in der die logischen Verknüpfungen nur $\wedge, \vee$ und $\exists$ sind. Es ist entscheidbar, ob eine Anfrage q mit diesen logischen Verknüpfungen domain-unabhängig ist.
Theorem 2. Der positive existentielle Relationskalkül ist äquivalent zu der Familie der konjunktiven Anfragen mit Vereinigung [AHV95].
Definition 9. Sei **R** ein Datenbankschema und *relname* Menge der Relationsnamen. Eine regelbasierte konjunktive Anfrage über **R** ist ein Ausdruck der Form:

$Ans(u) \leftarrow R_1(u_1), ..., R_l(u_l)$, wobei:

- $l > 0$,
- $R_1, ..., R_l \in \mathbf{R}$, $Ans \in relname/\mathbf{R}$, und
- $u, u_1, ..., u_l$ freie Tupel (Tupel mit vorkommenden freien Variablen) der Stelligkeiten $arity(Ans), arity(R_1), ..., arity(R_l)$, so dass jede Variable, die in u vorkommt, auch in mindestens einem der Tupel $u_1, ..., u_l$ vorkommt.

Beispiel 4.

$Ans(X_{Kino}, X_{Adr}) \leftarrow$ Filme(X_{Titel}, X_{Regie}, "George Clooney"),

Programm($X_{Kino}, X_{Titel}, X_{Zeit}$),

Orte($X_{Kino}, X_{Adr}, X_{Tel}$) .

Welche Kinos (Name & Adresse) spielen einen Film mit "George Clooney")

6 Oracle DBMS Teil 1

6.1 Über Oracle

Oracle Database ist eine relationale Datenbank der Firma Oracle. Oracle Database ist eine Datenbankmanagementsystem-Software. Oracle gilt als Marktführer im RDBMS-Segment. Im Großrechner-Bereich sind Sun Fire Maschinen mit dem Unix-System Solaris oder IBM-Maschinen häufig verwendeten Plattformen.

6.1.1 Oracle-Werkzeuge

Oracle stellt ein reichhaltiges Angebot an Werkzeugen für das Entwerfen und die Wartung von Datenbanken bereit. Einige wichtige Oracle-Werkzeuge sind:

- **RDBMS Kernel** Die Datenbank-Engine, Oracles Arbeitspferd.
- **SQL** Die relationale Abfragesprache.
- **SQL*PLUS** Oracles Benutzerschnittstelle zu SQL und Datenbank.
- **PL/SQL** Die „Procedural Language SQL“, die eine prozedurale Verarbeitung von SQL-Anweisungen ermöglicht.
- **SQL*Loader** Ermöglicht das Entfernen und Einfügen von Daten und Strukturinformationen aus Oracle Datenbanken in bzw. aus Archiven.
- **JDBC** Oracles Implementierung der Schnittstelle „Java Database Connection“.
- **JAVA** Java befindet sich in den Releases Seit 8.1 und in allen *9i*-Releases im Oracle-Kernel.

  Und es gibt noch viele Werkzeuge mehr.

Die Informationen dieses Abschnitts stammen größtenteils von [Aul03] und [ACA02].

Name	Null?	Type
EMP_LAST_NAME		VARCHAR(20)
EMP_FIRST_NAME		VARCHAR(15)
EMP_MIDDLE_NAME		VARCHAR(15)
SALARY		NUMBER(7,2)
DEPARTMENT		NUMBER(2)
POSITION		VARCHAR2(15)

Tabelle 3: Datentype in DB

6.2 Sicherheitsimplementierung bei Oracle

6.2.1 Hintergründe der Sicherheit bei Oracle

Einer der ersten Datenbank-Kunden von Oracle war die Central Intelligence Agency (CIA). Um die Auswirkungen dieser frühen Beziehung zwischen Oracle und der CIA zu verstehen, wollen wir uns anschauen, was passiert, wenn Sie in einer aktuellen Version des Oracle-RDBMS versuchen, auf eine Tabelle zuzugreifen, deren Name falsch geschrieben wurde.
Hier der Aufbau der Tabelle EMPLOYEES im Schema MYSTER: (Tabelle 3)

- describe EMPLOYEES

Das Ergebnis wurde in der Tabelle 3 dargestellt.

Nun nehmen wir an, dass Sie die folgende Abfrage stellen und dabei den Tabellennamen falsch schreiben:

- select EMP_LAST_NAME
  from ELMPOYEES;

ERROR at line 2:
ORA-00942: table or view does not exist

Wir erhalten eine Fehlermeldung mit der Nummer ORA-00942: „Table or view does not exist.“ Es wird erzählt, dass diese Fehlermeldung als eine der ersten Sicherheitsmaßnahmen bei Oracle eingeführt und von der CIA angefordert wurde, um eventuelle Hacker zu verunsichern. Dahinter stand der Gedanke, dass Hacker, die versuchen den Namen eines Datenbank-Objekts zu erraten, durch eine Fehlermeldung wie „Tabellenname falsch geschrieben“ erfahren würden, dass sie auf dem richtigen Weg sind.
Wir erhalten dieselbe Fehlermeldung, wenn zwar der Name korrekt ist, uns aber die Berechtigungen für diese Tabelle fehlen.

Backups Es gibt viele verschiedene Möglichkeiten, wie Daten im System oder in der Datenbank verloren gehen können:
Benutzerfehler, Entwicklerfehler, Administratorfehler, Hacker-Fehler, Hardware-Fehler und Naturkatastrophen.
Man muss für alle der aufgeführten Notfälle gerüstet sein, um sicherzustellen, dass die Daten geschützt sind.

Betrachten wir nun die ersten Backup- und Recovery-Optionen von Oracle, um ein Gefühl für die Richtung zu bekommen, die Oracle einschlug, um Benutzer bei der Sicherheit in einer Oracle-Datenbank zu unterstützen.

Ursprünglich bot Oracle zwei Tools names „**Export**“ und „**Import**“ an. Mit dem Export-Tool können wir eine Momentaufnahme (*Snapshot*) erstellen, von einer oder mehreren Tabellen oder der gesamten Datenbank zu einem bestimmten Zeitpunkt des Exports.
Mit dem Import-Tool können wir dann die gespeicherten Objekte wiederherstellen.
Durch das Export- und Import-Tool haben wir also die Möglichkeit, eine neue Datenbank so zu erstellen und mit Daten zu füllen, dass sie genau wie eine bestehende Datenbank aussieht.

Die Sicherheit robuster machen Bis zur Version 6 des RDBMS gab es nur drei Berechtigungsebenen, die Benutzern zugewiesen werden konnten, wenn sie sich bei der Datenbank anmelden wollten: die Berechtigungen **Connenct**, **Resource** und **DBA**.

1. Mit der *Connect*-Berechtigung konnte der Benutzer gerade mal eine Sitzung eröffnen und vielleicht ein, zwei Dinge mehr tun.

2. Die *Resource*-Berechtigung war für Personen gedacht, die Objekte wie zum Beispiel Tabellen, Indizes und Views in der Datenbank anlegen wollten.

3. Die Berechtigung *DBA* ermöglichte jedem, der sie erhielt, die vollständige Kontrolle über die Datenbank.

Version 6 und neue Sichrheitsansätze Zu der Zeit, zu der Oracle Version 6 ausgeliefert wurde, fordeten die Benutzer mehr Sicherheits-Features, die leichter zu implementieren waren. Die drei Pseudo-Privilegien, die bisher im Oracle RDBMS existierten, wurden nun durch ein neues Konzept namens *Rollen (Roles)* ersetzt. Jede der neuen Rollen enthielt einen eindeutigen Satz an Berechtigungen, die sie einem Benutzer zuweisen konnten, und jede Rollenebene wies diesen Benutzern mehr Freiheiten und Berechtigungen zu. (Tabelle 4)

Rolle	Zugewiesen an	Berechtigungen
connect	Benutzer	Verbinden mit der Datenbank; (bzw.Views) synonyme und Datenbank-Links anlegen; Daten verändern; Benutzer-und Tabellenexporte durchführen
resource	Entwickler	Berechtigungen auf Tabellen, Indizes, Cluster und Sequenzen erstellen und vergeben
dba	Datenbankadministratren	Berechtigungen vergeben und entziehen; Public Objects erstellen; Benutzer anlegen und ändern; Benutzerdaten einsehen; alle Arten von Exporten durchführen

Tabelle 4: Verfügbare Berechtigungen in Version 6

Das Problem bei diesen neuen Rollen bestand darin, dass es keine Zwischenstufen gab. Man konnte keine eigenen Rollen anlegen, um die Berechtigungen einzuschränken, die man einem Benutzer zuteilen wollte. Man konnte individuelle Berechtigungen zuweisen, dies aber nicht so leicht auf eine ganze Gruppe ausdehnen.

Was ist mit Auditing?

Auditing (von lat. audire, „hören"): Auf Rechnern mit sehr sensiblen Daten will man genau wissen, wer wann was gemacht hat. Das detaillierte Protokollieren dieser Informationen auf einem Betriebssystem bezeichnet man als Auditing.

Ab Version 6 bot Oracle viele Optionen zum Überwachen der Datenbank an. Wenn man in dieser Version das Auditing aktivierte, wurden alle Überwachungsaktionen in eine Tabelle namens SYS.AUD$ geschrieben.
Standardmäßig wurden sowohl erfolgreiche wie auch erfolglose Befehle protokolliert. Oracle ermöglichte es Administratoren, die Audit-Daten zu löschen oder zu verschieben.
Action Audits zeichnen jede Aktion auf, die ein Datenbankobjekt betrifft, und sind Überwachungen auf Systemebene. In Version 6 konnten nur Connect, Ressource und DBA auditiert werden.
Beispiel:

- audit CONNECT

Object Audits verfolgen, wie der Name schon vermuten lässt, die Änderungen an Objekten. Man kann einen Audit-Datensatz jedes Mal erstellen lassen, wenn ein Benutzer Daten in einer bestimmte Tabelle einfügt.
Um jede Aktion zu auditieren, die an einer Tabelle vorgenommen wird, kann man den Befehl

- audit all on

verwenden.

Oracle 7 In den frühen 90ern hatte sich die Client/Server-Technologie in den Unternehmen ausgebreitet, und nun gab es überall PCs und überall Datenbanken! Mit der Veröffentlichung von Oracles Version 7 versuchte Oracle, die Sichtweise auf seine Produkte zu ändern, daher veränderte man den Namen in „Oracle7“. Mehr und mehr Features wurden eingeführt, um immer größere Datenbanken zu unterstützen. Ab Version 7.3 konnte man auf die Daten über Views fast genauso einfach zugreifen, wie direkt über Tabellen.
Wenn in der VIEW beispielsweise verschiedene Spalten (z.B. mit persönlichen Daten) ausgeblendet sind, so können diese auch später nicht mehr ausgegeben werden. Man könnte auch sagen, eine VIEW ist eine gefilterte Tabelle (bestehend aus einer oder mehreren Tabellen).

Mehr und mehr Unternehmen entdeckten die Vorteile, die sich aus der Bereitstellung großer Datenmengen in riesigen Tabellen innerhalb von Datenrepositories, so genannten Data Warehouses, ergaben (Repository = Lager).
Mit einem Data Warehouse konnten die verschiedenen Führungsebenen gleichzeitig auf dieselben Daten zugreifen und damit zeitnahere und objektivere Entscheidungen treffen. Data Warehouses bieten einen Weg, um die Firmendaten einfacher in mehrere Mitarbeiterebenen zu verteilen.

Ab Version 7.3 fügte Oracle eine Erweiterung namens **Union all** hinzu, die es in Kombination mit dem Befehl **create view** den Entwicklern ermöglichte, die Daten einer riesigen Tabelle in mehrere kleinere Tabellen aufzuteilen.

Diese Informationen kann man dann in einer großen View wieder logisch miteinander verknüpfen.

Um zu verstehen, wie das funktioniert, nehmen wir an:

- Wir haben eine Tabelle, in der wir die Verkaufsdaten zu unseren Produkten speichern.
- Die Tabelle hat Millionen von Zeilen und enthält eine Datumsspalte.
- Wir wollen sehen, wie viele Artikel im Januar verkauft wurden.

Oracle musste in jede Zeile dieser riesigen Tabelle schauen, um herauszufinden, ob sie dazu passt. Die Abfrage ist einfach und übersichtlich, aber sehr zeitaufwendig.

Jetzt erstellen wir ein Tabelle für jeden Monat des Jahres. Die erste Tabelle ist JANUARY_ORDERS, mit den Aufträgen im Januar.

Wenn man alle 12 Tabellen hat, kann man mit Hilfe von **union all** alle 12 Tabellen in einen großen Tabelle kombinieren. Ein Teil der Syntax könnte wie folgt sein:

- create view ALL_ORDERS as
  select ORDER_ID, CUSTOMER_NAME, PART_NO
  from JANUARYY_ORDERS
  union all
  select ORDER_ID, CUSTOMER_NAME, PART_NO
  from FEBRUARY_ORDERS
  union all ...

Diese Ansatz erwies sich als sehr fehleranfällig und lästig. Wenn man allerdings die Daten eines bestimmten Monats sehen wollte, funktionierte es wunderbar. Bei Anzeige sämtlicher Daten oder der von Januar, März und Mai konnte die Abfrage groß und unhandlich werden. Oracle begann dieses Problem mit der Version Oracle 8.0 zu lösen.

Während dieser Zeit entstand der Begriff *VLDB (Very Large Database)*, was Datenbank mit mehreren Gigabytes an Daten repräsentierte. (Heutzutage gibt es Datenbanken im Terabyte_Bereich und darüber hinaus).

Audit-Verbesserungen Ab Version 7 konnte man Trigger zum Erweitern der Überwachungsmöglichkeiten nutzen.
Früher konnte man Änderungen an einer Tabelle protokollieren und mitteilen, wer die Änderungen wann vorgenommen hat. Aber es wurde nicht protokolliert, welche Daten geändert wurden.

Mit den selbst geschriebene Triggern konnte man die Daten vor und nach der Änderung aufzeichnen. Dies ist besonders wichtig beim Löschen von Daten.
Bei einer bestimmten Art der Änderungen (z. B. INSERT, UPDATE, DELETE bei SQL) von Daten in einer Tabelle wird ein gespeichertes Programm aufgerufen, das diese Änderung erlaubt, verhindert und/oder weitere Tätigkeiten vornimmt.
Trigger werden u. a. zur Wahrung der Datenkonsistenz (Integritätschecks) und zum Einfügen, Löschen oder Ändern von Referenzdaten eingesetzt. Der Trigger wird ausgeführt, wahlweise bevor die Änderung an der referenzierten Tabelle vorgenommen wird oder danach.

Backup-Verbesserungen In Version 7 bot Oracle eine neue Möglichkeit namens **Hot Backups**, die es uns ermöglichte, eine Kopie der Datenbankdateien zu machen, während die Datenbank lief.
Allerdings waren **Hot Backups** umständlich und führten zu einigen schwer wiegenden Problemen:
Wurde die Datenbank heruntergefahren oder stürzte ab, solange das System sich in Hot backup Modus befand, hatten wir Schwierigkeiten, die Datenbank wiederherzustellen.

Oracle 8 Mit den größeren Datenbanken verlagerte sich der Schwerpunkt auf eine bessere Unterstützung von VLDBs mit Oracle 8. Die Nutzung von Views, die union all verwendeten, genügte nicht allen Kunden. Mit Oracle8 führte Oracle partitionierte Tabellen und Indizes ein. Große Tabellen und Indizes konnte man in kleinere Tabellen und Indizes aufteilen.
Das ermöglichte ein ungehindertes Wachsen von der Datenbank.

Verbesserung der Kennwortverwaltung Mit der Veröffentlichung von Oracle8.0 gab es nun diverse Passwortmanagement-Optionen. Man konnte nun:

- Kennwörter beim nächsten Mal verfallen lassen.
- Kennwörter erzwingen, die bestimmten vorgegebenen Mustern entsprechen.
- Eine Kennwort-Historie verwalten, damit nicht dasselbe Kennwort immer wieder benutzt wird.
- Ein Benutzerkonto unter Oracle explizit sperren.

Datensicherung und Wiederherstellung Ein Tool namens RMAN wurde eingeführt, das das Sichern und Wiederherstellen von Daten verbessert.
Mit RMAN kann man ein komplettes Backup auf Dateiebene von allen Datenbankdateien durchführen, während die Datenbank verfügbar ist.

6.2.2 Oracle 8*i* und das Internet

Die Release Oracle8*i*, die als erste Release von Oracle speziell dafür konzipiert wurde, elektronischen Handel im Internet zu unterstützen (als E-Commerce bezeichnet), bot diverse neue Sicherheits-Features und viele Verbesserungen.

Fähigkeiten (Features) von Oracle 8*i* Advanced Security Dazu gehört die Möglichkeit, Daten verschlüsselt mit dem SSL-Protokoll zu transportieren und zu empfangen.
Secure Sockets Layer (SSL) ist ein Verschlüsselungsprotokoll für Datenübertragungen im Internet.

Fähigkeiten (Features) von Oracle 8*i* Advanced Security SSL nutzt zunächst einen Public Key-Algorithmus, um einen geheimen, zufällig generierten Schlüssel auszutauschen, der für jede Benutzersitzung einmalig ist.
Unter einem public key versteht man in asymmetrischen Kryptosystemen Schlüssel, die jedem bekannt sein dürfen und zur Verschlüsselung eines Klartextes in einen Geheimtext genutzt werden können.

Kennt ein Angreifer den öffentlichen Schlüssel, so kann er daraus dennoch weder auf den geheimen Schlüssel noch auf die verschlüsselte Nachricht schliessen.

Die Geheimtexte können hierbei später nur mit dem geheimen Schlüssel wieder entschlüsselt werden.

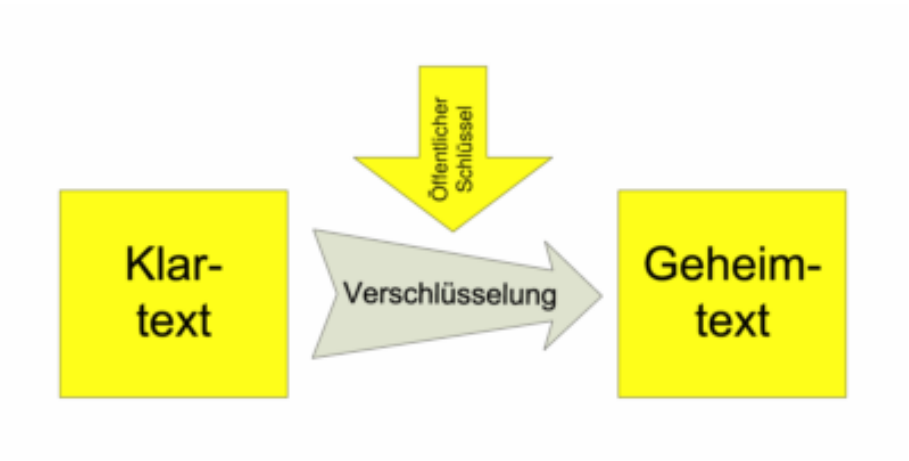

Abbildung 3: Verschlüsselung mit Public key

Wenn der Schlüssel zu Beginn jeder Sitzung geändert wird, sollen alle Übertragungen zwischen Client und Server während dieser Sitzung geschützt sein. Oracle unterstützt Schlüssel von 40 Bits, 56 Bits oder 128 Bits Länge. (Je länger ein Schlüssel ist, desto schwieriger ist es, eine verschlüsselte Mitteilung zu knacken.)
Das Untersuchen von Paketen, die durch das Netzwerk transportiert werden, wird als **Packet Sniffing** bezeichnet. Dabei werden die Pakete nur analysiert, aber nicht verändert. Schlimm ist es wenn jemand ein Paket abfängt und dessen Inhalt verändert.

Um sicherzustellen, dass eine Nachricht auf keinen Fall verändert wird, generiert SSL einen kryptografisch sicheren Dienst für jede gesendete Nachricht.

6.3 Privilegien, Berechtigungen und Rollen

Wo wir auch sind oder was wir auch tun, es gibt im Leben immer gleichzeitig Rechte und Pflichten. Wenn man ein Datenbank-Administrator oder Sicherheitsmanager ist, vergibt man ebenfalls Rechte oder schränkt die Möglichkeiten von Benutzern in seiner Datenbank ein.

Durch **Rollen** lassen sich die Berechtigungen leichter verwalten und mit Hilfe von **Views** die Beschränkungen für Benutzer umsetzen.

6.3.1 Über Objekte und Berechtigungen

Es gibt viele verschiedene Arten von Objekten in einer Oracle-Datenbank. Die Tabelle 5 führt die gebräuchlichen Objekte mit einer kurzen Beschreibung auf.

Es gibt weiterhin verschiedene Arten von Berechtigungen in einer Oracle-Datenbank, die wir einem Benutzer erteilen können.
Wir können den Zugriff auf System- und Objektberechtigungen erteilen oder entziehen.
Objektberechtigungen ermöglichen es einem Benutzer, Daten in einer Tabelle oder View zu verändern oder ein Paket, eine Prozedur oder Funktion auszuführen.
Sie beinhalten Aktionen wie die in der Tabelle 6:

Object	Beschreibung
Table	Grundlegende Einheit der Datenspeicherung. Tabellendaten werden in Zeilen gespeichert, die Spalten enthalten. Jede Spalte hat einen Namen, einen Datentyp und eine Speichergröße.
Index	Eine optionale Datenbank-Struktur zum schnellen Finden einer Zeile in einer Tabelle. (Es gibt drei Arten von Indizes.)
View	Eine virtuelle Tabelle, die nicht physisch gespeichert ist, aber für den Benutzer wie eine echte Tabelle aussieht. Eine View kann Zeilen und Spalten aus einer oder mehreren Tabellen enthalten und als Sicherheits-Feature verwendet werden, da sie die Daten der zu Grunde liegenden Tabellen verbergen kann.
Sequence	Eine serielle Liste von eindeutigen Zahlen, die zusammen mit numerischen Spalten zum Erstellen eindeutiger Werte für Zeilen in einer oder mehreren Tabellen genutzt werden kann.
Snapshot/Materialized View	Dienen dem indirekten Zugriff auf Tabellendaten und Speichern das Ergebnis einer Abfrage. Materialisierte Views speichern Daten und nehmen Platz in der Datenbank ein. Snapshots sind materialisierte Views für die Datenreplikation mit entfernten Datenbanken. Ab Oracle 8*i* sind die Begriffe identisch.
Synonym	Ein Alias für eine Tabelle, View, Sequenz oder Programmeinheit, um den wirklichen Namen und Besitzer eines Objekts zu verbergen. Ein Synonym ist kein wirkliches Objekt, wird aber als direkte Referenz auf ein solches verwendet.
Cluster	Optionale Gruppen einer oder mehrerer Tabellen, die zusammen gepeichert werden, da sie häufig gemeinsam genutzt werden.

Tabelle 5: Gebräuchliche Datenbank-Objekte

Berechtigung	Aktion
Select	Informationen aus einer Tabelle oder View anzeigen
Insert	Neue Zeilen mit Informationen in eine Tabelle oder View eintragen
Update	Eine oder mehrere Spalten mit Informationen in einer Tabelle oder View verändern
Delete	Eine oder mehrere Zeilen mit Informationen aus einer Tabelle oder View löschen
Alter	Die Definition eines Objektes ändern
Execute	Eine Prozedur oder Funktion, auf die in einem Programm zugegriffen wird, kompilieren, ausführen oder nutzen
Read	Dateien in einem Verzeichnis lesen
Reference	Eine Abhängigkeit erstellen, die auf eine Tabelle verweist
Index	Einen Index auf eine Tabelle erstellen

Tabelle 6: Objektberechtigungen

6.3.2 Vergabe der Berechtigungen

Die Informationen dieses Abschnitts stammen größtenteils von [TN01]. Bisher haben wir uns mit einigen Arten von Privilegien in einer Oracle-Datenbank beschäftigt. Wenn wir nun tatsächlich die Berechtigungen an unsere Benutzer vergeben:

Wir starten mit **grant**, um Oracle mitzuteilen, dass wir einen Benutzer oder eine Rolle dazu ermächtigen wollen, eine oder mehrere System- oder Objektberechtigungen zu besitzen.

Es gibt zwei Formen des Befehls **grant** in Abhängigkeit davon, ob wir eine System- oder Objektberechtigung erteilen. (Der erste Teil des Befehls bleibt gleich.)

Wir müssen die zu erteilende Berechtigung angeben und den Benutzer oder die Rolle, dem oder der wir sie zuweisen möchten. Dazu wird folgendes Format benutzt:

- **grant <Berechtigung> to <Benutzer- oder Rollenname>;**

  oder bei Objektberechtigung:

- **grant <Berechtigung> on<Object> to <Benutzer- oder Rollenname>;**

Wir haben einem Benutzer eine Berechtigung erteilt, wollen ihn nun aber davon abhalten, diese weiter zu verwenden. Dann können wir den Befehl **revoke** einsetzen, um ihm die Berechtigung zu entziehen.

Wenn wir Objektberechtigungen erteilen, können wir dem Benutzer auch erlauben, diese Berechtigungen mittels „**with grant option**“ weiter zu geben.

Genauso ist dies bei Systemberechtigungen mit „**with admin option**“ möglich.

7 Oracle DBMS Teil 2

7.1 Sicherheitsoptionen bei Oracle

Die Informationen dieses Abschnitts stammen größtenteils von [TN01].

7.1.1 Virtual Private Databases

Virtual Private Database (abgekürzt VPD) stellt sicher, dass für jeden angemeldeten Benutzer welche Sicherheitseinstellungen geeignet sind. Oracle hat VPD eingeführt, um eine detaillierte Zugriffskontrolle verbunden mit einem sicheren Applikationskontext in Oracle8i bieten zu können. Mit VPD definieren wir Richtlinien, die Oracle für unsere Datenbankbenutzer umsetzen kann. Bei diesem Ansatz muss ein Unternehmen nur ein Mal eine Sicherheitsstruktur auf dem Datenserver aufbauen. Da die Sicherheitsrichtlinien mit den Daten statt mit der Applikation verknüpft werden, werden die Sicherheitsregeln immer umgesetzt, unabhängig davon, wie auf die Daten zugegriffen wird. Daher entsprechen die Daten, auf die ein Benutzer über eine Applikation Zugriff hat, denen, die er über SQL*Plus sieht. Die VPD basiert auf mehreren Mechanismen, um sicherzustellen, dass die Daten für jedes Unternehmen und jeden Kunden vertraulich behandelt werden. Um Datentrennung in einer Application Service Provider-Umgebung durchführen zu können, muss man zunächst sicherstellen, dass die Tabellen so entworfen wurden, dass der Datenzugriff entsprechend den Werten in einer oder mehreren Spalten eingeschränkt werden kann. Ein üblicher Einsatz ist der, ein Unternehmens-Schlüsselfeld in jede Tabelle aufzunehmen. Man kann Benutzer mit einem bestimmten Schlüssel verknüpfen und dann eine Richtlinie festlegen, nach der Benutzer nur auf Zeilen zugreifen können, die zu ihrem Kundenschlüssel passen. Jede Abfrage, die gestartet wird, enthält eine `where`-Klausel, die den Benutzer auf seine Kundendaten beschränkt. Man kann zum Beispiel die folgende Abfrage ausführen: `select * from EMPLOYEES;` Wenn die Tabelle EMPLOYEES eine Sicherheitsrichtlinie enthält, die festlegt, dass Mitarbeiter nur die Informationen der eigenen Firma einsehen dürfen, wird die Abfrage automatisch wie folgt umgeschrieben:

```
select * from EMPLOYEES
where CUSTOMER_ID = sys_context('HR_CONTEXT', 'CUST_ID');
```

Wobei die Systemfunktion `sys_context` den Wert für das Attribut `CUST_ID` in `HR_CONTEXT` zurückliefert.

7.1.2 Eine VPD erstellen

Analyse der Datenbankobjekte und ihrer Beziehungen Wenn man eine VPD erstellt, muss man zunächst die betroffenen Datenbankobjekte, ihre Beziehungen untereinander und die Schlüssel zusammenstellen, auf denen der Sicherheitsansatz basieren soll. Um das Ganze zu verdeutlichen, gehen wir ein Beispiel durch. Wir gehen davon aus, dass unsere Firma ein neuer ASP ist, der nur zwei Unternehmen unterstützt. Die Namen der Unternehmen lauten A und B. Jede von ihnen stellt ein Produkt her und

verkauft es. Die Kundennummern sind nur innerhalb eines Unternehmens eindeutig.

Der primäre Schlüssel, der alle Informationen eines Unternehmens miteinander verknüpft, besteht aus der Unternehmens-ID zusammen mit einer eindeutigen Kundennummer. Daher benötigen wir die Unternehmens-ID und die Kundennummer, um die Datensätze identifizieren zu können, auf die ein Mitarbeiter Zugriff hat.

Wir wollen die Tabelle `CUSTOMERS` so anlegen, dass wir sehen können, was in ihr gespeichert werden soll:

```
creat table CUSTOMERS(
 COMPANY_ID varchar2(2),
 CUST_ID number(9),
 CUST_NAME varchar2(20),
 CUST_ADDRESS1 varchar2(20),
 CUST_ADDRESS2 varchar2(20),
 EMPLOYEE_ID number(5) nt null,
 constraint CUSTOMER_SEC_CTX_PK primary key (COMPANY_ID, CUST_ID)
);
```

Und wir legen die Auftragstabelle `PRODUCT_ORDERS` wie folgt an:

```
create table PRODUCT_ORDERS(
 COMPANY_ID varchar2(2),
 CUST_ID number(9),
 ORDER_NO number(10),
 ORDER:DATE date,
 constraint CUSTOMER_SEC_CTX_FK foreign key (COMPANY_ID, CUST_ID)
);
```

Den Applikationskontext erstellen Ein Applikationskontext ist eine benannte Menge von Attributen und Werten, die man setzt und dann der aktuellen Benutzersitzung zuweisen kann. Oracle bietet einen Standardkontext namens `USERNV` an, der Systeminformationen über die aktuelle Sitzung hat, wie zum Beispiel den Benutzernamen, den Host und den Programmnamen. Wenn man andere Attribute für einen Benutzer definieren will, wie die Mitarbeiter- oder Kundennummer, kann man dazu einen Applikationskontext nutzen.

Nachdem wir in Oracle8i die neue Berechtigung Create Any Context erhalten haben, können wir den Applikationskontext anlegen. Wir geben einen eindeutigen Kontextnamen an und weisen diesem das Paket zu, das den Kontext implementiert. Kontextnamen müssen innerhalb der gesamten Datenbank eindeutig sein.

In unserem Beispiel wollen wir einen Kontext namens `CUSTOMER_SEC_CTX` anlegen, der zu unserem PL/SQL-Paket gehört, das im Schema `SECAP` liegt und den Namen `COMPANY_SEC` trägt. Die Syntax lautet:

```
create context CUSTOMER_SEC_CTX using SECAP.COMPANY_SEC;
```

Ein Paket erstellen, das den Kontext verwaltet Nachdem wir den Kontext angelegt haben, müssen wir als nächstes das Paket und die Funktionen anlegen, die den Kontext setzen. Das folgende Beispiel zeigt, wie wir die Kontextattribute `EMPLOYEE_ID` und `COMPANY_ID` setzen, indem wir auf den aktuellen Benutzernamen zurückgreifen, der sich im Standardkontext `USERNV` befindet. Die Funktion nutzt den Benutzernamen, um die notwendigen Attribute in der Tabelle `EMPLOYEE` nachzuschlagen.

```
create or replace package COMPANY_SEC is
   procedure GET_EMPLOYEE_ID;
end COMPANY_SEC;
create or replace package body COMPANY_SEC is
   procedure GET_EMPLOYEE_ID is
   EMPLOYEE_ID_VAR number;
   COMPANY_ID_VAR varchar2(2);
   begin
      select EMPLOYEE_ID, COMPANY_ID
         into EMPLOYEE_ID_VAR, COMPANY_ID_VAR from EMPLOYEES
          where
         EMPLOYEE_USERNAME = SYS_CONTEXT('USERNV','SESSION_USER');
      dbms_session.set_context('CUSTOMER_SEC_CTX', 'EMPLOYEE_ID',
                                                  EMPLOYEE_ID_VAR);
      dbms_session.set_sontext('CUSTOMER__SEC_STX', 'COMPANY_ID',
                                                   COMPANY_ID_VAR);
   end GET_EMPLOYEE_ID;
end COMPANY_SEC;
```

Oracle stellt die Funktion `SYS_CONTEXT` und den Standardkomntext `USERNV` bereit, so dass wir den Namen des Benutzers erhalten können, der diese Prozedur ausführt. Es gibt natürlich viele andere Werte, die über `SYS_CONTEXT` ermittelt werden können. Wir verzichten aber in dieser Ausarbeitung alle zu nennen.

Die Funktionen für die Sicherheitsrichtlinie erstellen Lassen wir uns überlegen, was passieren muss, um die detaillierte Zugriffskontrolle für eine Abfrage durchzuziehen, nachdem unsere Funktion eingerichtet wurde. Mit dem Begriff Abfrage meinen wir jede Form von Datenzugriff auf eine Tabelle.

Wir wissen, dass wir dafür sorgen müssen, dass jeder Mitarbeiter der Unternehmen nur seinen eigenen Kundeninformationen einsehen oder verändern darf. Daher wollen wir erreichen, dass jede gegebene Abfrage dynamisch um eine `where`-klausel erweitert wird, die die `COMPANY_ID` und `EMPLOYEE_ID` enthält.

Die Funktion, die die Sicherheitsrichtlinie für die Tabelle `PRODUCT_ORDERS` umsetzen soll, muss ein Prädikat zurückliefern, das einem Benutzer nur die Sicht auf die Bestellungen seiner eigenen Kunden erlaubt. Dazu fragen wir die Tabelle `CUSTOMERS` nach der aktuellen Mitarbeiternummer und Unternehmens-ID ab. Diese Attribute sind Teil des Applikationskontext des Benutzers und können damit über die Systemfunktion

SYS_CONTEXT ausgelesen werden. Eine mögliche Umsetzung der Sicherheitsrichtlinie-Funktion für die Tabelle PRODUCT_ORDERS ist im folgenden Code-Abschnitt zu sehen:

```
create or replace package body EMPLOYEE_SEC as
/* Schränkt SELECT-Befehle auf Basis der Kundennummer ein: */
  function EMPLOYEE_ID_SEC return varchar2
  is
    My_PREDICATE varchar2 (2000);
    begin
      MY_PREDICATE := 'EMPLOYEE_ID = SYS_CONTEXT(
        ' 'CUSTOMER_SEC_CTX' ', ' 'EMPLOYEE_ID' ')
                  and COMPANY_ID = SYS_CONTEXT(
        ' 'CUSTOMER_SEC_CTX' ', ' 'COMPANY_ID' ')';
    return MY_PREDICATE;
  end EMPLOYEE_ID_SEC;


funktion ORDERS_SEC return varchar2
  is
    MY_PREDICATE varchar2 (2000);
    begin
      MY_PREDICATE := 'CUST_ID in (select CUST_ID from CUSTOMERS
         where EMPLOYEE_ID = SYS_CONTEXT(
         ' 'CUSTOMER_SEC_CTX' ', ' 'EMPLOYEE_ID' ')
         and COMPANY_ID = SYS_CONTEXT(
         ' 'CUSTOMER_SEC_CTX' ', ' 'COMPANY_ID' '))';
    return MY_PREDICATE;
  end ORDERS_SEC;
end EMPLOYEE_SEC;
```

In diesem Code-Beispiel ermittelten wir die EMPLOYEE_ID und die COMPANY_ID aus dem Applicationskontext CUSTOMERS_SEC_CTX und erstellten ein Prädikat, das an die Abfrage der Tabelle CUSTOMERS angehängt werden muss. Um zu zeigen, wie das Ergebnis aussehen kann, nehmen wir an, dass die EMPLOYEE_ID den Wert 2435 und die COMPANY_ID den Wert A hat. Das zurückgelieferte Prädikat sieht dann so aus:

```
EMPLOYEE_ID = 2435 and COMPANY_ID = 'A';
```

7.2 Java Database Connectivity

Die Informationen dieses Abschnitts stammen größtenteils von [FC05].

7.2.1 Einführung

Java Database Connectivity (abgekürzt JDBC) ist ein plattformneutrales Interface von „Sun“ zwischen Java und Datenbanken. JDBC ermöglicht Java-Anwendungen

und Applets einen einfachen Zugriff auf relationale Datenbanken. Sie besteht aus einer Menge von Java-Klassen und Schnittstellen, die folgende Aufgaben unterstützen:

- Aufbau einer Verbindung zu einer Datenbank
- Verschicken der SQL-Anweisungen
- Verarbeiten der Ergebnisse

7.2.2 JDBC Architektur

JDBC enthält im wesentlichen zwei Schnittstellenmengen: die JDBC-API, die von Java-Applets und Anwendungen benutzt wird, und die JDBC Treiber API, die ein JDBC-Treiber implementieren muss.

Die JDBC-API Das Paket `java.sql` enthält die meisten und wichtigsten Klassen und Schnittstellen der JDBC-API, insbesondere:

- DriverManager, eine Klasse, die eine Liste von Treibern verwaltet und unter Auswahl eines geeigneten Treibers Verbindungen zu einer bestimmten Datenbank herstellt,
- Connection, eine Schnittstelle, deren Objekte Verbindungen zu einer Datenbank darstellen,
- Statement, eine Schnittstelle zur Ausführung von SQL-Anweisungen,
- ResultSet, eine Schnittstelle, die den Zugriff auf Ergebnisse von SQL-Anweisungen ermöglicht.

Die JDBC Driver-API Die Kommunikation mit der Datenbank findet über einen Treiber statt. Es gibt vier Arten von Treibern:

Typ-1 JDBC-ODBC bridge drivers: Bei dieser Variante wird für den Datenbankzugriff ein nativer ODBC-Treiber verwendet, der lokal beim Client installiert werden muss und über die `JDBC-ODBC-Bridge` von Sun angesteuert wird.

Typ-2 Native-API partly Java drivers: Hier wird der native ODBC-Treiber durch einen nativen herstellerabhängigen Treiber (zum Beispiel einen Oracle-Treiber) ersetzt. Auch dieser Treiber muss lokal beim Client installiert werden.

Typ-3 Net-protocol All-Java drivers: Diese Architektur ersetzt die lokale Installation eines Treibers beim Client durch eine serverseitige Middleware-Installation, welche den Datenbankzugriff realisiert. Das bietet den Vorteil, dass beim Client keinerlei Installationen mehr notwendig ist, da der universelle Treiber vom Server geladen werden kann und der „echte“ Treiber auf dem Server ausgeführt wird.

Typ-4 Native-protocol All-Java drivers: Bei dieser Möglichkeit werden Treiber verwendet, welche in reinem Java-Code programmiert sind. Die Anbindung von nativem Code entfällt. Diese Alternative wird als die modernste betrachtet, da sie durch die fehlende Einbindung von nativem Code auf der einen Seite die Plattformunabhängigkeit von Java unterstützt und auf der anderen Seite den Download der Treiber vom Webserver auf den Client ermöglicht.

7.2.3 Verbindung herstellen

JDBC-URLs Ein JDBC-Treiber braucht eine JDBC-URL, um eine Datenbank zu identifizieren und mit dieser eine Verbindung aufzubauen. URLs sind im Allgemeinen von der Form:

jdbc:*driver***:***databasename*

Registrieren der Treiber Bevor ein Treiber benutzt werden kann, sollte er mit JDBC-DriverManager registriert werden. Meistens wird es anhand der Methode `Class.forName()` gemacht.

Connection Eine Verbindung zu einer Datenbank wird durch ein Connection-Objekt repräsentiert. Dieses erhält man durch einen Aufruf einer der DriverManager-Methoden `getConnection()`. Abhängig davon, welche Voraussetzungen unsere Datenbank hat, muss die passende Methode mit benötigten Eingabeparameter benutzt werden. Die möglichen Methoden sind:

`getConnection(String url)`,
`getConnection(String url, Properties info)` und
`getConnection(String url, String user, String password)`

Um die vom Objekt benutzten Speicher und die Datenbank-Ressourcen freizugeben, muss die Verbindung anhand der Methode `close()` beendet werden, falls sie nicht mehr gebraucht wird. (Eigentlich ist hier die Freigabe der Datenbank-Ressourcen unser Hauptargument für das Beenden der Verbindung.)

7.2.4 Anweisungen und Ergebnisse

Statements Die Schnittstelle Statement stellt Methoden zum Versenden von SQL-Anweisungen an die Datenbank zur Verfügung. Um eine SQL-Anweisung auszuführen, braucht man ein `Statement`-Objekt. Dieses Statement-Objekt kann man anhand der Methode `createStatement()` von `Connection` bilden. Eine SQL-Anweisung wird dem Treiber als String übergeben. Man benutzt dazu eine der Methoden `executeQuery()`, `executeUpdate()` und `execute()`, um diese Anweisung auszuführen. Falls die auszuführende Anweisung eine Ausgabe liefert, braucht man auch ein `ResultSet`-Objekt.

SQL Data type	**java type**	**get-Method**
CHAR	String	getString()
VARCHAR	String	getString()
LONGVARCHAR	String	getString()
NUMERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	Boolean	getBoolean()
TINYINT	Integer (byte)	getByte()
SMALLINT	Integer (short)	getShort()
INTEGER	Integer (int)	getInt()
BIGINT	Long (long)	getLong()
REAL	Float (float)	getFloat()
FLOAT	Double (double)	getDouble()
DOUBLE	Double (double)	getDouble()
BINARY	byte[]	getBytes()
VARBINARY	byte[]	getBytes()
LONGVARBINARY	byte[]	getBytes()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimeStamp()
BLOB	java.sql.Blob	getBlob()
CLOB	java.sql.Clob	getClob()

Tabelle 7: Verwendung von get-Methoden

ResultSets Ein `ResultSet`-Objekt enthält alle Zeilen der Ergebnistabelle einer SQL-Anfrage und wird von einer `executeQuery`-Anweisung erzeugt. Es besitzt einen Cursor, der auf die aktuelle Zeile verweist und anfänglich vor der ersten Zeile positioniert ist. Mit der Methode `next()` wird der Cursor um eine Zeile weitergesetzt und liefert, wenn er sich schließlich hinter der letzten Zeile befindet, den Wert false zurück.

Um auf die Spaltenwerte der aktuellen Zeile zuzugreifen, benutzt man die `get`-Methoden. Es ist immer wichtig, die korrekte und passende `get`-Methode dafür zu verwenden. Eine korrekte Anwendung der `get`-Methoden ist in der *Tabelle 7* zu sehen.

Mehrfache Resultsets Es ist möglich, dass ein SQL-Statement mehrere ResultSets oder Updates ausgibt, zum Beispiel bei Transaktionen. In diesem Fall kann das letzte Beispiel anhand der Methode `getMoreResultSets()` so modifiziert werden, dass alle Ausgaben gerettet werden.

getMoreResultSets() schließt das existierte ResultSet und geht in das nächste ResultSet für das Statement.
Sie gibt `true` aus, wenn ein nächstes ResultSet vorhanden ist und falls nicht, gibt sie `false` aus. Es wird auch `false` ausgegeben, wenn das nächste Statement ein Update ist!

Prepared Statements Die Erweiterung `PreparedStatement` unterscheidet sich von `Statement` in folgenden zwei Punkten:

- `PreparedStatement`-Objekte enthalten eine bereits an die Datenbank gesendete, kompilierte SQL-Anweisung.
- Die in einem `PreparedStatement`-Objekt enthaltene Anweisung darf ein oder mehrere IN-Parameter besitzen.

Die gleiche SQL-Anweisung eines `PreparedStatement`-Objekts kann mehrfach ausgeführt werden, solange sie von der Datenbank geöffnet bleibt. Da aber nur einmal eine Ausführungsstrategie ermittelt werden muss, resultiert daraus eine Effizienzsteigerung bei der Verwendung von `PreparedStatement`-Methoden für häufig ausgeführte SQL-Anweisungen. Es ist zu beachten, dass bei Vorliegen einer Verbindung im Auto-Commit-Modus nach Abschluss einer Anweisung Commit bzw. Rollback durchgeführt wird. Eine Effizienzsteigerung kann hier nur dann erreicht werden, wenn die Anweisung auch nach Commit oder Rollback nicht durch die Datenbank oder den Treiber geschlossen wird.

Ein `PreparedStatemet`-Objekt wird durch Aufruf der Methode `prepareStatement()` eines geeigneten `Connection`-Objekt erzeugt.

Callable Statements Die Schnittstelle `CallableStatement`, eine Erweiterung von `PreparedStatement`, bietet Methoden zur Ausführung von bei der Datenbank gespeicherte DB-Prozeduren. Der Aufruf der DB-Prozedur wird in einer Escape-Syntax geschrieben, die IN-, OUT-, INOUT- und Ergebnis-Parameter besitzen kann. Der Aufruf für eine DB-Prozedur ohne Ergebnisparameter hat die Form

`{ call procedure_name[ (?,?,...) ] }`,

und für eine DB-Prozedur mit Ergebnisparameter sieht es so aus:

`{ ? = call procedure_name[ (?,?,...) ] }`.

Ein `CallableStatement`-Objekt wird durch die Methode `prepareCall()` von `Connection` erzeugt. Es ist zu beachten, dass die OUT-Parameter durch die Methode `registerOutParameter()` registriert werden sollen.

Ein Beispiel Wir schauen uns jetzt ein einfaches Beispiel an, um was bisher gesprochen worden ist zu veranschaulichen.

```
import java.sql.*;

public class JDBCSample{

  public static void main(java.lang.String[] args) {
    try {
      Class.forName("Sun.jdbc.odbc.JcbcOdbcDriver");
    }
```

```
    catch{ (ClassNotFoundException e) {
      System.out.println("Unable to load Driver Class");
      return;
    }

    try {
      Connection con = DriverManager.getConnection(
        "jdbc:odbc:companydb", "", "");
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(
        "SELECT FIRST_NAME FROM EMPLOYEES");

      while(rs.next()) {
        System.out.println(rs.getString("FIRST_NAME"));
      }

      rs.close();
      stmt.close();
      con.close();
    }
    catch (SQLException se) {
      System.out.println("SQL Exception: " + se.getMessage());
      se.printStackTrace(System.out);
    }
  }
}
```

7.2.5 MetaData

Nicht immer verfügt man über genügend Wissen über die zugrundeliegende Datenbank (z.B. bei Programmierung von Entwicklungsumgebungen). Die Schnittstelle `DatabaseMetaData` stellt sowohl Informationen über die in der Datenbank gespeicherten Daten und Prozeduren als auch über den verwendeten Treiber und das DBMS zur Verfügung, wie z.B.: `getTables()`, `getProcedures()`, `getURL()`, `getTypeInfo()`, `getDriverName()`, `supportsOuterJoins()`, `supportsPositionedUpdate()`, `supportsTransactions()`.

Einige der Methoden besitzen String-Parameter, die als Suchmuster dienen. Innerhalb eines Such-Strings ist ein „_“ ein Platzhalter für ein einzelnes Zeichens und ein „%“ ein Platzhalter für einen Substring von 0 oder mehr Zeichen. Für Katalog und Schema-Werte entspricht ein leerer String dem Wert „unbenannt“, und bei einem `null` kann das Suchkrititerium ignoriert werden.

Das Interface `ResultSetMetaData` stellt Informationen über ein bestimmtes ResultSet-Objekt.

7.2.6 Transaktionen

Standardmäßig ist eine Verbindung im Auto-Commit-Modus, d.h. Commit bzw. Rollback wird nach Abschluß jeder Anweisung automatisch durchgeführt. Dabei gilt die zu einem `Statement`-Objekt gehörende Anweisung als abgeschlossen, wenn sie ausgeführt worden ist und alle ihre Ergebnisse zurückgegeben worden sind.

Manchmal möchte man Datenbankoperationen nur zusammen ausführen, z.B. weil die Ausführung einer Operation von der einer anderen abhängig ist. Durch Aufruf der `Connection`-Methode `setAutoCommit(false)` und expliziter Verwendung von `commit()` bzw. `rollback()` kann man den Standardmodus verlassen und Operationen zu Transaktionen gruppieren. Der Aufruf von `commit()` macht die Datenbankänderungen dauerhaft und gibt die von der Transaktion gehaltenen Sperren frei; durch den Aufruf von `rollback()` werden die seit dem letzten `Commit()` durchgeführten Änderungen zurückgesetzt.

Zur Synchronisation von Transaktionen bietet die `Connection`-Interface fünf verschiedene „Transaction Isolation Levels“ an:
`TRANSACTION_NONE`,
`TRANSACTION_READ_UNCOMMITED`,
`TRANSACTION_READ_COMMITED`,
`TRANSACTION_REPEATABLE_READ`,
`TRANSACTION_SERIALIZABLE`

Auf der niedrigsten Stufe werden Transaktionen nicht unterstützt, auf der höchsten dürfen an Daten, die von einer noch nicht beendeten Transaktion gelesen worden sind, andere Transaktionen keinerlei Änderungen vornehmen.

8 Theorembeweiser Teil 1

8.1 Einleitung

Die Anfänge im Bereich automatisches Beweisen lassen sich in die Mitte der 50er Jahre zurückverfolgen. Damals gab es erste Versuche, mathematische Resultate durch Computer programme zu berechnen. Daraus ergeben sich die Fragen, was automatisches Beweisen eigentlich bedeutet oder leisten soll und warum man sich mit Inferenzsystemen und maschinellem Beweisen beschäftigt. Eine einfache Definition ist die Aussage, dass automatisches Beweisen bedeutet, mathematische Beweise mittels Computerprogrammen zu führen. Etwas ausführlicher heisst das, mathematische Wahrheiten formal zu modellieren und mit einer endlichen Anzahl von Axiomen und Regeln formal zu beweisen. Im Einzelnen werden dazu zwei Aktionen unternommen: Zuerst wird ein Problem modelliert und anschließend ein Beweis abgeleitet.
Bei der Modellierung wird als erstes ein Problem aus der natürlichen Sprache über Formalisierung nach einer festgelegten Syntax zu Zielformeln transformiert. Anschließend können den formalen Formeln Wahrheitswerte zugeordnet werden, wenn man eine Interpretation und gegebenenfalls eine Variablenbelegung kennt. Dadurch wird semantische Korrektheit hergestellt. Syntax und Semantik zusammen bilden eine Logik. Für das automatische Beweisen wird für gewöhnlich die Prädikatenlogik 1.Stufe verwendet. Allerdings können auch schwächere oder stärkere Logiken verwendet werden, die für die Umsetzung aber gegebenenfalls nicht mächtig genug sind oder zu kompliziert werden können.
Ein Theorembeweiser ist also ein autonomes oder interaktives Programm, dass die Aufgabe hat, für vorgegebene Sätze mittels eines Beweiskalküls Beweise zu finden. Mit Hilfe solcher Programme können anspruchvolle Probleme der Informatik gelöst werden. Auf den folgenden Gebieten werden Theorembeweiser eingesetzt: in der mechanisierten Mathematik zur Ableitung mathematischer Theoreme; viele KI-Systeme bauen auf einer Form der Deduktion auf bzw. enthalten einen Modul für maschinelle Deduktion; logisches Programmieren, deduktive Datenbanken, deduktive Programm-Synthese; in der Modellierung und Analyse von Systemen und System-Komponenten z.B. zum Nachweis kritischer Eigenschaften.
An dieser Stelle wollen wir ausführlicher den Begriff Deduktionssysteme erläutern, da Theorembeweiser unmittelbar solche darstellen. Nach [**?**] ist ein Deduktionssystem ein programmierbares Verfahren zur Erkennung von Folgerungsbeziehungen zwischen Aussagen. Die Folgerungsbeziehung sei durch ein typisches beispiel illustriert, dessen generelle Form bereits auf Aristoteles zurückgeht:

Aussage 1:	Alle Menschen sind sterblich
Aussage 2:	Sokrates ist ein Mensch
Aussage 3:	Sokrates ist sterblich

In diesem Fall folgt Aussage 3 aus den ersten beisen Aussagen. Sobald man die ersten beiden als wahr annimmt, muss man zwangsläufig auch die dritte als wahr akzeptieren.
Es gibt verschiedene Varianten, die genaue Aufgabe eines Deduktionssystems zu spe-

zifizieren. Sie kann zum Beispiel darin bestehen, zu entscheiden, ob eine gegebene Aussage aus anderen gegebenen folgt. Oder das Deduktionssystem soll Folgerungsbeziehungen zwischen gegebenen Aussagen nachweisen, wenn diese Beziehungen tatsächlich bestehen. Eine weiter Möglichkeit wäre, dass das Deduktionssystem von gegebenen Aussagen ausgehen und daraus neue Aussagen erzeugen soll, die aus den gegebenen folgen.
Die Mechanismen, die einen Rechner zur Lösung solcher Aufgaben befähigen, sind im Prinzip sehr ähnlich zu denen, die ihm zum Rechnen mit Zahlen befähigen. Die Aussagen müssen als Datenobjekte repräsentiert und die Schlussfolgerungen als Operationen auf diesen Datenobjekten programmiert werden.
Ein komplettes Deduktionssystem baut sich im Wesentlichen aus vier Schichten auf. Die unterste Schicht wird durch eine *Logik* gebildet, die mit der Festlegung der Syntax einer formalen Sprache und deren Semantik die zulässige Struktur und die Bedeutung von Aussagen vorgibt. Aussagen entsprechen Formeln der Logik. Die gewählte Logik bestimmt ganz konkret, welche Arten von Aussagen erlaubt und welche verboten sind. Beispielsweise kann sie festlegen, dass eine Quantifizierung „Für alle Zahlen gilt...“ erlaubt, eine Quantifizierung „Für alle stetigen Funktionen gilt...“ dagegen verboten sein soll. Die Definition einer Bedeutung für die Formeln liefert in den Logiken, die für Deduktionssysteme interessant sind, darüberhinaus eine Beziehung „aus A folgt B“ zwischen Aussagen A und B. Damit ist ein semantischer *Folgerungsbegriff* etabliert und formal präzise definiert. Diese Definition hilft jedoch zunächst in keiner Weise, für gegebene A und B algorithmisch zu bestimmen, ob B wirklich aus A folgt.
Dies ist Gegenstand des Kalküls, der zweiten Schicht eines Deduktionssystems. Ein Kalkül definiert syntaktische *Ableitungen* als Operationen auf den Formeln. Damit kann aus einer Formel A durch reine Symbolmanipulation eine Formel B gewonnen werden, wobei die Bedeutung der in A und B vorkommenden Symbole überhaupt keine Rolle spielt. Im obigen Beispiel sollte man also nicht wissen müssen, wer oder was Sokrates eigentlich ist, um die Ableitung durchzuführen. Ein syntaktisch aus A abgeleitetes B soll aber trotzdem semantisch folgen und umgekehrt. Ein korrekter Kalkül stellt daher nur solche Ableitungsoperationen zur Verfügung, die garantieren, dass alles syntaktisch Ableitbare auch semantisch folgt. Wenn umgekehrt alles, was semantisch folgt, auch syntaktisch ableitbar ist, ist der Kalkül vollständig. Nach dem berühmten Unvollständigkeitssatz von Kurt Gödel sind vollständige Kalküle ab einer gewissen Ausdrücksstärke der Logiken jedoch nicht möglich.
Die dritte Schicht eines Deduktionssystems, die *Repräsentation*, bestimmt die Darstellung von Formeln oder Formelmengen, von deren Beziehungen zueinander, sowie von den jeweiligen Zuständen der Ableitungsketten. Auf dieser Ebene werden oft zusätzliche Operationen eingeführt, etwa das Löschen von redundanten Aussagen, so dass im Kalkül noch mögliche Ableitungen nun nicht mehr möglich - und hoffentlich auch nicht mehr nötig - sind. Eine gute Repräsentationsschicht trägt ganz entscheidend zur Effizienz eines Deduktionssystems bei.
Die letzte Schicht eines Deduktionssystems, die *Steuerung*, enthält schließlich die Strategien und Heuristiken, mit denen unter den möglichen Ableitungsschritten die jeweils sinnvollen ausgewählt werden. Hier steckt die eigentliche Intelligenz des Sytems.

1) Steuerung	spezielle Strategien und Heuristiken
2) Repräsentation	Tableau, Matrix, Klauselgraph
3) Kalkül	Gentzenkalküle, Resolution, Theorieresolution
4) Logik	Aussagenlogik, Prädikatenlogik höherer Stufe

8.2 Logische Grundlagen: Aussagenlogik

In diesem Abschnitt der Ausarbeitung wollen wir die Grundlagen der klassischen Aussagenlogik kurz zusammenfassen.

8.2.1 Syntax

Eine aussagenlogische Signatur Σ ist eine Menge von Bezeichnern, genannt *Aussagenvariablen.*
Für eine aussagenlogische Signatur Σ wird die Menge Formel(Σ) der aussagenlogischen Formeln wie folgt gebildet:

1. Eine *atomare Formel* ist eine aussagenlogische Formel, die nur aus einer Aussagenvariablen besteht.
2. Falls A und B aussagenlogische Formeln sind, dann sind auch die folgenden Konstrukte aussagenlogische Formeln, wobei die darin auftretenden Operationssymbole $\wedge$, $\neg$ etc. *Junktoren* heißen:

$(\neg A)$	Negation	nicht A
$(A \wedge B)$	Konjunktion	A und B
$(A \vee B)$	Disjunktion	A oder B
$(A \Rightarrow B)$	Implikation	wenn A, dann B
$(A \Leftrightarrow B)$	Koimplikation, Äquivalenz	A genau dann, wenn B

Um Klammern in Formeln einzusparen, vereinbaren wir die folgenden Bindungsprioritäten:

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$$

8.2.2 Semantik

Sei Σ eine aussagenlogische Signatur. Dann ist eine Abbildung $I : \Sigma \rightarrow BOOL$ eine (aussagenlogische) *Interpretation* (oder *Belegung*) für Σ. Int(Σ) ist eine Menge aller Σ-Interpretationen.

Das heisst insbesondere, dass eine Aussagenvariable durch eine Belegung beliebig interpretiert werden kann. Die Semantik der komplexeren aussagenlogischen Formeln ergibt sich dann aus der klassisch-logischen Bedeutung der Junktoren.
Sei I eine aussagenlogische Interpretation für Σ. Für eine aussagenlogische Formel A ist ihr *Wahrheitswert* $[\![A]\!]_I : Formel(\Sigma) \rightarrow BOOL$ für die $[\![A]\!]_I = I(A)$ gilt, falls A eine atomare Formel ist.

8.2.3 Äquivalenzen und Normalformen

Die Menge der aussagenlogischen Formeln ist reich an Redundanzen. So werden z.B. die Formeln A und $A \vee A$ sicherlich von genau den gleichen Interpretationen erfüllt. In vielen Situationen werden aber sog. Normalformen benötigt, die eine gewisse Standarddarstellung für Formeln sind. Als Beispiele für Normalformen können KNF und DNF genannt werden.
Zwei Formeln F und G sind *(semantish) äquivalent*, geschrieben $F \equiv G$, falls für alle Interpretationen I gilt

$$[\![F]\!]_I = [\![G]\!]_I$$

Da Formeln beliebig verschachtelt sein können, wird in Inferenzsystemen oft zunächst versucht, diese Formeln in semantisch äquivalente, technisch aber einfacher zu handhabende Formeln zu transformieren. Der folgende Satz liefert die Basis dafür.

Theorem (Semantische Ersetzbarkeit) Seien F und G äquivalente Formeln. Sei H eine Formel mit mindestens einem Vorkommen der Teilformel F. Dann ist H äquivalent zu einer Formel $H1$, die aus H dadurch gewonnen ist, dass (irgend)ein Vorkommen von F durch G ersetzt worden ist.
Das Theorem besagt, dass man durch Austausch semantisch äquivalenter Teilformeln wieder eine zur Ausgangsformel semantisch äquivalente Formel erhält.

8.2.4 Wahrheitstafeln in der Aussagenlogik

Für die Aussagenlogik stehen verschiedene Inferenzverfahren zur Verfügung. Die bekannteste Methode basiert auf den Wahrheitstafeln.
Das Verfahren der Wahrheitstafeln bildet ein einfaches Entscheidungsverfahren, eine aussagenlogische Formel auf Allgemeingültigkeit zu überprüfen. Allerdings wächst der Aufwand exponentiell mit der Anzahl der in einer Formel auftretenden Variablen: Für eine Formel mit n atomaren Formeln müssen 2^n Zeilen der Wahrheitstafel berechnet werden.

8.2.5 Äquivalenzen für die Aussagenlogik

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)

Mit angegebenen Transformationen lässt sich jede aussagenlogische Formel semantisch äquivalent transformieren, so dass darin als Junktoren nur noch Konjunktion, Disjunktion und Negation auftreten, wobei die Negation dabei nur unmittelbar vor Atomen auftritt.

8.2.6 Entscheidbarkeitsresultate

Mit den Begriffen Korrektheit und Vollständigkeit haben wir die wichtigste Charakterisierung von Kalkülen festgelegt. Eine ganz andere Frage ist, ob es zu einer gegebenen Logik überhaupt Kalküle mit diesen Eigenschaften gibt. Dies hängt natürlich von der Art der Logik ab. Ob es etwa einen Algorithmus gibt, der die Frage nach der Erfüllbarkeit oder Gültigkeit von Formeln einer Logik beantwortet, hängt davon ab, ob diese Fragestellung für die gegebene Logik überhaupt entscheidbar ist.
Für beliebige Menge M gilt:

M	ist entscheidbar	gdw.	es gibt einen Algorithmus, der für jedes x aus der Menge M 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. (Insbesondere muss der Algorithmus für ein x, dass nicht aus M ist, nicht unbedingt terminieren!)

Dabei bedeutet Algorithmus z.B. Turing-Maschine, Markov-Algorithmus etc.
Die Ausssagenlogik (also der Spezialfall von PL1, in der keine Funktionssymbole, keine Quantoren und keine Individuenvariablen und damit nur null-stellige Prädikatensymbole auftreten) ist entscheidbar, allerdings NP-Vollständig (s.SAT). D.h. es gibt

Algorithmen, die für jede aussagenlogische Formel entscheiden, ob sie allgemeingültig, erfüllbar, falsifizierbar oder unerfüllbar ist. Beispielsweise lassen sich diese Fragen mit der Methode der Wahrheitstafeln beantworten. Alle weiteren Informationen können in [CB03] nachgelesen werden.

8.3 Tableaus als Beweisverfahren für Aussagenlogik

Das erste Verfahren, bei dem über Formelmengen hinaus komplexere Strukturen zur Repräsentation des Suchzustandes eingesetzt werden, ist das Tableauverfahren fur Gentzen-Kalküle. Gentzen-Kalküle sind positive deduktive Kalküle, d.h. ausgehend von elementaren logischen Tautologien werden immer komplexere Formeln hergeleitet, die per Konstruktion ebenfalls Tautologien sind. Man kann daraus einen Testkalkül konstruieren, indem man die Gentzenregeln einfach umdreht und die Behauptung soweit in ihre Teilformeln zerlegt, bis sich über deren logischen Status eine Aussage machen lässt. Nicht ganz unproblematisch sind Formeln wie $F \vee G$, die Alternativen und damit Indeterminismen ermöglichen. Sie erzwingen, die Zerlegung baumförmig zu organisieren, so das in jedem Zweig der jeweilige Fall einzeln verfolgt werden kann.

Nach einigen Ansätzen von Jaakko Hintikka und Evert Beth hat schließlich Raymond Smullyan mit seinen analytischen Tableaus diese Idee zu einem negativen Testkalkül ausgearbeitet, der heute weit verbreitet ist. Alle Informationen dazu sind dem [BH01] zu entnehmen. Das Verfahren baut ausgehend von der Negation der als allgemeingültig nachzuweisenden Formel durch Anwendung von Dekompositionsregeln auf die Formeln einen Baum auf. Die Dekomposition endet, wenn die Formel in ihre atomaren Bestandteile zerlegt ist und entweder jeder einzelne Zweig als widersprüchlich erkannt wurde, oder zumindest ein Zweig eine konsistente Belegung der atomaren Formeln mit Wahrheitswerten zulässt, aus dem sich dann ein Modell ergibt. Im ersten Fall ist die negierte Formel widersprüchlich, und damit die unnegierte Formel allgemeingültig; im zweiten Fall hat man ein Gegenbeispiel gefunden. Bei einer erfüllbaren Wurzel terminiert die Zerlegung nicht notwendigerweise, sonst ergäbe sich ein Entscheidungsverfahren für PL1. Die Dekompositionsregeln garantieren, das alle Fallunterscheidungen systematisch untersucht werden und kein Fall vergessen werden kann.
Die Dekompositionsregeln sind:

$$\frac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{X \wedge Y}{\begin{array}{c}X\\ Y\end{array}} \qquad \frac{\neg(X \vee Y)}{\begin{array}{c}\neg X\\ \neg Y\end{array}} \qquad \frac{\neg(X \Rightarrow Y)}{\begin{array}{c}X\\ \neg Y\end{array}}$$

$$\frac{\beta}{\beta_1 | \beta_2} \qquad \frac{X \vee Y}{X | Y} \qquad \frac{\neg(X \wedge Y)}{\neg X | \neg Y} \qquad \frac{X \Rightarrow Y}{\neg X | Y}$$

Die Diagramme sind folgendermaßen zu lesen: Um die Formel über dem Strich zu widerlegen, widerlege die unter dem Strich stehenden. Stehen zwei durch einen senkrechten Strich getrennte Formeln nebeneinander, müssen beide widerlegt werden. Von untereinanderstehenden Formeln reicht die Widerlegung einer einzigen. Als Anweisung zur Konstruktion des Tableaus lesen sie sich dementsprechend: Wenn im gerade

bearbeiteten Zweig des Tableaus eine Formel vorkommt, die einem Typ entspricht, wie er über dem Strich steht, so erweitere den Zweig, indem die unter dem Strich untereinanderstehenden Formeln als neue Blätter angehängt werden und die nebeinanderstehenden Formeln zwei neue Zweige eröffnen. Ein Zweig heißt *beendet*, wenn keine Regel mehr anwendbar ist (*offen*), oder wenn in ihm eine Formel und ihre Negation vorkommen (*geschlossen*). Das Verfahren terminiert, wenn alle Zweige beendet sind.
Beispiel-Beweis für die Formel: $(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))$. Die Formel ist also als allgemeingültig nachzuweisen. Ihre Negation bildet somit die Wurzel des Tableaus. Die Struktur der ersten vier Zeilen ergibt sich mit den Regeln α aus jeder Formel des Typs $\neg[Voraussetzungen \Rightarrow Behauptung]$.

(1)	$\neg[(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))]$	
(2)	$(P \Rightarrow (Q \Rightarrow R))$	(α aus 1)
(3)	$\neg[(P \Rightarrow Q) \Rightarrow (P \Rightarrow R)]$	(α aus 1)
(4)	$(P \Rightarrow Q)$	(α aus 3)
(5)	$\neg(P \Rightarrow R)$	(α aus 3)
(6)	P	(α aus 5)
(7)	$\neg R$	(α aus 5)
(8)	$\neg P$ | (9) $(Q \Rightarrow R)$	(β aus 2)
(10)	$\neg Q$ | (11) R	(β aus 9)
(12)	$\neg P$ | (13) Q	(β aus 4)

Da alle Zweige geschlossen sind, ist die negierte Formel an der Wurzel unerfüllbar, die unnegierte also allgemeingültig.
Ein auf Tableaus basierender automatischer Beweiser mit starken Heuristiken zur Auswahl der Dekompositionsregeln und zum Erkennen von Sackgassen wurde von F. Oppacher und E. Suen an der Carleton University in Ottawa entwickelt. Viele der Indeterminismen beim Tableauverfahren wurden inzwischen ausgemerzt. Z.B. wurde Unifikation eingeführt und neue Abarbeitungsstrategien entwickelt. Ein Beweiser, der viele dieser Ideen integriert hat, ist der an der TU München entwickelte SETHEO Beweiser. In der konkreten technischen Realisierung ist in solchen Systemen dann kaum noch auszumachen, ob es sich um einen Tableaubeweiser, einen Konnektionsbeweiser oder einen Modell- Eliminationsbeweiser handelt.

8.4 Resolution

Resolution ist ein Beweisverfahren mit der Resolventenregel als (einziger) Schlussregel. Es ist ein Widerlegungsverfahren, d.h. ein Verfahren für Beweise durch Widerspruch.

Theorem: Eine Formel Y folgt semantisch aus einer Formelmenge $\{X_1, \ldots, X_n\}$ genau dann wenn

$$X_1 \wedge \ldots \wedge X_n \Rightarrow Y$$

gültig ist.

Eine Formel X ist (allgemein)gültig genau dann, wenn ihre Negation $\neg X$ unerfüllbar, also widersprüchlich ist.
Beweis durch Widerspruch: Um zu zeigen, dass aus einer Formel X die Formel Y folgt, d.h. dass $X \Rightarrow Y$ gültig ist, weist man nach, dass die Negation $\neg(X \Rightarrow Y)$, also $X \wedge \neg Y$, widersprüchlich ist.

Resolutionssatz: Eine Klauselmenge ist widersprüchlich genau dann, wenn sich aus ihr mit der Resolventenregel die leere Klausel ableiten läßt.

Als *Klausel* bezeichnet man eine Menge atomarer und negierter atomarer Formeln.

Resolventenregel:

$$\frac{\{A,L_1,...,L_m\}\{\neg A,K_1,...K_n\}}{\{L_1,...,L_m,K_1,...,K_n\}}$$

Die Schlussfolgerung der Regel heisst *Resolvente* der Prämissen. Die Resolventenregel kann als Verallgemeinerung von *Modus ponens* aufgefaßt werden:

$$\frac{\{A\},\{\neg A,B\}}{B}$$

Um mit Resolution zu zeigen, dass eine Formel Y aus einer Menge von Formeln $X_1, \ldots, X_n$ folgt, geht man folgendermaßen vor:
Z.z: $\neg(X_1 \wedge \ldots \wedge X_n \Rightarrow Y) \Leftrightarrow \neg(\neg(X_1 \wedge \ldots \wedge X_n) \vee Y) \Leftrightarrow (X_1 \wedge \ldots \wedge X_n) \wedge \neg Y$
Die Formel $X_1 \wedge, \ldots, \wedge X_n \wedge \neg Y$ wird zunächst in KNF gebracht, so dass sie als Menge von Klauseln dargestellt werden kann. Auf geeignete Paare von Klauseln aus der Klauselmenge wird die Resolventenregel angewendet und anschließend wird die Resolvente zu der Klauselmenge hinzugefügt. Wenn die leere Klausel abgeleitet wird, ist der Widerspruch gezeigt. Andernfalls bricht das Verfahren ab, wenn keine neuen Klauseln mehr mit der Resolventenregel abgeleitet werden können.

8.5 Implementierungen

8.5.1 SPASS

SPASS ist ein Projekt des Max-Planck-Instituts für Informatik in Saarbrücken und ist ein automatischer Theorembeweiser für Prädikatenlogik erster Stufe mit Gleichheit. Das generelle Problem, das SPASS löst, ist das Erfüllbarkeitsproblem der Prädikatenlogik. Da dieses im Allgemeinen unentscheidbar ist, bedeutet „rechnen“ hier suchen nach einer Lösung. Besondere Anwendung findet SPASS in Forschung und Industrie bei der Analyse von Sicherheitsprotokollen der Kommunikationstechnik. SPASS ist

zur Zeit der einzige Theorembeweiser, der in der Lage ist, Sicherheitsprotokolle zu untersuchen.
Beispiel: Der Startpunkt ist ein Problem, was gelöst werden soll. Es sind zwei Aussagen gegeben:
1. Sokrates ist ein Mensch.
2. Alle Menschen sind sterblich.

Wir wollen schließen
1. Sokrates ist sterblich.

Der erste Schritt ist die Formalisierung des Problems in Prädikatenlogik. Die Formeln sind:
1. *Human(sokrates)*
2. $\forall x : Human(x) \Rightarrow Mortal(x)$
3. *Mortal(sokrates)*
wo syntaktisch *Human* und *Mortal* Prädikate, während *sokrates* eine Konstante ist. Im nächsten Schritt folgt die Konstruktion der Eingabe für SPASS.

```
begin_problem(Sokrates1).
list_of_descriptions.
name({*Sokrates*}).
author({*Christoph Weidenbach*}).
status(unsatisfiable).
description({* Sokrates is mortal and since all humans are mortal,
he is mortal too. *}).
end_of_list.
list_of_symbols.
functions[(sokrates,0)].
predicates[(Human,1),(Mortal,1)].
end_of_list.
list_of_formulae(axioms).
formula(Human(sokrates),1).
formula(forall([x],implies(Human(x),Mortal(x))),2).
end_of_list.
list_of_formulae(conjectures).
formula(Mortal(sokrates),3).
end_of_list.
end_problem.
```

Die Eingabe für SPASS besteht aus drei Teilen, der Beschreibungsteil startet mit `list_of_descriptions.`, der Teil, der die Sigantur deklariert, wird mit `list_of_symbols.`gestartet, der Teil, wo alle Axiome angegeben sind, startet mit `list_of_formulae(axioms).`, `list_of_formulae(conjectures).`beschreibt den letzen Teil, der alle Vermutungen repräseniert. Die Formeln befinden sich in der Liste `formula` und sind immer in Präfixnotation geschrieben. Dann versucht SPASS zu beweisen, ob die Konjunktion aller Axiome die Disjunktion aller Vermutungen impliziert.
SPASS liest die Eingabe und transformiert die Formeln in eine Klauselmenge, die

Voraussetzungen plus negierte Behauptung enthält, denn SPASS basiert auf der Widerlegung. So sehen die Eingabe-Klauseln aus

```
1[0:Inp] || -> Human(sokrates)*.
2[0:Inp] || Mortal(sokrates)* -> .
3[0:Inp] || Human(U) -> Mortal(U)*.\\
```

SPASS analysiert das Problem und findet heraus, dass

```
This is a monadic Horn problem without equality.
This is a problem that has, if any, a finite domain model.
There are no function symbols.
This is a problem that contains sort information.
The conjecture is ground.
The following monadic predicates have finite extensions: Human.
Axiom clauses: 2 Conjecture clauses: 1
```

SPASS benutzt folgende Einstellungen basierend auf der Information oben

```
Inferences: IEmS ISoR IORe
Reductions: RFClR RBClR RObv RUnC RTaut RSST RSSi RFSub RBSub RCon
Extras : Input Saturation, Always Selection, No Splitting, Full Reduction
Precedence: Mortal > Human > sokrates
Ordering : KBO
```

Die Ausgabe stellt folgende Informationen zur Verfügung

```
Here is a proof with depth 1, length 5 :
1[0:Inp] || -> Human(sokrates)*.
2[0:Inp] || Mortal(sokrates)* -> .
3[0:Inp]Human(U) || -> Mortal(U)*.
4[0:Res:3.1,2.0]Human(sokrates) || -> .
5[0:ClR:4.0,1.0] || -> .
Formulae used in the proof : 1 3 2
```

Klausel 4 ist das Resultat des Resolutionsschrittes zwischen Klauseln 3 und 2. Klausel 5 wurde via Klauselreduktion aus den Klauseln 4 und 1 gewonnen. Die Nummern, die hinter `Formulae used in the proof` stehen, geben Auskunft darüber, welche Formeln aus der Eingabe für den Beweis benutzt wurden.

8.5.2 OTTER

Der Theorembeweiser `otter` ist ein Werkzeug, das mit Hilfe des Resolutionskalküls die Unerfüllbarkeit einer aussagenlogischen Formel überprüft. Man kann `otter` auch für die Prädikatenlogik einsetzen. Wie im Fall der Aussagenlogik kann man auch hier die Unerfüllbarkeit einer Formel beweisen. Allerdings kann es im Gegensatz zur Aussagenlogik vorkommen, dass `otter` nicht terminiert, falls die Formel erfüllbar ist. Wenn

die Formel unerfüllbar ist, dann sollte `otter` immer terminieren (das kann jedoch sehr lange dauern und auch der zur Verfügung stehende Speicher kann unter Umständen nicht ausreichen). Das Werkzeug `otter` wurde an den Argonne National Laboratory in Illinois, USA, entwickelt.
Das Tool `otter` liest die zu analysierende Formel aus einer Textdatei aus. Die Formel muss sich dabei in konjunktiver Normalform befinden. Zu beachten ist dabei, dass jede Klausel in einer eigenen Zeile steht und mit einem Punkt abgeschlossen wird. Die Disjunktion wird durch | dargestellt, die Negation durch -. Atomare Formeln werden klein geschrieben, Kommentare werden mit % eingeleitet. Zu Beginn muss man einige Optionen definieren.

```
set(prologstylevariables).
set(binaryres).
clear(unitdeletion).

list(sos).                    % Beginn der Klauselliste

a| b                          % (A ∨ B)
-a| b                         % (¬A ∨ B)
-b| -r| -a                    % (¬B ∨ ¬R ∨ ¬A)
r.                            % R
a| -b                         % (A ∨ ¬B)

end of list.                  % Ende der Klauselliste
```

Dies enspricht der Formel
$(A \vee B) \wedge (\neg A \vee B) \wedge (\neg B \vee \neg R \vee \neg A) \wedge R \wedge (A \vee \neg B)$
Die Ausgabe von `otter` enthält zahlreiche Informationen, die wichtigsten davon sind:

- Die Angabe, ob `otter` die Herleitung der leeren Klausel gelungen ist. In diesem Fall findet man den Satz

  ```
  That finishes the proof of the theorem.
  ```

  ganz am Ende der Ausgabe. Andernfalls fehlt dieser Satz und `otter` gibt vor der Statistik folgenden Satz aus:

  ```
  Search stopped because sos empty.
  ```

- Der Resolutionsbeweis, d.h. die Herleitung der leeren Klausel, falls eine solche gefunden wurde. In obigem Beispiel erhält man

```
1   [] a|b.
2   [] -a|b.
3   [] -b|-r|-a.
4   [] r.
5   [] a|-b.
6   [binary,2.1,1.1] b.
7   [binary,5.2,6.1] a.
8   [binary,3.1,6.1] -r|-a.
9   [binary,8.1,4.1] -a.
10  [binary,11.1,7.1] F.
```

Die Information zu einem Resolutionsschritt enthält die Nummer der neu erzeugten Klausel (z.B. 7), die Angabe dass die Klausel 7 ein Resolvent der Klauseln 5 und 6 ist, wobei nach dem zweiten Literal von Klausel 5 und dem ersten Literal von Klausel 6 resolviert wurde ([binary,5.2,6.1]) und dass die Klausel 7 nur aus der atomaren Formel a besteht. Die leere Klausel wird repräsentiert durch $F. Aus diesen Informationen kann man dann den graphischen Resolutionsbeweis leicht herauslesen.

9 Theorembeweiser Teil 2

9.1 Logische Grundlagen

Um die Funktionsweise von Theorembeweisern verstehen zu können, sind gewisse Kenntnisse in der Logik, insbesondere der Aussagen-, Prädikaten- und in bestimmten Fällen auch der Gleichungslogik erforderlich. Dieses Kapitel soll die wichtigsten Begriffe und Methoden vorstellen.

9.1.1 Aussagenlogik

Siehe hierzu Kapitel 8.2 auf Seite 77

9.1.2 Prädikatenlogik

Die folgenden Definitionen wurden angelehnt an [CB03].

Signaturen und Interpretation In der Prädikatenlogik können, im Gegensatz zur Aussagenlogik, mithilfe von Funktionen und Relationen, deutlich komplexere Zusammenhänge formuliert werden. Eine Signatur Σ stellt ein Vokubular zur Verfügung, mit dem dies realisiert werden kann.

Definition (Signatur) Eine (PL1-) Signatur $\Sigma = (Func, Pred)$ besteht aus einer Menge $Func$ von *Funktionssymbolen* und einer Menge $Pred$ von *Prädikatensymbolen*. Diesen Symbolen sind feste Stelligkeiten ≥ 0 zugeordnet. Ein Funktionssymbol mit der Stelligkeit 0 heißt *Konstante*.

Eine Interpretation I weist den Symbolen einer Signatur Bedeutungen über einer Menge von Objekten zu. Hierfür notwendig ist ein Universum U, das eine beliebige, nichtleere Menge ist. Sie enthält alle Objekte der Interpretation.
Den Funktions- und Prädikatensymbolen werden durch I Funktionen bzw. Relationen zugeordnet. Nullstelligen Funtionssymbolen 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. Zuletzt werden mehrstellige Prädikatensymbole durch Relationen entsprechender Stelligkeit über U interpretiert.

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

- einer nichtleeren Menge U_I, dem Universum

- einer Menge $Func_I = \{f_I : \underbrace{U_I \times \ldots \times U_I}_{n-mal} \rightarrow U_I | f \in Func$ mit Stelligkeit $n \geq 0\}$ von Funktionen
- einer Menge $Pred_I = \{p_I \subseteq \underbrace{U_I \times \ldots \times U_I}_{m-mal} | p \in Pred$ mit der Stelligkeit $m \geq 0\}$ von Prädikaten

Terme (Syntax) In der Prädikatenlogik können im Gegensatz zur Aussagenlogik Terme formuliert werden, die ausgewertet durch ein Objekt des Universums interpretiert werden.

Definition (Term) Die Menge der Terme $Term_\Sigma(V)$ über einer Signatur $\Sigma = (Func, Pred)$ und einer Menge V von Individuenvariablen, ist die kleinste Menge, die folgende Elemente enthält:

- x, falls $x \in V$
- c, falls $c \in Func$ mit Stelligkeit 0
- $f(t_1, \ldots, t_n)$, falls $f \in Func$ mit Stelligkeit $n > 0$ und $t_1, \ldots, t_n \in Term_\Sigma(V)$

Ein Term, der keine Variablen enthält, heißt Grundterm.

Terme (Semantik)

Definition (Variablenbelegung) Sei $I = (U_I, Func_I, Pred_I)$ eine Interpretation und V eine Menge von Individuenvariablen. Eine *Variablenbelegung* ist eine Abbildung $\alpha : V \rightarrow U_I$.

Definition (Termauswertung) Seien $t \in Term_\Sigma(V)$ ein Term, I eine Interpretation und α eine Variablenbelegung. Die Termauswertung ist eine Funktion $[\![\_]\!]_{I,\alpha} : Term_\Sigma(V) \rightarrow U_I$ mit

$$[\![x]\!]_{I,\alpha} = \alpha(x)$$
$$[\![f(t_1, \ldots, t_n)]\!]_{I,\alpha} = f_I([\![t_1]\!]_{I,\alpha}, \ldots, [\![t_n]\!]_{I,\alpha})$$

Formeln (Syntax)

Definition (atomare Formel) Eine atomare Formel oder ein Atom über einer Signatur $\Sigma = (Func, Pred)$ und einer Menge V von Individuenvariablen hat die Form:

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

Komplexere Formeln werden nun aus den, aus der Aussagenlogik bekannten, Junktoren $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ und den Quantoren

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

gebildet.

Definition (Formel) Sei $\Sigma = (Func, Pred)$ und V eine Menge von Individuenvariablen. Die Menge $Formel_\Sigma(V)$ ist definiert als die kleinste Menge, die folgende Elemente enthält:

- P, falls P eine atomare Formel ist
- $\neg F, F_1 \wedge F_2, F_1 \vee F_2, F_1 \Rightarrow F_2, F_1 \Leftrightarrow F_2$, mit $F, F_1, F_2 \in Formel_\Sigma(V)$
- $\forall x F, \exists x F$, mit $x \in V$ und $F \in Formel_\Sigma(V)$

Ein Literal ist ein Atom A oder ein negiertes Atom $\neg A$. Eine Formel, in der keine Variablen vorkommen, heißt Grundformel.

Formeln (Semantik)

Definition (Formelauswertung) Einer atomaren prädikatenlogischen Formel kann nun folgendermaßen ein Wahrheitswert zugeordnet werden:

- $p(t_1, \ldots, t_n) \equiv true$ gdw. $([\![t_1]\!]_{I,\alpha}, \ldots, [\![t_n]\!]_{I,\alpha}) \in p_I$
- $p(t_1, \ldots, t_n) \equiv false$ gdw. $([\![t_1]\!]_{I,\alpha}, \ldots, [\![t_n]\!]_{I,\alpha}) \notin p_I$

Entsprechend den Junktoren werden prädikatenlogischen Formeln folgendermaßen Wahrheitswerte zugeordnet:

- $p(t_1, \ldots, t_n) \wedge q(s_1, \ldots, s_m) \equiv true$ gdw. $([\![t_1]\!]_{I,\alpha}, \ldots, [\![t_n]\!]_{I,\alpha}) \in p_I$ und $([\![s_1]\!]_{I,\alpha}, \ldots, [\![s_m]\!]_{I,\alpha}) \in q_I$, sonst *false*

- $p(t_1, \ldots, t_n) \vee q(s_1, \ldots, s_m) \equiv true$ gdw. $([\![t_1]\!]_{I,\alpha}, \ldots, [\![t_n]\!]_{I,\alpha}) \in p_I$ oder $([\![s_1]\!]_{I,\alpha}, \ldots, [\![s_m]\!]_{I,\alpha}) \in q_I$, sonst $false$
- $\neg p(t_1, \ldots, t_n) \equiv true$ gdw. $([\![t_1]\!]_{I,\alpha}, \ldots, [\![t_n]\!]_{I,\alpha}) \notin p_I$, sonst $false$

$\Rightarrow$ und $\Leftrightarrow$ werden gemäß ihrer Definitionen auf analoge Weise ausgewertet. Quantifizierten Formeln werden so ausgewertet:

- $\forall x F \equiv true$ gdw. für **alle** $c \in U_I$ gilt $[\![F]\!]_{I,\alpha_{x/c}} = true$
- $\exists x F \equiv true$ gdw. es gibt **ein** $c \in U_I$ mit $[\![F]\!]_{I,\alpha_{x/c}} = true$

Tritt eine Variable ohne einen sie umgebenden Quantor auf, so heißt sie frei, ansonsten gebunden.

Ein zentrales Problem in der Prädikatenlogik ist die Frage nach der Erfüllbarkeit gegebener Formeln. In der kontrollierten Anfrageauswertung ist außerdem die Frage, ob man aus einer gegebenen Formel eine andere Formeln folgern kann, von besonderem Interesse. Es gibt keinen Algorithmus, der für eine gegebene prädikatenlogische Formel entscheidet, ob diese erfüllbar ist. Mit der prädikatenlogischen Resolution existiert allerdings ein Verfahren mit dem man ihre *Unerfüllbarkeit* nachweisen kann. Man nennt das Resolutionsverfahren deshalb *widerlegungsvollständig*. Um die Frage, ob $F \models G$ gilt, zu beantworten, zeigt man die Unerfüllbarkeit von $F \wedge \neg G$. Um die prädikatenlogische Resolution anwenden zu können, sind allerdings einige vereinfachende Umformungen an den Formeln vorzunehmen. Die Formeln werden in sogenannte *Normalformen* überführt.

Normalformen Im Folgenden werden die notwendigen Schritte angegeben, um eine prädikatenlogische Formel, in eine äquivalente prädikatenlogische Formel zu überführen, in der alle Quantoren außen stehen. Eine solche Form nennt sich *Pränexform*. Wenn in der Pränexform nur noch die Junktoren $\neg, \wedge, \vee$ vorkommen, wobei Negationen nur vor Atomen stehen, so heißt sie *verneinungstechnische Normalform*. Um zu dieser Form zu gelangen sind allerdings einige Äquivalenzen zu betrachten, welche man hierzu heranziehen muss:

$$
\begin{array}{lll}
(1.1) & (\forall x F) \wedge G & \equiv \quad \forall x (F \wedge G) \\
(1.2) & (\forall x F) \vee G & \equiv \quad \forall x (F \vee G) \\
(1.3) & (\exists x F) \wedge G & \equiv \quad \exists x (F \wedge G) \\
(1.4) & (\exists x F) \vee G & \equiv \quad \exists x (F \vee G)
\end{array}
$$

$$
\begin{array}{lll}
(2.1) & (\forall x F) \wedge (\forall x G) & \equiv \quad \forall x (F \wedge G) \\
(2.2) & (\exists x F) \vee (\exists x G) & \equiv \quad \exists x (F \vee G)
\end{array}
$$

$$
\begin{array}{lll}
(3.1) & \forall x F & \equiv \quad \forall y F[x/y] \\
(3.2) & \exists x F & \equiv \quad \exists y F[x/y]
\end{array}
$$

$F[x/y]$ bezeichnet die Formel, die man erhält, wenn man jedes Vorkommen von x

in F durch y ersetzt. Bei den Regeln unter (1) darf x nicht in G frei vorkommen. Bei den Regeln unter (3) muss y ein neues Variablensymbol sein, darf also in F nicht vorkommen.
Nun kann die verneinungstechnische Normalform durch folgende Schritte erzeugt werden:

1. Erreiche durch geeignete Variablenumbenennungen, dass keine Variable sowohl frei als auch gebunden vorkommt. Hierbei sind die Regeln unter (3) heranzuziehen.

2. Ersetze die Junktoren $\Rightarrow$ und $\Leftrightarrow$, gemäß Definition, durch $\neg, \wedge, \vee$.

3. Wende die de Morgan'schen Regeln an, sodass Negationen nur noch unmittelbar vor Atomen auftreten.

4. Mit den Äquivalenzen aus (2) und (3) lassen sich nun alle Quantoren nach außen schieben.

Um die (prädikatenlogische) Resolution anwenden zu können, sind allerdings noch weiter Vereinfachungsschritte notwendig. Hierbei geht es um das Eliminieren der Existenzquantoren. Die Methode um dies zu erreichen heißt *Skolemisierung*. Jeder Existenzquantor, der nicht im Geltungsbereich eines Allquantors vorkommt, kann einfach weggelassen werden. Die durch diesen Quantor gebundenen Variablen werden jeweils durch ein neues Konstantensymbol, eine sogenannte*Skolemkonstante*, ersetzt. Wenn $\exists x$ allerdings im Geltungsbereich der allquantifizierten Variablen $y_1, \ldots, y_m$ auftritt, so muss jedes Auftreten von x durch $f(y_1, \ldots, y_m)$ ersetzt werden, wobei f ein neues Funktionssymbol, eine sogenannte *Skolemfunktion*, ist. So ist z.B. $P(c)$ die Skolemisierung von $\exists x P(x)$ oder $\forall y P(f(y), y)$ die Skolemisierung von $\forall y \exists x P(x, y)$. Es gilt zu beachten, dass die so erzeugten Formeln nur noch *erfüllbarkeitsäquivalent* zur Ausgangsformel sind. Die folgenden beiden Schritte vervollständigen die Liste der Vereinfachungsschritte:

5. Entferne alle Existenzquantoren durch Skolemisierung.

6. Alle Variablen sind nun allquantifiziert und die Allquantoren können einfach weggelassen werden.

Mithilfe der de Morgan'schen Regeln kann man nun die KNF der Formel bilden. Die KNF hat die Form $(L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{m,1} \vee \ldots \vee L_{m,n_m})$, wobei die $L_{i,j}$ Literale bezeichnen. Die KNF kann dann in ihrer *Klauselform* $c(F)$ als Menge dargestellt werden. Obige Formel hätte die Klauselform
$c(F) = \{\{L_{1,1}, \ldots, L_{1,n_1}\}, \ldots, \{L_{m,1}, \ldots, L_{m,n_m}\}\}$
Die Klauselform ist der Ausgangspunkt der prädikatenlogischen Resolution. Um diese anwenden zu können, muss noch der Begriff der Unifikation eingeführt werden.

Definition (Unifikation) Eine Substitution σ heißt *Unifikation* der Terme $t_1, \ldots, t_n$, wenn $\sigma(t_1) = \ldots = \sigma(t_n)$ gilt; $t_1, \ldots, t_n$ heißen dann *unifizierbar*.
Ein Unifikator μ von $t_1, \ldots, t_n$ heißt *allgemeinster Unifikator* (mgu), wenn es zu jedem Unifikator σ von $t_1, \ldots, t_n$ eine Substitution σ' mit $\sigma = \sigma' \circ \mu$ gibt.

Nun liegt das notwendige Handwerkszeug vor, um die Resolution durchzuführen. Idee der Resolution ist die folgende Inferenzregel:

$$\frac{\{L,K_1,...,K_n\}\{\neg L',M_1,...,M_m\} \quad \sigma(L)=\sigma(L')}{\sigma(K_1),...,\sigma(K_n),\sigma(M_1),...,\sigma(M_m)\}}$$

mit $\sigma = mgu(L, L')$.
Gelingt es, in endlich vielen Schritten die leere Menge abzuleiten, so hat man einen elementaren Widerspruch gefunden und die Formel ist unerfüllbar.

Grundinstanziierung Jede prädikatenlogische Formel, die über einem **endlichen** Universum definiert ist, kann mithilfe der Grundinstaziierung als eine endliche Konjunktion/Disjunktion dargestellt werden. Seien $c_1, c_2, c_3, \ldots, c_n$ Konstanten, die alle Objekte des Universums bezeichnen. Dann gilt:

- $\forall x F \equiv F[x/c_1] \wedge F[x/c_2] \wedge F[x/c_3] \wedge \ldots \wedge F[x/c_n]$
- $\exists x F \equiv F[x/c_1] \vee F[x/c_2] \vee F[x/c_3] \vee \ldots \vee F[x/c_n]$

Auf diese Weise dargestellt, liegt offensichtlich eine Formel vor, die sich als eine äquivalente aussagenlogische Formel darstellen läßt. Somit ist die PL1 über einem endlichen Universum entscheidbar. Allerdings ist das Universum in Datenbanken potentiell unendlich. Deshalb sind diese Probleme im Kontext der kontrollierten Anfrageauswertung nicht entscheidbar.

9.1.3 Gleichungslogik

Die Informationen der nächsten Abschnitte stammen größtenteils von [NR01] und [Sch06].
Ebenfalls von großem Interesse ist häufig die Frage nach der Erfüllbarkeit gleichungslogischer Formeln. Deshalb implementieren die meisten Theorembeweiser auch ein Inferenzsystem für dieses logische System. In diesem Kapitel sollen zunächst die Grundbegriffe der Gleichungslogik erörtert werden, um diese dann um ein Inferenzverfahren zu erweitern.

Die Syntax der Gleichungslogik Die Definitionen der Begriffe *Variable, Funktion* und *Term* werden aus der Prädikatenlogik übernommen. Eine gleichungslogische Formel hat dann die Form $s \simeq t$, wobei s und t Terme wie in der Prädikatenlogik sind. Dabei kann die Gleichheit als das einzig existierende Prädikat aufgefaßt werden. Eine gleichungslogische Formel könnte beispielsweise so aussehen: $f(f(x, y), z) \simeq f(x, f(y, z))$ bzw. in Präfixnotation: $\simeq (f(f(x, y), z), f(x, f(y, z)))$.

Die Semantik der Gleichungslogik Der Wahrheitswert einer gleichungslogischen Formel unter einer gegebenen Interpretation I ist folgendermaßen definiert:

- $s \simeq t \equiv true$ gdw. $I(s) = I(t)$ für jede Variablenbelegung α, sonst $s \simeq t \equiv false$

In der Gleichungslogik hat ein Literal die Form $s \simeq t$. Entsprechend werden Klauseln als $(s_1 \simeq t_1 \vee s_2 \simeq t_2 \vee \ldots \vee s_n \simeq t_n)$ dargestellt.
Theoretisch ist es möglich gleichungslogische Formeln ebenfalls mithilfe der Resolution zu behandeln, um die Frage nach deren Unerfüllbarkeit zu beantworten. Hierzu ist es notwendig der Formelmenge eine Menge von *Kongruenzaxiomen* hinzuzufügen:

$$\rightarrow x \simeq x \qquad \text{(Reflexivität)}$$
$$x \simeq y \rightarrow y \simeq x \qquad \text{(Symmetrie)}$$
$$x \simeq y \wedge y \simeq z \rightarrow x \simeq z \qquad \text{(Transitivität)}$$
$$x_1 \simeq y_1 \wedge \ldots \wedge x_n \simeq y_n \rightarrow f(x_1, \ldots, x_n) \simeq f(y_1, \ldots, y_n) \qquad \text{(Monotonie I)}$$
$$x_1 \simeq y_1 \wedge \ldots \wedge x_n \simeq y_n \wedge P(x_1, \ldots x_n) \rightarrow P(y_1, \ldots y_n) \qquad \text{(Monotonie II)}$$

Würde man eine gleichungslogische Formel auf diese Weise behandeln, kann dies schnell zu einer explosionsartig wachsenden Formelmenge führen, denn die Anzahl möglicher Ersetzungen ist in den meisten Fällen extrem groß. Und nur die wenigsten Ersetzungen führen tatsächlich zum Ziel. Genau deshalb werden hier spezielle Inferenzregeln eingeführt, die eine effizientere Behandlung gleichungslogischer Formeln erlauben.

Ein Inferenzsystem für die Gleichungslogik Innerhalb eines geeigneten Inferenzsystems für gleichungslogische Formeln soll zunächst die *Paramodulation* betrachtet werden. Seien $s \simeq t$ ein Literal und $C \vee s \simeq t$ und D Klauseln. Paramodulation ist dann die Inferenz:

$$\frac{C \vee s \simeq t \quad D}{\sigma(C \vee D[t]_p)} \quad \text{mit } \sigma = mgu(s, D|_p)$$

hierbei bezeichnet $D|_p$ den Teilterm an der Stelle p in D. $D[t]_p$ bedeutet, dass der Teilterm an der Stelle p in D durch t ersetzt wird.
Paramodulation basiert auf der Idee des Ersetzens von Gleichem durch Gleiches. Diese Regel allein bildet aber noch kein geeignetes Inferenzsystem, sondern ist lediglich die Basis eines Ersetzungssystems. Gerade bei großen Klauselmengen, können durch Paramodulation extrem viele neue Klauseln hergeleitet werden, was im Allgemeinen aber nicht zum Ziel führt, die leere Menge zu erzeugen. Die entscheidende Einschränkung besteht darin, nur größere Terme durch kleinere zu ersetzen. Es wird also eine Ordnung auf den Termen benötigt, um diese miteinander vergleichen zu können. Dieses Verfahren nennt sich geordnete *Termersetzung*.

Definition (A-Ordnung) Eine A-Ordnung $<_a$ ist eine irreflexive und transitive binäre Relation auf atomaren Formeln mit:

1. Für jede Klauselmenge gibt es eine Aufzählung ihrer Atome $A_1, A_2, \ldots$, sodass für $A_i <_a A_j$ gilt: $i < j$

2. Aus $A_i <_a A_j$ folgt $\sigma(A_i) <_a \sigma(A_j)$

Diese Definition beschreibt noch keine konkrete Ordnung, sondern charakterisiert sie lediglich durch die notwendigen Eigenschaften. Ziel ist es, größere Terme durch kleinere Terme zu ersetzen. Also benötigt man eine konkrete Ordnung, die die „Komplexität“ von Termen misst. Eine in Theorembeweisern üblicherweise herangezogene Ordnung, ist die *Knuth-Bendix-Ordnung.*

Definition (Knuth-Bendix-Ordnung) Sei t ein prädikatenlogischer Term und $\#t$ die Komplexität von t, wobei jede Variable die Wertigkeit 1 besitzt. Sei $\#_x(t)$ die Anzahl der Vorkommen der Variablen x in t.

- Aus $\#t_1 > \#t_2$ und $\#_x(t_1) \geq \#_x(t_2)$ für jede Variable x folgt $t_1 >_{kbo} t_2$
- Sind t_1, t_2 keine Variablen , $\#t_1 = \#t_2$ und $\#_x(t_1) \geq \#_x(t_2)$ für jede Variable x. Dann gilt $t_1 = f(u_1, \ldots u_n)$ und $t_2 = g(v_1, \ldots, v_m)$.
  - Gilt nun $f > g$, so folgt $t_1 >_{kbo} t_2$.
  - Gilt nun $f = g$ und $n = m$, so folgt $t_1 >_{kbo} t_2$, wenn $(u_1, \ldots u_n) >_{lex} (v_1, \ldots v_m)$.
- Sonst $t_1 >_{kbo} t_2$, wenn $t_1 >_{lex} t_2$

Der letzte Fall wurde von mir eingefügt, da die Fallunterscheidung ansonsten offensichtlich unvollständig wäre. In der von mir herangezigenen Literatur wird der letzte Fall nicht erwähnt.

Definition (Termersetzung) Der Term t entsteht in einem Schritt aus s, wenn es eine Gleichung $F : \widetilde{s} \simeq \widetilde{t}$ und eine Substitution σ gibt, sodass gilt:

1. $\sigma(\widetilde{s})$ ist Teilterm von s.
2. t entsteht aus s, indem man $\sigma(\widetilde{s})$ an genau einer Stelle durch $\sigma(\widetilde{t})$ ersetzt.

mit $\sigma(\widetilde{s}) >_{kbo} \sigma(\widetilde{t})$. Man schreibt $s \rightarrow_F t$.
Mithilfe der vorgestellten Begriffe lässt sich nun ein widerlegungsvollständiges Inferenzsystem für die Gleichungslogik aufstellen. Es seien im Folgenden einige wichtige Expansions- und Kontraktionsregeln vorgestellt.

Expansionsregeln (Die Resolventen werden der Klauselmenge hinzugefügt)

$$\text{Superposition} \quad \frac{s \simeq t \vee S \qquad u \simeq v \vee R}{\sigma(u[p \leftarrow t] \simeq v \vee S \vee R)}$$

mit $\sigma = mgu(u|_p, s), \sigma(s) \not\preceq \sigma(t), \sigma(u) \not\preceq \sigma(v), \sigma(s \simeq t)$ ist $>_{kbo}$ $-maximal$ in $\sigma(u \simeq v \vee R)$ und $u|_p$ ist keine Variable.

$$\text{Equality Resolution} \quad \frac{u \not\simeq v \vee R}{\sigma(R)}$$

mit $\sigma = mgu(u, v)$ und $\sigma(u \not\simeq v)$ ist $>_{kbo}$ $-maximal$ in $\sigma(u \not\simeq v \vee R)$.

$$\text{Equality Faktorisierung} \quad \frac{s \simeq t \vee u \simeq v \vee R}{\sigma(t \not\simeq v \vee u \simeq v \vee R)}$$

mit $\sigma = mgu(s, u)$ und $\sigma(s) \not\preceq \sigma(t)$ und $\sigma(s \simeq t)$ ist $>_{kbo}$ $-maximal$ in $s \simeq t \vee u \simeq v \vee R$.

Kontraktionsregeln (Die Resolventen ersetzen die Elternklauseln)

$$\text{Subsumption} \quad \frac{C \quad \sigma(C \vee R)}{C}$$

mit C und R sind (Teil)Klauseln und σ ist eine Substitution.

$$\text{Equality Subsumption} \quad \frac{s \simeq t \quad u[p \leftarrow \sigma(s)] \simeq u[p \leftarrow \sigma(t)] \vee R}{s \simeq t}$$

$$\text{Tautologie Entfernung} \quad \frac{C}{}$$

mit C ist eine Tautologie.

Insgesamt ist dieses System widerlegungsvollständig und korrekt. Es existieren weitere Möglichkeiten, diesem System Regeln hinzuzufügen. Allerdings reichen Paramodulation, Resolution und Faktorisierung für die Korrektheit aus. Ein Theorembeweiser muss also nicht alle Regeln implementieren. Verschiedene Theorembeweiser implementieren unterschiedliche Inferenzregeln. Die drei zur Korrektheit notwendigen Inferenzen **müssen** und die Erkennung von Tautologien und Subsumptionen **sollten** allerdings enthalten sein.

9.2 Klassen von Theorembeweisern

Aktuelle Implementierungen von Theorembeweisern arbeiten vor allem mit einem (oder mehreren) der drei folgenden Verfahren:

- Resolution (PL1)
- Tableauverfahren (PL1)

- Paramodulation/Termersetzung (Gleichungslogik)

Es soll nun auf einige wichtige Prinzipien eingegangen werden, die bei Theorembeweisern dringend eingehalten werden sollten, um die Ineffizienz einzugrenzen. Wichtige Fragen sind vor allem:

- Nach welchen Kriterien wählt man die beiden Elternklauseln für den nächsten Resolutionsschritt aus?
- Wie hält man die Klauselmenge möglichst gering?

Um eine Antwort auf die erste Frage geben zu können, betrachtet man die aus Kapitel 2.3.3 bekannte A-Ordnung. Für einen Theorembeweiser, der auf Resolution basiert, könnte eine konkrete A-Ordnung z.B. so aussehen:

Beschreibe Variablen, Terme und Atome durch ihre funktionale Größe:

- Größe einer Variablen: 1
- Größe eines Terms $f(t, \ldots, t_n)$: 1 + Maximum der Größen von $t_1, \ldots t_n$.
- Größe eines Atoms: Maximum der Größen seiner Argumente

Die entsprechende Restriktion ist nun die gleiche wie bei der geordneten Termersetzung. Es muss für jede Resolvente c mit resolviertem Literal L gelten: L ist $<_a -maximal$. Diese Einschränkung hält die funktionale Schachtelung in den Klauseln gering.
Es existieren noch weitere mögliche Auswahlverfahren für die Elternklauseln eines Resolutionschritt, auf die allerdings hier nicht weiter eingegangen werden soll.

Die zweite Frage lässt sich durch die Eliminierung von Redundanzen beantworten. Dies ist eine elementare Technik, die in praktisch allen Theorembeweisern verwendet wird. Ohne diese Technik wären Theorembeweiser nicht praktikabel. Redundanzen treten in zwei Ausprägungen auf: Als Tautologien und als Subsumptionen. Nach jedem Resolutionsschritt müssen folgende Schritte durchgeführt werden:

- Entferne alle Tautologien. Eine Klausel ist eine Tautologie, wenn sie die Literale L und $\neg L$ enthält.
- Eine Klausel c_1 subsumiert eine Klausel c_2, wenn eine Substitution existiert, so dass $\sigma(c_1) \subseteq c_2$ gilt. In diesem Fall kann c_2 entfernt werden.

In ähnlicher Weise arbeiten auch Theorembeweiser basierend auf Gleichungslogik. Die Eliminierung von Redundanzen ist somit ein essenzieller Bestandteil des (maschinellen) Beweisprozesses.

9.3 Der Theorembeweiser E

E ist ein Theorembeweiser für Gleichungslogik basierend auf dem Superpositionsverfahren. Er ist unter allen gängigen Unix-Derivaten auf der Kommandozeile lauffähig. Die Klauseln werden als Textdateien in das Hauptprogramm eingegeben. Innerhalb dieser Textdateien müssen die Klauseln in LOP verfaßt sein. LOP ähnelt in seiner Syntax PROLOG. Die Formeln müssen nicht erst in KNF überführt werden. Dies führt E von allein aus. Auch können die Eingaben prädikatenlogische Formeln sein. E erzeugt automatisch Formeln der Gleichungslogik. Informationen über E und über LOP sind unter *www.eprover.org* verfügbar. Eine mögliche Eingabe könnte so aussehen:

```
f(f(X,Y),Z)=f(X,f(Y,Z)).
  animal(X) <- wolf(X).
```

Diese Beispiele sollten relativ selbsterklärend sein.

Wenn man die Inferenzregeln von E betrachtet, stellt man fest, dass man exakt die Regeln wiederfindet, die bereits in 2.3.3 vorgestellt wurden. Zusätzlich wurde in E sehr viel Wert auf starke und effiziente Kontraktionregeln, wie z.B. „simplify-reflect“, gelegt. Diese sind allerdings sehr technischer Natur und sollen hier nicht weiter ausgeführt werden. Stattdessen sollen hier eingehender der Suchalgorithmus und die Heuristiken zur Suche von passenden Klauseln beleuchtet werden.

9.3.1 Suchalgorithmus und Suchheuristiken

Der Suchalgorithmus zieht in jedem Durchlauf zunächst sogenannte *Prioritäts-* und *Gewichtsfunktionen* heran. Anhand dieser wird die Reihenfolge bestimmt, in der die Klauseln behandelt werden. Für jede ausgewählte Klausel werden als erstes die Kontraktionsregeln und Vereinfachungen angewendet und geprüft, ob hierdurch die leere Menge entstanden ist. Ist dies nicht der Fall, so werden die Expansionregeln angewendet. Die Kontraktions- und Vereinfachungsregeln werden dann wiederum auf jede neu gewonnene Klausel angewendet.

Suchheuristiken E bietet einige Möglichkeiten, auf den Auswahlprozess für die Klauseln Einfluss zu nehmen. Hierfür stehen dem Benutzer die bereits oben genannten Prioritäts- und Gewichtsfunktionen zur Verfügung. Diese sollen nun näher beschrieben werden.

Prioritätsfunktionen Diese geben an, welche Klauseln bevorzugt (d.h. als erstes) behandelt werden sollen. Hier kann z.B. angegeben werden, dass Grundatome oder soeben erst neu erzeugte Klauseln eine höhere Priorität gegenüber anderen Klauseln haben sollen.

Gewichtsfunktionen Mit diesen Funktionen werden Klauseln mit der gleichen Priorität verglichen. Das Gewicht einer Klausel ergibt sich dabei aus den Gewichten der Variablen und Funktionen, die in ihr enthalten sind. Welches Gewicht einer Variablen bzw. einer Funktion zugewiesen wird, kann frei gewählt werden. Es werden diejenigen Klauseln zuerst behandelt, die ein geringeres Gewicht haben.

Zur Ordnung der Terme stehen in E zwei mögliche Standardordnungen zur Verfügung. Die Knuth-Bendix-Ordnung und die lexikographische Ordnung. Bei der lexikographischen Ordnung werden die Terme wie Stringsequenzen behandelt. Diese Ordnung wird standarmäßig nicht benutzt, sondern dient lediglich als generische Ordnung, falls keine spezielle Ordnung gewünscht wird.

9.3.2 Ausgaben von E

Der einfachste Ausgabemodus von E ist der sogeannte *Silentmode*. Hier wird, nachdem der Algorithmus terminiert hat, lediglich die Ausgabe gemacht, zu welchem Ergebnis E bezüglich der Erfüllbarkeit der Klauselmenge gekommen ist. Es gibt im Wesentlichen drei unterschiedliche Möglichkeiten:

- `Proof found!`
- `No proof found!`
- `Failure: Resource limit exceeded (memory/time)`

Lautet die Ausgabe `Proof found!`, so konnte die leere Menge abgeleitet werden und die Klauselmenge ist unerfüllbar. Sind alle Klauseln behandelt und eine Ableitung der leeren Menge ist nicht gelungen, so lautet die Ausgabe `No proof found!`. Die dritte Möglichkeit besteht darin, dass die vorgegebene maximale Laufzeit bzw. der zur Verfügung stehende Speicher ausgeschöpft ist. In diesem Falle wird von E `Failure: Resource limit exceeded (memory/time)` ausgegeben. Es existieren zwei weitere mögliche Fehlschlagsituationen, die allerdings nur in Spezialfällen auftreten können. Desweiteren stehen E verschiedene Ausgabemodi zur Verfügung, in denen E während des Beweisprozesses weitere Ausgaben macht. So z.B. welche Klauseln gerade behandelt und welche neuen Klauseln gewonnen wurden.

Die Möglichkeit, zwischen verschiedenen Ausgabemodi zu wählen und die Flexibilität bei der Eingabe lassen E geeignet erscheinen, an ein System zur kontrollierten Anfrageauswertung angebunden zu werden.

Beispiel einer Ableitung Abschließend soll noch eine mögliche Ableitungsfolge von E betrachtet werden, um den Beweisprozess an einem Beispiel zu verdeutlichen:

(1)	$f(a) \simeq a$	initial
(2)	$f(f(f(X))) \not\simeq a \vee X \simeq b$	initial
(3)	$P(a) \quad (P(a) \simeq \top)$	initial
(4)	$\neg P(b) \quad (P(b) \simeq \bot)$	initial
(5)	$f(f(f(a))) \not\simeq a \vee a \simeq b$	Paramodulation 1 nach 2
(6)	$f(f(a)) \not\simeq a \vee a \simeq b$	Termersetzung 5 durch 1
(7)	$f(a) \not\simeq a \vee a \simeq b$	Termersetzung 6 durch 1
(8)	$a \not\simeq a \vee a \simeq b$	Termersetzung 7 durch 1
(9)	$a \simeq b$	Vereinfache 8
(10)	$P(b)$	Termersetzung 3 durch 9
(11)	$\emptyset$ Widerspruch gefunden	4 und 10

10 Projektmanagement

10.1 Einführung

10.1.1 Begriffe

Der Projektbegriff

Ein Projekt ist ein Vorhaben, das im Wesentlichen durch Einmaligkeit der Bedingungen in ihrer Gesamtheit gekennzeichnet ist, wie z.B.

- Zielvorgabe
- zeitliche, finanzielle, personelle oder andere Begrenzungen
- Abgrenzung gegenüber anderen Vorhaben
- projektspezifische „Organisaton“ (DIN69901)

Der Begriff „Management “

Management beinhaltet „die Planung, Durchführung, Kontrolle und Anpassung von Maßnahmen zum Wohl der Organisation bzw. des Unternehmens und alle daran Beteiligten“ unter Einsatz der „zur Verfügung stehenden betrieblichen Ressourcen.“

Der Begriff „Projektmanagement“

Projektmanagement ist ein Leitungs- und Führungskonzept für Projekte, welches

- den Entwicklungsprozess definiert,
- die notwendigen Aufgaben definiert,
- die Methoden für die Lösung der Aufgaben vorschlägt,
- Institutionen schafft und nutzt, von denen diese Aufgaben realisiert werden können,
- und abteilungsübergreifend arbeitet.

10.2 Projektmanagement und Prozessmodelle

10.2.1 Projektablauf

In der Startphase werden die Projektziele und die daraus abgeleiteten Anforderungen an das zu erstellende System ausgearbeitet. Anschließend wird die Planung von Arbeitspaketen mit Terminen, Kosten u.a. vorgenommen.Nach der Planungsphase muss

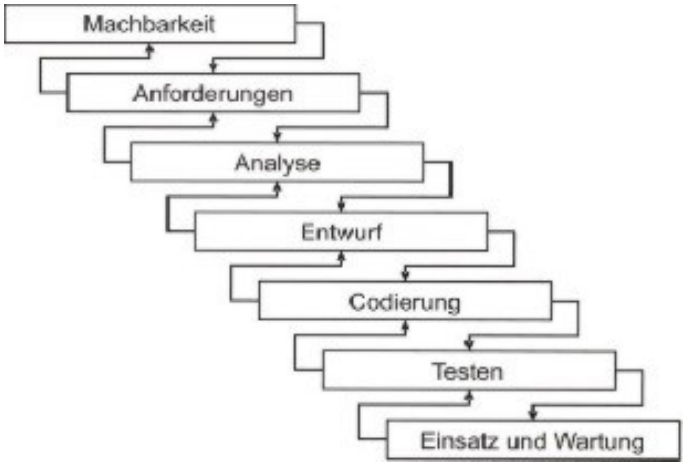

Abbildung 4: Wasserfallmodell

mit der Projektdurchführung begonnen werden (d.h. Kontrollieren und Steuern der Aufgaben und wenn notwendig Vornehmen von Plananpassungen). Und die letzte Phase, der Projektabschluss enthält die Auswertung, ob das Projekt erfolgreich oder nicht erfolgreich war, indem die Ursachen aufgezeigt und den Projektablauf dokumentiert werden sollen.

10.2.2 Prozessmodelle

Ein Prozessmodell ist eine abstrakte Beschreibung ähnlicher Softwareprozesse, also eine Musterstruktur für Organisation und Durchführung eines Softwareprozesses.

Wasserfallmodell Das Wasserfallmodell, wie Abbildung 4 zeigt, bezeichnet ein Vorgehensmodell in der Softwareentwicklung, bei dem die Software in definierten und abgetrennten Phasen entwickelt werden soll. Dabei gehen die Phasenergebnisse wie bei einem Wasserfall immer als bindende Vorgaben in die nächst tiefere Phase ein. Rückkopplung zwischen benachbarten Phasen ist erlaubt und z.T. auch notwendig. Übergreifende Rückkopplungen sind nicht erlaubt. Die Reihenfolge der einzelnen Phasen ist vorgeschrieben und jede Phase ist zu durchlaufen. Alle Ergebnisse einer Phase werden dokumentiert.

1. Phase: Machbarkeit
Der Zweck dieser Phase ist das Abschätzen des Gesamtprojekts, ob es überhaupt machbar ist.
Das Ergebniss sollte dann sein:
ein Lastenheft (enthält grobe Beschreibung der Anforderungen)
2. Phase: Anforderungsdefinition
In dieser zweiten Phase soll nun genau definiert werden, was die Software machen soll.
Ergebnisse:

Pflichtenheft (=Anforderungsdefinitionsdokument, Soll-Konzept)
Benutzerhandbuch (erste Version)
3. Phase: Analyse
In dieser Phase werden die Anforderungen überprüft, verfeinert und anschließend zerteilt.
Ergebnisse:
Analysedokument
Benutzerhandbuch (zweite Version)
4. Phase: Entwurf
Da jetzt alle Anforderungen und Aufgaben ausführlich definiert sind und vorliegen, wird nun ein konkreter Entwurf gefertigt. Nun wird genau festgelegt, wie was zu realisieren ist. Das heißt es wird ein Bauplan für die Software und die Softwarearchitektur entwickelt. Weiters werden konkrete Softwarebibliotheken, Frameworks und der gleichen ausgewählt.
Ergebnisse:
Entwurfsdokument mit Softwarebauplan
detaillierte Testpläne für Module, Klassen und Komponenten
5. Phase: Implementierung
Nachdem jetzt der Softwarebauplan vorliegt, geht man in die konkrete Implementierung des Systems in einer gewählten Programmiersprache über.
6. Phase: Test
Einzelne Module, Klassen und Komponenten werden schrittweise getestet und in das Gesamtsystem integriert. Am Ende wird dann noch einmal ein Gesamtsystemtest durchgeführt, den man als Alpha-Test bezeichnet.
7. Phase: Einsatz und Wartung
Die Software wird beim Kunden eingesetzt. Nach erfolgreicher Auslieferung und Integration der Software folgt die Phase der Wartung. Ihre Aufgaben liegen in der Fehlerbehebung und der Anpassung der Software, wenn nötig.

Spiralmodell Das Spiralmodell, wie Abbildung 5 zeigt, arbeitet mit Zyklen, von denen jeder im Wesentlichen einer Phase des Wasserfallmodells entspricht. In jedem Zyklus werden die folgenden Phasen durchlaufen:

- Festlegung von Zielen, Identifikation von Alternativen und Beschreibung von Rahmenbedingungen
- Evaluierung der Alternativen und das Erkennen, Abschätzen und Reduzieren von Risiken
- Realisierung und Überprüfung des Zwischenprodukts
- Review der abgelaufenen Phase und Planung der folgenden

Ein Zyklus im Spiralmodell entspricht einer Phase im Wasserfallmodell. Diese wird aber durch einleitende Risikoanalyse und abschließenden Review dynamischer als

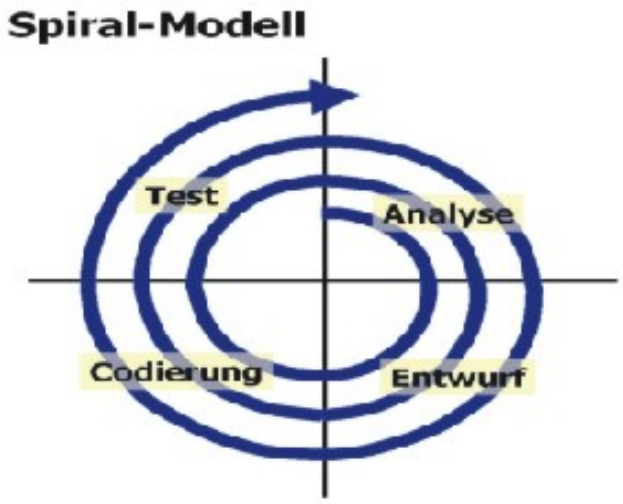

Abbildung 5: Spiralmodell

im Wasserfall-Modell an den Projektstand angepasst. Es ist eine Weiterentwicklung des Wasserfallmodells, in der die Phasen mehrfach spiralförmig durchlaufen werden. Durch das Spiralmodell wird das Risiko eines Scheiterns bei großen Softwareprojekten entscheidend minimiert, weil jede Phase mehrfach durchlaufen wird.

Extreme Programming (XP) Extreme Programming, wie Abbildung 5 zeigt, arbeitet mit kleinen Releases, unterteilt in Iterationen und Arbeitspakete. Ein Release ist eine Zusammenstellung von fertigen Softwareeinheiten, die in ihrer Gesamtheit eine bestimmte Funktionalität abdecken. Releases werden für einen Zeitraum bis zu 3 Monaten erstellt. Die Entwicklung erfolgt in Perioden von ein bis drei Wochen (kurze Iterationen). Am Ende jeder Iteration steht ein funktionsfähiges, getestetes System mit neuer, für den Kunden wertvoller Funktionalität. Die Arbeitspakete sollen in bis zu 3 Tagen erledigt sein.
Die Kunden halten ihre Anforderungen in Form einfacher Geschichten (User Storys), was das Produkt können soll (ca. 3 Sätze), auf gewöhnlichen Karteikarten fest. Jeder geschriebenen Story-Karte kommt das Versprechen nach, den genauen Funktionsumfang zum rechten Zeitpunkt im Dialog mit den Programmierern zu verfeinern und zu verhandeln.
Jede Iteration beginnt mit einem Planungsmeeting, in dem jedes Kundenteam seine Geschichten erzählt und mit den Programmierern diskutiert. Die Programmierer schätzen der Aufwand grob ab, den sie zur Entwicklung jeder einzelnen Geschichte benötigen werden. Die Programmierer zerlegen die geplanten Geschichten am Flipchart in technische Aufgaben, übernehmen Verantwortung für einzelne Aufgaben und schätzen deren Aufwände vergleichend zu früher erledigten Aufgaben. Aufgrund der genaueren Schätzung der kleineren Aufgaben verpflichten sich die Programmierer auf genau soviele Geschichten, wie sie in der vorhergehenden Iteration entwickeln konnten. Diese Planungsspiele schaffen eine sichere Umgebung, in welcher geschäftliche

und technische Verantwortung zuverlässig voneinander getrennt werden.
Das Release sollte mit den User Stories begonnen werden, die das höchste Risiko bezüglich Zeitplan und den höchsten Nutzen auf sich vereinen. Danach sind diejenige User Stories zu verwirklichen, die geringes Risiko aber hohen Nutzen haben. Anschließend geht das Team die User Stories an, die geringes Risiko und geringen Nutzen auf sich vereinigen. Die Fertigstellung von User Stories mit geringem Nutzen aber hohem Risiko ist zu vermeiden.
Die Kunden spezifizieren während der Iteration funktionale Abnahmekriterien. Die Tests sind ein wichtiger Punkt in diesem Modell. Es werden Tests für Storys geschrieben (Funktional Tests) und Tests für Klassen (Unit Tests) entwickelt. Unit Tests werden mit JUnit (Java-Framework zum Schreiben und Ausführen automatischer Unit Tests) wenn in Java implementiert wird, durchgeführt.
Unit Testing ist der Test von Programmeinheiten in Isolation von anderen im Zusammenhang eines Programms benötigten, mitwirkenden Programmeinheiten. Die Größe der unabhängig getesteten Einheit kann dabei von einzelnen Methoden über Klassen bis hin zu Komponenten reichen.
Die Refaktorisierung ist ein weiterer wichtiger Punkt. Sie dient zur ständigen Verbesserung der Lesbarkeit, Verständlichkeit, Wartbarkeit, Erweiterbarkeit und Struktur (der Funktionen und Klassen) eines Computerprogramms, ohne dabei die Funktionalität der Software zu verändern. Nach jedem Schritt müssen Unit Tests ausgeführt werden, um keine bestehende Funktion zu zerstören.
Die Programmierer arbeiten stets zu zweit am Code und diskutieren während der Entwicklung intensiv über Entwurfsalternativen. Sie wechseln sich minütlich an der Tastatur und rotieren stündlich ihre Programmierpartner. Das Ergebnis ist höhere Codequalität, größere Produktivität und bessere Wissensverarbeitung. Es gibt keinen privaten Code. Das Modell verzichtet auf umfangreiche Dokumentation. Man verlässt sich auf den gut leserlichen Quelltext. Das Modell ist für kleine und innovative Projekte (bis zu 15 Mitarbeitern) geeignet.

10.3 Projektstart

In der Projektstartphase werden die Anforderungen an die Software und das dazugehörige Budget zwischen Auftragnehmer und Auftraggeber vereinbart, die zu erwartenden Risiken analysiert und eine geeignete Struktur für die Projektabwicklung im Unternehmen geschaffen und zum Schluss ein verbindlicher Vertrag zwischen Auftraggeber und Auftragnehmer abgeschlossen.
Die Anforderungen werden in einem Dokument zusammengefasst, das als Vertragsgrundlage für Leistungen, Termine und Budget zwischen Auftraggeber und Auftragnehmer dient. Begriffe für dieses Dokument:

- Nach Institute of Electrical and Electronics Engineers (IEEE)- Software Requirements Specification
- Die ISO 9001 (internationale Qualitätsnorm) - Anforderungen
- Im deutschen Sprachraum - Lastenhefte bzw. Pflichtenhefte

10.4 Projektplanung

10.4.1 Definition

Planung ist die gedankliche Vorwegnahme des notwendigen Handelns im Projekt und geht den rationalen Entscheidungen voraus.

10.4.2 Planungsvorgehen

Die Reihenfolge der Planungen hängt vom Planungszustand (Erstplanung oder eine Überarbeitung), vom Detaillierungsgrad und vom betrachteten Zeitraum ab. Bei der Erstplanung muss zuerst die Aufgabenstruktur geplant werden.

Strukturplanung Die Strukturplanung enthält die Elemente Aufgabenstruktur und Projektstruktur.

- Aufgabenstruktur (Bildung von Arbeitspaketen): Unter Aufgabenstruktur ist die Summe der Arbeitspakete zu verstehen, die realisiert werden müssen, damit das Gesamtprodukt entstehen kann.
- Projektstruktur (Mitarbeiterzuordnung): Unter Projektstruktur wird das Zerlegen von Aufgaben in Arbeitspaketen verstanden, so dass sie einer Gruppe/ einem Mitarbeiter zugeordnet werden können.

Ablaufplanung In der Ablaufplanung werden die Abhängigkeiten zwischen den Arbeitspaketen festgelegt, d.h. es müssen Vorgänger und Nachfolger der Arbeitspakete bestimmt werden. Die Ablaufplanung hat Einfluss auf die Personalplanungen und Terminplanungen. Das Gelingen eines Personaleinsatzes hängt von der Definition und der Gestaltung der Abhängigkeiten ab, so dass arbeitsteilig gearbeitet werden kann.

Terminplanung Ohne Strukturplanung ist eine Terminplanung nicht möglich. Der Strukturplan enthält die Ablaufplanung (es werden die Vorgänger und Nachfolger der Arbeitspakete bestimmt). Für jedes Aufgabenpaket muss der Aufwand bekannt sein, denn die Dauer der Vorgänge ergibt sich aus dem Aufwand und dem eingesetzten Personal. Als Terminplanungsinstrument sind die Balkendiagramme zu benutzen. Die 6 zeigt, dass zum Bearbeiten eines Arbeitspakets mit Aufwand von 20 MannTagen, brauchen Chris und Peter 10 Tagen, die 17 Kalendertagen entsprechen.

Risikomanagement Das Risikomanagement erfolgt in folgenden Schritten:
Risikoidentifikation
Zunächst müssen die Risiken in Softwareprojekten identifiziert werden. Eine mögliche Technik zur Identifizierung von Risiken ist die Verwendung von Expertenwissen und standardisierten Fragebögen.

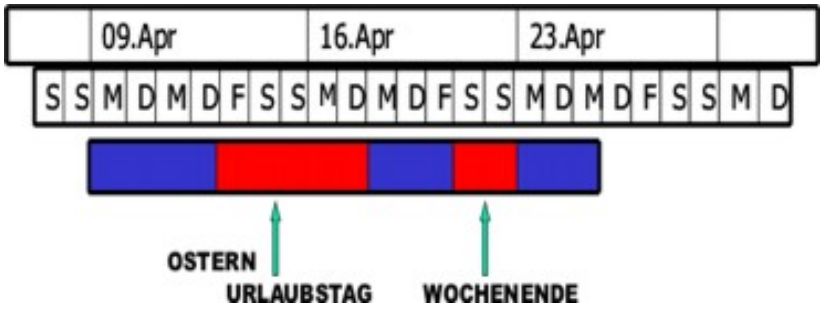

Abbildung 6: Balkendiagramm

Risikoanalyse
Nach der Risikoidentifikation erfolgt die Analyse der zuvor identifizierten Risiken. Die Risikoanalyse ist notwendig, um festzulegen, mit welcher Priorität die einzelnen Risiken behandelt werden müssen.
Risikoplanung
Nach Identifikation und Analyse der Risiken wird die Risikoplanung vorgenommen, die festlegt, wie mit einem identifizierten Risiko umzugehen ist. Mögliche Vorgehensweisen zur Behandlung eines Risikos sind Risikovermeidung, Risikoverminderung, Risikobegrenzung, Risikoverlagerung und Risikoakzeptanz.
Risikoüberwachung
Das Risikomanagement findet nicht nur zu Beginn des Softwareprojektes statt, sondern während der gesamten Projektlaufzeit muss eine Überwachung der Risiken und Maßnahmen zur Risikobehandlung stattfinden.

Projektpläne In einem Projektplan wird die gesamte Projektplanung dokumentiert. Der Projektplan muss zu jeder Zeit den aktuell gültigen Stand des Projektes widerspiegeln. Es existieren Projektpläne für Organisation und Struktur, Durchführung und für Termine, Aufwände und Kosten des Projektes.

10.4.3 Planungstechniken

Planungstechniken sind methodische Werkzeuge für den Projektleiter. Sie beruhen auf erprobten Vorgehens- und Darstellungsweisen bei der Planung eines Projektes.

Versionierung und Revision von Dokumenten Während des Projektverlaufs entstehen zahlreiche Artefakte.
Definition: Ein Artefakt ist ein Dokument, Bericht oder ausführbares Modul, das erstellt, verändert oder verbraucht wird.
Voraussetzung für die Verwaltung der Artefakte, die für den Test und die Auslieferung konfiguriert werden müssen, ist die Versionskontrolle. Es ist unbedingt eine automatische Versionskontrolle und verteilung zu empfehlen (Beispiel: das Programm Concurrent Versions System (CVS).

CVS ist ein „version control system “ünd dient der Verwaltung und Archivierung der gesamten Entwicklungsgeschichte von Software-Projekten. Es bietet dabei die Möglichkeit, jeden Entwicklungsschritt zurückzuverfolgen und auf jede gesicherte Revision jeder zu dem Projekt gehörigen Datei zuzugreifen. Dabei wird die Entwicklung im Team unterstützt und es ist mehreren Mitarbeitern eines Projekts möglich, gleichzeitig dieselben Dateien zu bearbeiten.
Alle Projekte, welche unter der Kontrolle von CVS stehen, werden in einem sogenannten „repository“ gespeichert.
Hier werden im Ordner CVSROOT/ administrative Dateien gespeichert. Die Projekte an sich befinden sich typischerweise in eigenen Ordnern oder Unterordnern im „repository“ . Hier wird zu jeder Datei ein History-File gespeichert, aus dem sich jede Version rekonstruieren läßt. Außerdem werden darin ebenfalls alle Log-Einträge gespeichert und die Informationen darüber, wer wann welche Änderungen vorgenommen hat. Dateien werden nie direkt im „repository“ bearbeitet sondern jeder Mitarbeiter arbeitet immer nur mit Kopien der Dateien, welche er über ein „checkout“ anfordern muss und welche er nach der Bearbeitung wieder in das „repository “ legen muß.

Aufwandschätzung Die Schätzverfahren beruhen auf bei der Softwareentwicklung gemachten Erfahrungen. Sie teilen die Aufwandschätzung in zwei Schritten:

- Schätzung der Größe eines Arbeitspaketes
- Berechnung des Aufwands in MannMonaten aus der ermittelten Größenangabe

10.5 Projektdurchführung

Im Projekt enstehen viele Ergebnisse- Modelle und Quellcode. Dazu sind Daten zu erheben und immer wieder Soll-Ist-Vergleiche durchzuführen. Bei der Feststellung von Abweichungen sind Steuerungsmaßnahmen einzuleiten und die Pläne zu überarbeiten. Diese Phase schließt auch die Detailplanung für die nächste Iteration ein.

10.5.1 Kontrollvoraussetzungen

Durch die Kontrolle werden Informationen bereitgestellt, die entweder zu Steuerungsmaßnahmen oder zu Plankorrekturen führen und damit einen geregelten Projektablauf ermöglichen. Das rechtzeitige Erkennen von Abweichungen im Projektverlauf ist genauso wichtig wie die frühzeitige Reaktion auf Fehler bei der Softwareentwicklung. Die Daten für die Kontrolle werden mit Hilfe von Erhebungstechniken (Tätigkeitsberichte, Reviews) erhoben.

Tätigkeitsbericht Ein Tätigkeitsbericht ist, wie Abbildung 7 zeigt, ein computergestütztes Formular, das bereits die persönlichen Angaben des Mitarbeiters enthält:

- Angaben zur Person

Name Peter **Datum** 23.09

Arbeitszeiten

Arbeitspaket	Tätigkeit	Benötigte Zeit in Stunden								noch benötigt	Bemerkungen
EA05	Codierung	Mo 8	Di 5	Mi 5	Do 2	Fr 2	Sa 0	So 0	Summe 22	90	
EA05	Test	0	3	3	6	6	0	0	18	80	

Abbildung 7: Tätigkeitsbericht

Wie schätzt der Mitarbeiter die Projektqualität ein?	
Mitarbeitermotivation	
Checkliste	Checkliste für Mitarbeitermotivation
Bewertung	Sehr gut **gut** sehr schlecht
Bemerkung	Projektziele nicht immer klar ersichtlich

Abbildung 8: Review

- Die zugeteilten Aufgabengebiete in Form von Arbeitspaketen
- Eingeplante sowie zusätzliche Ressourcen

Das Formular wird durch den Mitarbeiter ergänzt, indem er seine Arbeitszeiten, gegliedert nach Wochentag, Arbeitspaket und Tätigkeit einträgt.

Reviews Die Reviews wie in Abbildung 8 stellen eine gute Möglichkeit für den Projektmanager dar, Abweichungen aufzudecken und Steuerungsmaßnahmen einzuteilen. Sie ergänzen die Tätigkeitsberichte, indem nicht nur Fakten, sondern auch Eindrücke gesammelt werden. Reviews werden im Team von mindestens 2 Leuten und dem Autor durchgeführt. Sie dauern nicht länger als 90 min. und sie finden für Dokumente und Quellcode statt, wenn eine Iteration oder ein Meilenstein erreicht ist.
Meilenstein bezeichnet ein Ereignis besonderer Bedeutung. Diese Ereignisse sind meist Unter- bzw. Zwischenziele eines Projektes.

10.5.2 Kontrollgrößen

Die Projektkontrolle ist in die folgenden Untergebiete unterteilt:

- Terminkontrolle
- Sachfortschrittskontrolle
- Qualitätssicherung

Termine Bei einer Terminkontrolle werden die Daten von den Tätigkeitsberichten der Mitarbeiter ausgewertet. Der Termin ist einer der wichtigsten Eckwerte, den man nicht gerne verschiebt. Das hängt mit der starken Konkurrenz und der damit verbundenen Wettbewerbsintensität zusammen. Die meisten Projekten weisen gerade da Probleme auf, da kaum ein Projekt pünktlich fertig wird.

Sachfortschritt Die Sachfortschrittskontrolle enthält die Produktfortschrittskontrolle und die Projektfortschrittskontrolle. Die Produktfortschrittskontrolle prüft den voranschreitenden Grad der Zielerreichung. Und die Projektfortschrittskontrolle überprüft, ob die Projektparameter wie Termine eingehalten werden.

Qualität Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht (Deutsche Industrie-Norm DIN 55350, Teil 11).
Die Qualitätskontrolle ist wichtig für den rechtzeitigen Abschluss von Software Projekten, für das Vermeiden von Fehlern, die oft erst in der Betriebsphase beim Kunden gefunden werden, und von Softwareabweichungen, die den Vorstellungen und Bedürfnissen des Kunden nicht entsprechen, und für eine vollständige und aktuelle Dokumentation.
Softwarequalität wird durch konstruktive und analytische Maßnahmen erreicht. Konstruktive Maßnahmen sollen das Entstehen von Fehlern und Qualitätsmängeln durch Vorgabe von geeigneten Methoden und Werkzeugen von vornherein verhindern. Analytische Maßnahmen dienen zur Erkennung und Lokalisierung von Mängeln und Fehlern.
Ein Beispiel für Software-Qualitätsmetriken ist die Zuverlässigkeit (Anzahl Fehler pro 1000 LOC (Lines of Code), die in einem bestimmten Zeitraum auftreten).

10.6 Projektabschluss

Diese Phase beschreibt das eindeutige Ende des Projekts, also das Produkt ist fertig und wird dem Auftraggeber übergeben. Das positive und eindeutige Ende des Projekts hat einen positiven Einfluss auf die Motivation der Mitarbeiter für zukünftige Projekte. Zur Produktübergabe werden Abnahmetests durchgeführt. Es sind Regelungen für eine weitere Betreuung des Produkts zu treffen.

10.6.1 Produktabnahme

Der wesentliche Punkt der Produktabnahme sind die Abnahmetests (Produkttest, Abschlusstest, Akzeptanztest, Pilottest). Im Übernahmeprotokoll werden die Schlussfolgerungen, zu denen die Ergebnisse der Tests und die im Projektvertrag formulierten Regelungen führen, dokumentiert. Falls die Abnahmetests Mängel aufzeigen sollen, müssen im Übernahmeprotokoll Regelungen getroffen werden, wie mit diesen Mängeln zu verfahren ist. Im besten Fall ist alles in Ordnung und das Produkt wird beim

Auftraggeber eingeführt.

10.7 Teamführung

Der Erfolg von Softwareprojekten ist im Wesentlichen abhängig von der Qualifikation und der Motivation der Mitarbeiter. Der Projektleiter hat einige Einflussmöglichkeiten auf Motivation bzw. Rahmenbedingungen des Projektes. Die Verhaltensbereitschaft oder auch Motivation eines Mitarbeiters lässt sich durch gute Führung positiv verändern. Das kann durch das Setzen von positiven Anreizen geschehen. Die positive Auswirkung dieser Anreize hängt auch von der Persönlichkeit ab.
„Motivation ist die Summe aller Motive eines Menschen, etwas zu tun. Sie bildet sich durch Erfahrungen und Erlebnisse positiver und negativer Art. Durch äußere Anreize werden diese Antriebe mehr oder weniger stark empfunden.“ [Buh04]
„Anreize sind Gegebenheiten, die wahrgenommen werden und Motive aktivieren“.

Die Auswahl der Anreize zur Erhöhung der Motivation hängt auch von den Menschentypen ab. Es werden 4 Menschentypen unterschieden:

- der rationale Mensch,
- der soziale Mensch,
- der selbstverwirklichende Mensch,
- der komplexe Mensch.

Der rationale Mensch ist durch monetäre Anreize zu motivieren. Er ist passiv und wird von der Organisation manipuliert und kontrolliert. Die Aufgabe der Organisation ist Neutralisieren und Kontrollieren irrationalen Verhaltens. Für das Management ergeben sich die Aufgaben: Planen, Organisieren, Kontrollieren. Er handelt ökonomisch rational.
Der soziale Mensch wird durch Erfüllung sozialer Bedürfnisse motiviert. Bei ihm beobachtet man stärkere Lenkung durch soziale Normen seiner Arbeitsgruppe (die Bedürfnisse nach Anerkennung, Zugehörigkeitsgefühl und Identität müssen befriedigt werden), nicht durch Anreize und Kontrolle des Vorgesetzten. Die Aufgabe des Managements ist der Aufbau und die Förderung von Gruppen.
Der selbstverwirklichende Mensch strebt nach Autonomie und bevorzugt Selbstmotivation und kontrolle. Die Delegation von Entscheidungen an diese Mitarbeiter ist wünschenswert
Der komplexe Mensch ist hochgradig lernfähig. Er erwirbt im Arbeitsprozess neue Motive und in unterschiedlichen Situationen werden unterschiedliche Motive bedeutsam. Er übernimmt Verantwortung, identifiziert sich mit Projekterfolg. Für diese Menschen ist das Projekt eine persönliche Angelegenheit. Sie leiten andere Mitarbeiter an. Solchen Menschen kann man Verantwortung übergeben.
Dauerhafte Motivation wird durch Befriedigung der drei wichtigsten Grundausrichtungen des menschlichen Verhaltens erreicht:

- Der Wille, gute Arbeit zu leisten
- Die Lust, sich in eine Gemeinschaft einzugliedern
- Der Wunsch nach Individualität

10.8 Zusammenfassung

Abgesehen von den ganzen Modellen sollte sich die PG nach einem eigenen Modell richten, also die PG sollte sich einigen, wie sie die große Aufgabe angeht, damit das Ziel erreicht wird. Sie soll das Ziel korrekt erfassen. Die PG sollte versuchen, die bei jeder Sitzung gestellten Aufgaben, d.h die gesetzten kleinen Ziele, möglichst effizient und rechtzeitig zu erledigen. Die Arbeit in Team ist zu empfehlen und nicht die Arbeit gegeneinander. Der Erfolg der PG liegt in den Händen der Leuten, die daran beteiligt sind. Jeder von der PG soll diese Aufgabe ernst nehmen und sich Mühe geben.

Teil III
Erstes Release

11 Anforderungen an das erste Release

Bei der Entwicklung einer kontrollierten Anfrageauswertung für relationale Datenbanken hat sich die Projektgruppe entschieden, die entsprechende Software in mehreren Iterationen gemäß Spiralmodell aufzubauen. Während der Seminarphase stellte sich heraus, dass eine Behandlung offener Anfragen, sofern diese sicher und domainunabhängig sind, durch eine Behandlung geschlossener Anfragen simuliert werden kann. So wird z.B. die offene Anfrage $krankheit(x, armbruch)$ durch eine Folge geschlossener Anfragen $krankheit(alfred, armbruch), krankheit(horst, armbruch) \ldots$ ersetzt. Deshalb soll in der ersten Iteration ein Programm entwickelt werden, dass zunächst nur geschlossene Anfragen für eine Teilmenge des Relationenkalküls behandelt. Dieses soll als Grundlage und Untersuchungsobjekt für die spätere Weiterentwicklung dienen. Dieses Dokument soll die Anforderungen an den Leistungsumfang des in dieser Phase zu entwickelnden Programms definieren. Folgendes soll das Programm leisten:

- Es gibt zunächst nur einen Benutzer, der seine Sicherheitspolitik und sein Vorwissen selber definieren und Anfragen an das System richten kann. Seine Sicherheitspolitik ist dem Benutzer somit selbstverständlich bekannt.

- Bezüglich der zulässigen Sprachen für Anfragen, Benutzerwissen und Sicherheitspolitik bestehen bestimmte Einschränkungen. So sind für Anfragen und für die aus potentiellen Geheimnissen bestehende Sicherheitspolitik prädikatenlogische Formeln in Pränexform ohne frei vorkommende Variablen und ohne Allquantoren erlaubt. Desweiteren dürfen Anfragen keine Negationen enthalten. Die Einschränkung auf Existenzquantoren liegt darin begründet, dass eine Mischung der Quantoren dazu führen würde, dass eine Implikation zwischen den Formeln unentscheidbar werden könnte. Genaueres hierzu ist dem Kapitel 5.4 zu entnehmen. Also darf nur eine Art Quantoren auftreten und existenzquantifizierte Anfragen sind hierbei die Üblicheren.
  Das Benutzerwissen soll durch prädikatenlogische Formeln in Pränexform ohne frei vorkommende Variablen dargestellt werden, wobei entweder kein Quantor oder genau eine Sorte Quantoren zugelassen werden. Hier sind Negationen explizit erlaubt, da sie insbesondere für Implikationen unabdingbar sind. In allen drei Fällen existieren desweiteren keine Funktionen. Diese Formeln sollen in einem möglichst generischen Format gehalten werden. Es soll eine Schnittstelle implementiert werden, die die uns interessierenden Informationen über diese Formeln zur Verfügung stellt. Sowohl das Benutzerwissen, als auch die Sicherheitspolitik sollen in einer hierfür geeigneten Datenstruktur persistent gehalten werden.

- Stellt der Benutzer eine gültige Anfrage (was vom Programm geprüft wird) an das System, so wird diese in ein passendes SQL-Statement transformiert, welches an das unter dem Programm liegende Oracle-DBMS abgesetzt wird.

- Als Ergebnis ist die Anfrage entweder wahr oder falsch. Der Einfachheit halber soll mit einem ablehnenden Zensor zensiert werden, was zur Folge hat, dass in

beiden Fällen geprüft werden muss, ob die entsprechende Antwort zusammen mit dem Benutzerwissen ein potentielles Geheimnis ableitbar machen würde. Dazu müssen für alle potentiellen Geheimnisse entsprechende Formeln erzeugt werden, die dann sukzessive einem Theorembeweiser zugeführt werden. Sobald eine gültige Implikation gefunden wurde, wird die Anfrage abgelehnt und der Benutzer erhält als Antwort *mum*. Ansonsten erhält der Benutzer die korrekte Antwort und diese wird seinem Wissen hinzugefügt. Optional können die Formeln auch noch durch einen sog. Optimierer behandelt werden, bevor sie durch den Theorembeweiser geprüft werden. Dieser könnte z.B. triviale Fälle erkennen oder die Formeln so aufbereiten, dass dem Theorembeweiser die Arbeit u.U. erleichtert wird.

- Als Theorembeweiser soll in dieser Iteration Prover9 zum Einsatz kommen (*http://www.cs.unm.edu/ mccune/prover9*). Dieser zeichnet sich vor allem durch seine einfache Handhabbarkeit und Flexibilität aus. Insbesondere kann beim Suchalgorithmus explizit eine Tiefen- oder Breitensuche erzwungen werden. Dies ist deshalb von besonderer Wichtigkeit, da es vorkommen kann, dass bei einer Tiefensuche unendlich tiefe Zweige durchlaufen werden und der Algorithmus niemals zu einem Ergebnis kommt. Mit einer Breitensuche kommt man in diesem Fall normalerweise dann doch noch zum Ziel, die leere Menge abzuleiten (sofern die Formel unerfüllbar ist). Die Schnittstelle soll aber so allgemein wie möglich gehalten werden, sodass möglichst problemlos auch eine andere Software zum Einsatz kommen kann.

Offene Punkte:

- Offen geblieben ist bislang, ob bereits in dieser Iteration beachtet werden soll, dass es sich bei den zu überprüfenden Implikationen um sog. *DB-Implikationen* (Kapitel 5.2) handelt. Da in diesem Durchlauf lediglich die Behandlung geschlossener Anfragen umgesetzt werden soll, hat dies hier allerdings keine besondere Relevanz.

Insgesamt soll diese Iteration den PG-Teilnehmern ein erstes Erfolgserlebnis verschaffen und eine Grundlage und Motivation für die weitere Entwicklung sein. Das Programm, das innerhalb dieser Iteration entsteht, soll als Ausgangspunkt für das gewünschte Endprodukt dienen. Gerade deshalb soll das Programm möglichst modular aufgebaut und alle Schnittstellen so generisch wie möglich gehalten werden.

12 Entwurf

In diesem Kapitel wird die Realisierung der Anforderungen an das erste Release anhand von UML-Diagrammen und auch eine textuelle Beschreibung entworfen, so dass auf dieser Basis mit der Implementierung des Programms angefangen werden könnte.

12.1 Eine Datenstruktur für Formeln

12.1.1 Einleitung

Für das gesamte Projekt soll unabhängig von der aktuellen Implementierung der kontrollierten Anfrageauswertung eine Formel in einer Datenstruktur gehalten werden, die eine generische Schnittstelle zur Verfügung stellt. Diese soll alle uns interessierenden Informationen über die jeweilige Formel zur Verfügung stellen und auch die nötigen Manipulationen zulassen.

12.1.2 Zulässige Formeln

Die in den Anforderungen für die erste Iteration festgelegte Klasse von Formeln für Anfragen, kann durch folgende Backus-Naur-Form beschrieben werden:
$N = \{ANF, EXP, R, VAR, KONST, RELSYMB, PARAM, JUNC,$
$KBUCHST, GBUCHST\}$
$T = \{\wedge, \vee, \neg, \exists, (,), ;, a, \ldots, z, A, \ldots, Z\}$
$S = ANF$

ANF	::=	$(\exists VAR)^* \quad EXP;$
EXP	::=	$R \mid (EXP \quad JUNC \quad EXP) \mid (\neg EXP)$
JUNC	::=	$\wedge \mid \vee$
R	::=	$RELSYMB(PARAM(,PARAM)^*)$
PARAM	::=	$KONST \mid VAR$
VAR	::=	$(GBUCHST)+$
KONST	::=	$(KBUCHST)+$
RELSYMB	::=	$(KBUCHST)+$
KBUCHST	::=	$a \mid \ldots \mid z$
GBUCHST	::=	$A \mid \ldots \mid Z$

Ferner müssen folgende kontextbezogene Nebenbedingungen gelten:

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 genügende Anfrage wäre z.B.:
$\exists X(((\neg beinbruch(gerd)) \vee krankheit(X, paranoia)) \wedge armbruch(X));$
Abbildung 9 zeigt den entsprechenden Syntaxbaum.

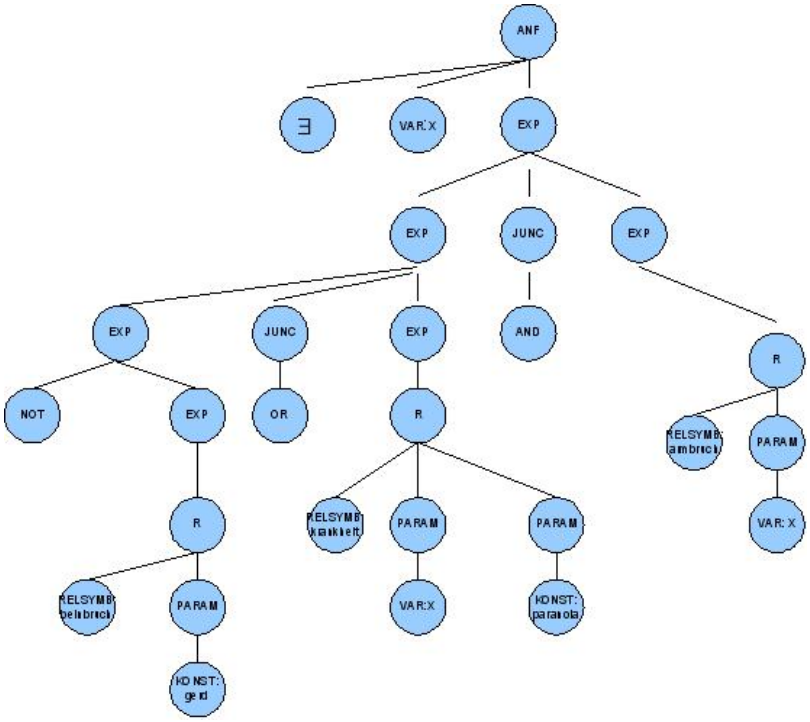

Abbildung 9: Syntaxbaum

12.1.3 JavaCC

Um vom Benutzer gestellte Anfragen auf ihre syntaktische Korrektheit hin zu überprüfen, soll ein von JavaCC automatisch generierter Scanner/Parser zu dieser BNF zum Einsatz kommen. JavaCC (Java Compiler Compiler) ist ein Parser-Generator für LL(k)-Parser. Die Grammatik wird bei JavaCC in einer BNF-ähnlichen Notation angegeben. Bei syntaktisch korrekten Anfragen wird durch das Zusatztool JJTree außerdem noch gleich der entsprechende Syntaxbaum übergeben. Hierbei handelt es sich um einen Verweis auf ein Objekt vom Typ SimpleNode. Dieses enthält ein Datum und eine Liste von Verweisen auf seine Kinder, die ebenfalls vom Typ SimpleNode sind. Die Datenstruktur SimpleNode, die die Knoten des Baumes repräsentiert, bietet allerdings für unsere Zwecke zu wenig Funktionalität. So ist das Manipulieren der Knoten hier nicht möglich. Deshalb soll eine eigene Datenstruktur konstruiert werden, die aus dem gegebenen Syntaxbaum aufgebaut wird und selbst wieder einen Baum repräsentiert. So kann man jede gewünschte Funktionalität selbst umsetzen. Diese Datenstruktur soll so in unsere Java-Datenstruktur für Formeln übernommen werden. Alle Informationen, die man über eine Formel haben möchte, kann man dann mittels einer Baumtraversierung einholen.

12.1.4 Schnittstelle

In gemeinsamer Arbeit wurden einige wichtige Informationen und Operationen herausgearbeitet, die eine Schnittstelle einer Fomel zur Verfügung stellen muss. Diese sind:

- Gib eine Liste der vorkommenden Variablensymbole zurück.
- Substituiere Variable durch ein neues gegebenes Variablensymbol.
- Gib den entsprechenden SQL-String zur einfachen Anfrageauswertung zurück.
- Gib die Formel als String aus.
- Negiere die Formel.

Weitere Operationen sind denkbar. Die Schnittstelle kann problemlos um benötigte Methoden erweitert werden.

12.2 Verwaltung von Schemainformationen

Im Laufe der Entwicklungsphase der ersten Iteration hat sich ein Bedarf für datenbankrelevante Informationen herausgestellt. Besonders werden Kenntnisse über die Relationenschemata für die syntaktische und lexikalische Analyse der von der Projektgruppe entwickelnden Anfragesprache benötigt. Für eine in der Anfrage vorkommende Relation soll also das Wissen gewonnen werden, ob die Relation in der Datenbank enthalten ist und wieviele Attribute sie besitzt. Dies dient als eine wichtige Voraussetzung für eine korrekte Übersetzung in ein SQL-Statement. Solche Schemainformationen sollen aus einer bestehenden Oracle Datenbank zuerst ausgelesen und danach programmintern effizient verwaltet werden.

12.2.1 Exkurs: Relationenschema

Ein Relationen-Schema in einer relationalen Datenbank ist durch den Namen der Relation und die Menge von Attributen mit ihren Wertebereichen gegeben. Die Wertebereiche sind einfache Datentypen, wie Integer, String, Date und Aufzählung. Als Beispiel ist ein Relationenschema
Armbruch(Name: String, jahr: integer, Art_des_Bruchs: String)
gegeben. Ein Relationenschema ist formal als $R(A_1 : D_1, \ldots, A_n : D_n)$ definiert. Wobei A_i Attribut und D_i Domäne zu A_i ist. Solche Informationen sollen in einer geeigneten Datenstruktur gespeichert und verwaltet werden.

12.2.2 Verwaltung der Datenbankinformationen

Die Verwaltung der Schemas kann mit Hilfe der JDBC-Metadaten-Funktionen durchgeführt werden.
Der JDBC-Client-Treiber unterstützt die folgenden Metadaten-Funktionen:

- getColumns - Liste aller Spalten in einer Tabelle.
- getColumnPrivileges - Liefert eine Beschreibung der Zugriffsrechte der Spalten einer Tabelle.
- getMetaData - liefert ein Objekt der Schnittstelle *DatabaseMetaData* zurück, das Informationen über die Struktur und die Funktionen der Datenbank bereithält. Dafür enthält die Schnittstelle über 100 verschiedene Methoden, die z.B. den Usernamen oder die maximale Anzahl der Spalten zurückgibt.
- getTypeInfo - Liefert Informationen über alle Standard-SQL-Typen, die von dieser Datenbank unterstützt werden.
- getTables - Zugriff auf Tabellen, z.B eines bestimmten Benutzers.
- getTableTypes - Liefert die Typen an Tabellen, die von der Datenbank unterstützt werden.

Interessant sind auch die Methoden der Schnittstelle ResultSetMetaData. Diese Schnittstelle ermöglicht es, auf ResultSet-spezifische Ergebnisse zuzugreifen. Beispiele für Methoden dieser Klasse sind:

- getColumnCount - Anzahl der Spalten in ResultSet.
- getColumnName - Spaltenname.
- getColumnType - Datentyp.

Somit sind alle für uns relevanten Funktionen von JDBC unterstützt. Weitere Informationen und Vorgehen sind den Aktivitäts- und Klassendiagramm zu entnehmen, die im Kapitel „UML-Modellierung “ nachgelesen werden können.

12.2.3 Schnittstelle Relationenschema

Die Schnittstelle Relationenschema soll die folgenden Funktionalitäten unterstützen:

- Ausgabe, ob eine bestimmte Relation in der Datenbank enthalten ist.
- Ausgabe, welche Attribute eine Relation besitzt.
- Ausgabe, welchen Typ die Attribute besitzen.
- Wenn eine Relation und die Nummer der Spalte bekannt sind, soll ermittelt werden, wie der Name der Spalte lautet.

Dies sind die wichtigsten Anforderungen an die Schnittstelle Relationenschema. Nach Bedarf kann die Klasse um weitere Methoden erweitert werden.

12.3 Erzeugung passender SQL-Statements zu Anfragen

Die Anforderungen an das System in dieser Iteration sehen vor, dass Anfragen in prädikatenlogischen Formeln in Pränexform ohne frei vorkommende Variablen, ohne Allquantoren und ohne Funktionen formuliert werden können. Negationen sind zwar in Anfragen nicht erlaubt, allerdings soll die Syntax diese bereits in dieser Iteration vorsehen. Damit nun der Wahrheitswert solcher Formeln innerhalb der dem Programm unterliegenden DB-Instanz bestimmt werden kann, sind hierzu passende SQL-Statements zu formulieren. Diese müssen auf eine bestimmte Art und Weise aus den gegebenen Formeln konstruiert und die Ergebnisrelationen entsprechend interpretiert werden. Um zu verstehen, wie dies vor sich geht, betrachtet man als erstes die kontextfreie Grammatik der Anfragesprache aus Kapitel 12.1.2 und die hieraus resultierenden Syntaxbäume.
Eine dieser BNF genügende Anfrage wäre z.B.
$\exists X((beinbruch(gerd) \vee krankheit(X, paranoia)) \wedge armbruch(X))$;
Abbildung 9 auf Seite 116 zeigt den entsprechenden Sytaxbaum.

12.3.1 Vorgehensweise

Legt man einen als Java-Datenstruktur gegebenen Syntaxbaum der oben gezeigten Art zugrunde, so wird das entsprechende SQL-Statement aus einer Tiefensuche durch den Baum heraus konstruiert. Da uns nur interessiert, ob eine gegebene Formel in einer DB-Instanz wahr ist und die Tupel, die diese Formel erfüllen, für uns nicht relevant sind, soll als Rahmen der folgende Ausdruck dienen: `SELECT DISTINCT 'c1' FROM(<zu konstruierender Ausdruck>);` , wobei $c1$ ein beliebiger Name ist, der nicht als Konstanten- bzw. Relationensymbol in der betrachteten Formel auftritt. Die Ergebnisrelation soll ein Tupel ($c1$) enthalten, wenn die Formel wahr ist, sonst leer sein.
Beim Konstruieren des benötigten Teilstrings macht man Gebrauch von der folgenden Tatsache: Eine Formel ist dann wahr, wenn es mindestens ein Tupel gibt, dass die gewünschten Eigenschaften erfüllt.
Die kleinsten betrachteten Teilformeln sind Literale, die die Form $R(e_1, \ldots, e_n)$ haben, wobei R das Relationensymbol und die e_i Terme (hier: Variablen- oder Konstantensymbole) bezeichnen. Für diese macht man eine SQL-Anfrage, indem man alle Attribute, die mit Konstanten belegt sind, in der WHERE-Klausel an diese Konstanten bindet. Mit Variablen belegte Attribute dürfen dabei einen beliebigen Wert haben. Diese SQL-Anfrage wird allerdings nicht für sich ausgeführt, sondern geht als Teilstring in die Gesamtanfrage ein. Für die Behandlung von Literalen ohne vorkommende Variablen gilt folgende Besonderheit: Da uns lediglich interessiert, ob eine (Teil-) Formel wahr ist oder nicht, benötigen wir nicht alle Tupel, sondern konstruieren wie bei der Gesamtanfrage lediglich eine Relation mit einem Attribut und einem Wert, sofern das Literal wahr ist. Dieser Wert ist ein neuer Name cn, der nicht in der Formel vorkommt. Dies spart bei der späteren Vereinigung von Tabellen Overhead und verhindert, dass dabei unerwünscht gleichnamige Attribute verglichen werden. Lediglich beim Auftreten von Variablen interessieren uns auch die Tupel, weshalb in

diesem Fall alle Tupel in einer Relation abgelegt werden müssen. Die Tupel werden für spätere Vereinigungen benötigt, da dabei nur passende Tupel ausgewählt werden sollen. Wenn in einem Literal Variablen auftreten, so werden über eine Projektion innerhalb der SQL-Anfrage nur die mit Variablen belegten Attribute ausgewählt und diese so benannt wie die Variablen. Das Ergebnis ist eine Relation mit sovielen Attributen, wie Variablen in dem entsprechenden Literal vorkommen. Dies macht es später möglich, Konjunktionen von Teilformeln, in denen die gleichen Variablen vorkommen, zu konstruieren.
Eine Konjunktion von Formeln wird nun über ein NATURAL JOIN der Teilformeln realisiert, hat also die Form

```
SELECT * FROM
(<Teilausdruck des linken Teilbaums>)
NATURAL JOIN
(<Teilausdruck des rechten Teilbaums>);
```

Bei zwei Relationen, die keine gemeinsamen Variablen haben, wird hier das Kreuzprodukt erzeugt, sofern beide Ergebnisrelationen nicht leer sind, ansonsten ist die Ergebnisrelation leer. Kommen im linken und im rechten Teilausdruck gleiche Variablen vor, so werden durch die Benennung der Attribute nach den Variablennamen, automatisch die passenden Tupel für die Vereinigung ausgesucht.
Eine Disjunktion von Formeln wird über ein NATURAL FULL OUTER JOIN realisiert, hat also die Form

```
SELECT * FROM
(<Teilausdruck des linken Teilbaums>)
NATURAL FULL OUTER JOIN
(<Teilausdruck des rechten Teilausdrucks>);
```

Hierbei wird eine Relation erzeugt, die auch dann Tupel enthält, wenn eine der beteilgten Ausgangsrelationen leer war.

12.3.2 Beispiel

Gegeben sei die obige Formel
$\exists X((beinbruch(gerd) \vee krankheit(X, paranoia)) \wedge armbruch(X))$;
Der gesamte SQL-Ausdruck wird über eine Tiefensuche erzeugt. Für die neuen Konstanten in den Teilausdrücken mit `SELECT DISTINCT ...` werden fortlaufend die Symbole c1,...,cn benutzt. Da für Relationensymbole und Konstanten in einer Anfrage nur Kleinbuchstaben und keine Zahlen erlaubt sind, sind Kollisionen diesbezüglich ausgeschlossen. Der Algorithmus führt nun die folgenden Schritte aus:

1. Der erste Knoten ist das Startsymbol. Es wird das Gerüst `SELECT DISTINCT 'c1' FROM (<Ausdruck>);` aufgebaut. Für <Ausdruck> wird die Tiefensuche fortgesetzt.

2. Der nächste Knoten enthält den Operator $\wedge$. Also wird das Gerüst `(SELECT * FROM <Ausdruck1> NATURAL JOIN <Ausdruck2>)` aufgebaut. Für <Ausdruck1> wird die Tiefensuche fortgesetzt.

3. Der nächste Knoten enthält den Operator $\vee$. Also wird das Gerüst `(SELECT * FROM <Ausdruck3> NATURAL FULL OUTER JOIN <Ausdruck4>)` aufgebaut. Für <Ausdruck3> wird die Tiefensuche fortgesetzt.

4. Der nächste Knoten enthält das Relationensymbol *beinbruch*, in dem keine Variablen vorkommen. Es wird der Ausdruck `(SELECT DISTINCT 'c2' FROM (SELECT * FROM beinbruch WHERE name='gerd'))` zurückgegeben. Danach kehrt der Algorithmus zum $\vee$-Knoten aus 3. zurück.

5. Für <Ausdruck4> wird die Tiefensuche fortgesetzt.

6. Der nächste Knoten enthält das Relationensymbol *krankheit* mit der Variablen X. Es wird der Ausdruck `(SELECT name AS X FROM krankheit WHERE krankheit='paranoia')` zurückgegeben. Danach kehrt der Algorithmus zum $\vee$-Knoten aus 3. zurück.

7. Der fertige Teilausdruck
   `(SELECT * FROM (SELECT DISTINCT 'c2' FROM (SELECT * FROM beinbruch WHERE name='gerd')) NATURAL FULL OUTER JOIN (SELECT name AS X FROM krankheit WHERE krankheit='paranoia'))`
   wird zurückgegeben. Danach kehrt der Algorithmus zum $\wedge$-Knoten aus 2. zurück.

8. Für <Ausdruck2> wird die Tiefensuche fortgesetzt.

9. Der nächste Knoten enthält das Relationensymbol *armbruch* mit der Variablen X. Es wird der Ausdruck `(SELECT name AS X FROM armbruch)` zurückgegeben. Danach kehrt der Algorithmus zum $\wedge$-Knoten aus 2. zurück.

10. Der fertige Teilausdruck
    `(SELECT * FROM (SELECT * FROM (SELECT DISTINCT 'c2' FROM (SELECT * FROM beinbruch WHERE name='gerd')) NATURAL FULL OUTER JOIN (SELECT name AS X FROM krankheit WHERE krankheit='paranoia')) NATURAL JOIN (SELECT name AS X FROM armbruch))`
    wird zurückgegeben. Danach kehrt der Algorithmus zum Startsymbol aus 1. zurück.

11. Der fertige Gesamtausdruck
    `SELECT DISTINCT 'c1' FROM((SELECT * FROM (SELECT * FROM (SELECT DISTINCT 'c2' FROM (SELECT * FROM beinbruch WHERE name='gerd')) NATURAL FULL OUTER JOIN (SELECT name AS X FROM krankheit WHERE krankheit='paranoia')) NATURAL JOIN (SELECT name AS X FROM armbruch)));`
    wird zurückgegeben.

12.4 Die Schnittstelle zum Theorembeweiser

12.4.1 Einleitung

Die Schnittstelle unseres Programms zu einem Theorembeweiser (für die erste Iteration wurde Prover9 ausgewählt) besteht aus der Interface-Klasse „TpInterface“.
Diese Klasse besitzt eine Methode *isImplied* mit den Übergabeparametern *log* (Benutzerwissen als Datentyp LinkedList), *query* (Anfrage des Benutzers als Datentyp Formula) und *pot_sec* (potentielle Geheimnisse als Datentyp LinkedList). Mit dieser Methode wird überprüft ob alle übergebenen potentiellen Geheimnisse durch das Benutzerwissen und die Anfrage impliziert werden. Die Methode *isImplied* hat den Rückgabetyp boolean, d.h. wenn die Implikation bewiesen werden kann, so erhält man den Rückgabewert *true* ansonsten *false*.

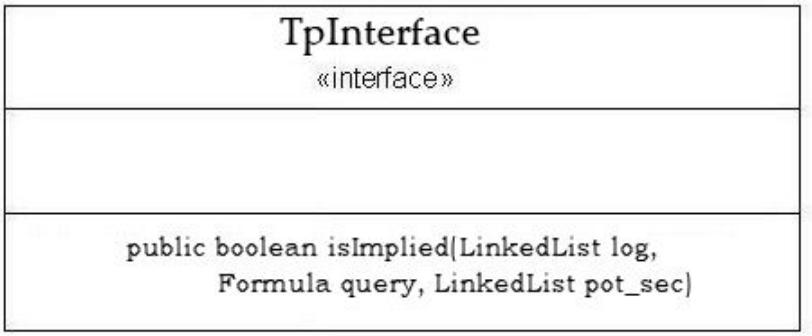

Abbildung 10: Interface-Klasse des Theorembeweisers

12.4.2 Inputdatei

Intern überführt die Methode *isImplied* die Übergabeparameter in eine Inputdatei für Prover9, welche im folgenden Format vorliegen muss:

formulas(sos).

„Benutzervorwissen (log) wird hier Zeile für Zeile eingefügt.“
„In die letzte Zeile wird die Anfrage geschrieben.“

```
man(horst).
god(hugo).

all x (man(x) -> mortal(x)).
all x (god(x) -> -mortal(x)).
```

„Anfrage sei:“

```
man(george).
```

end_of_list.

formulas(goals).

„Hier werden einzeln die übergebenen potentiellen Geheimnisse eingefügt und anschließend einzeln getestet.“

```
mortal(george).
```

end_of_list.

Kurze Erläuterung zur Syntax:
Die Formeln zwischen *formulas(sos).* und dem ersten Auftreten von *end_of_list.* sind konjunktiv verknüpfte Terme, die unsere Prämisse sind.
Die Formel zwischen *formulas(goals).* und dem zweiten *end_of_list.* ist die Konklusion, die überprüft wird.

12.5 UML-Modellierung

12.5.1 Das Anwendungsfalldiagramm

Aus den Anforderungen heraus, ergeben sich unmittelbar die relevanten Anwendungsfälle. Diese sind Abbildung 11 zu entnehmen. Hierbei ist

- *CQE* (*kontrollierte Anfrageauswertung*): Die Hauptprozedur. Es wird eine Anfrage ausgewertet.
- *addToLog*: Dem Benutzerwissen eine Formel hinzufügen.
- *addToPot_sec*: Den potentiellen Geheimnissen eine Formel hinzufügen.
- *deleteFromLog*: Eine Formel aus dem Benutzerwissen entfernen.
- *deleteFromPot_sec*: Eine Formel aus den potentiellen Geheimnissen entfernen.

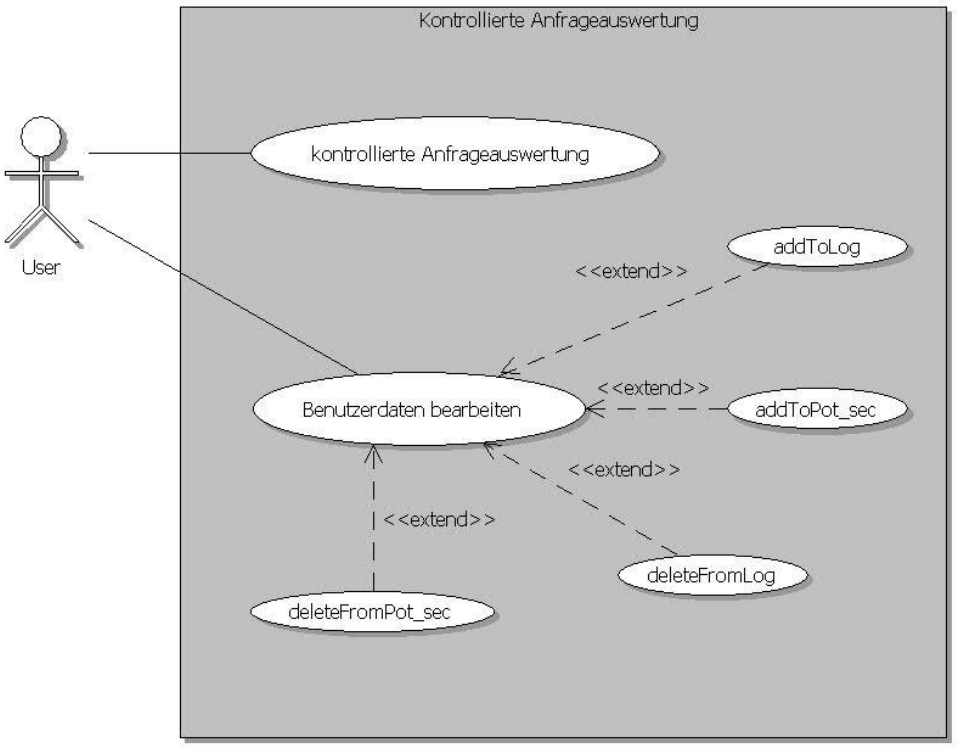

Abbildung 11: Anwendungsfälle in der ersten Iteration

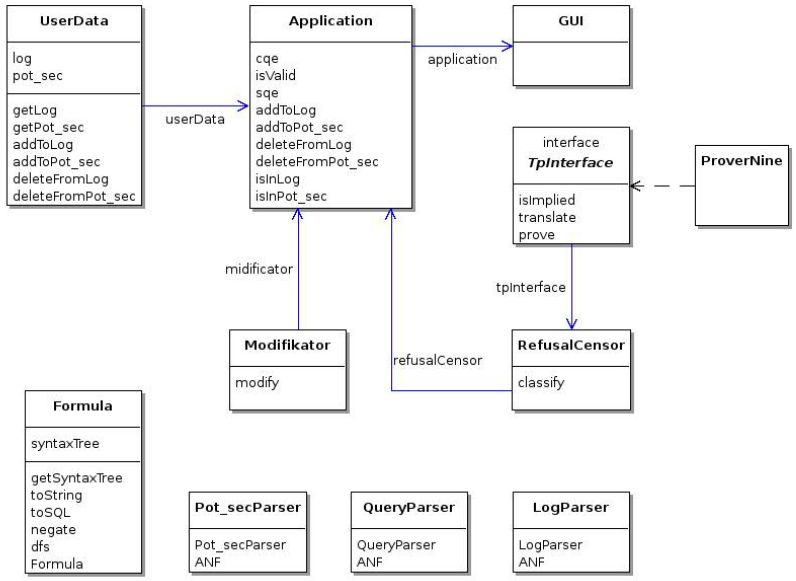

Abbildung 12: Klassendiagramm in der ersten Iteration

12.5.2 Klassendiagramm

Allgemeine Beschreibung des Klassendiagramms:
Um einen Einblick in die Struktur des Klassendiagramms zu geben, ist hier eine kurze Beschreibung der verwendeten Techniken.
Anfragen werden mit dem von JavaCC(Java Compiler Compiler) erzeugten Parser auf syntaktische Korrektheit überprüft und als ein Syntaxbaum zurückgegeben.
Anfragen werden anhand der vorhandenen Datenbank ausgewertet. Als Datenbanksystem wird Oracle verwendet und entsprechend werden die Anfragen als SQL-Statements dargestellt.
Die bei der Auswertung entstehenden Implikationen werden mit dem Theorembeweiser Prover9 gelöst. Damit das Implikationsproblem entscheidbar wird, werden die Sprachen für Log (Benutzerwissen), Pot_Sec (potentielle Geheimnisse) und Queries (Anfragen) so festgelegt, dass die dadurch entstehenden Implikationsprobleme in der Bernays-Schönfinkel-Klasse sind.
Das Klassendiagramm:

- **Klasse Formula**
  Diese Klasse stellt eine generische Schnittstelle für Formeln zur Verfügung.
  - **getSyntaxTree(): TreeNode** Liefert den zugehörigen Syntaxbaum zurück.

  - **toSQL(): String** Liefert die entsprechende SQL-Anfrage.
  - **negate():** Negiert die Formel.
  - **dfs(syntaxbaum: TreeNode, counter: int): String** Generiert über eine Tiefensuche aus dem Syntaxbaum den entsprechende SQL-Befehl und liefert ihn zurück.

- **Klasse GUI**
  Diese Klasse startet das Hauptprogramm und aktualisiert die Benutzerdaten ggf.
  Die Klasse besitzt nur die beiden Methoden *main* und *refresh*. Die Methode *main* startet das Hauptprogramm und die Methode *refresh* aktualisiert die Benutzeroberfläche. (z.B. Beim Einfügen von log oder pot_sec)

- **Klasse Application**
  Die Klasse stellt alle vorhandenen Methoden bereit, die für eine Anwendung relevant sind.

  - **cqe(query: String): String** Die Methode bekommt als Übergabeparameter den vom Benutzer eingegebenen String (Anfrage) und gibt einen String (die Antwort auf die Anfrage) zurück, siehe 12.5.3.
  - **isValid(syntaxbaum: TreeNode): boolean** Die Methode überprüft den Syntaxbaum (TreeNode) auf seine Gültigkeit bzgl der semantischen Nebenbedingungen.
  - **sqe(query: String): boolean** Die Methode führt die einfache Anfrageauswertung aus.
  - **addToLog(expr: String):** Die Methode fügt den Ausdruck (expr) zu dem Benutzerwissen hinzu, siehe 12.5.4.
  - **addToPot_sec(expr: String):** Die Methode fügt den Ausdruck zu der Menge der potentiellen Geheimnissen hinzu, siehe 12.5.4.
  - **deleteFromLog(index: Integer):** Die Methode entfernt Log (Index) aus dem Benutzerwissen, siehe 12.5.4.
  - **deleteFromPot_sec(index: Integer):** Die Methode entfernt Pot_sec (Index) aus der Menge der potentiellen Geheimnisse, siehe 12.5.4.
  - **isInLog(Syntaxbaum: TreeNode): boolean** Die Methode überprüft, ob eine Formel (Syntaxbaum) im Benutzerwissen schon vorhanden ist.
  - **isInPot_sec(Syntaxbaum: Treenode): boolean** Die Methode überprüft, ob eine Formel (Syntaxbaum) in der Menge der potentiellen Geheimnisse schon vorhanden ist.
    Die beiden letzten Methoden werden hauptsächlich innerhalb der Methoden *addToLog* und *addToPot_sec* benutzt, wo der entsprechende Syntaxbaum von der Formel (String) erzeugt wurde. Aus diesem Grund wird bei diesen Methoden ein Syntaxbaum als Eingabeparameter übergeben.

- **Klasse UserData**
  Diese Klasse verwaltet Log und Pot_Sec.

- **getLog(): LinkedList<Formula>** Liefert das Benutzerwissen als eine Liste.
- **getPot_sec(): LinkedList<Formula>** Liefert die potentiellen Geheimnisse als eine Liste.
- **addToLog(item: Formula):** Die Methode fügt den Ausdruck (item) zu dem Benutzerwissen hinzu.
- **addToPot_sec(item: Formula):** Die Methode fügt den Ausdruck (item) zu der Menge der potentiellen Geheimnissen hinzu.
- **deleteFromLog(index: Integer):** Die Methode entfernt Log (Index) aus dem Benutzerwissen.
- **deleteFromPot_sec(index: Integer):** Die Methode entfernt Pot_sec (Index) aus der Menge der potentiellen Geheimnisse.

Es ist zu beachten, dass die Methoden *addToLog*, *addToPot_sec*, *deletFromLog* und *deleteFromPot_sec* in der Klasse *UserData* unterscheiden sich von den gleichnamigen Methoden in der Klasse *Application* in dem, dass sie keine syntaktische und semantische Korrektheit der Formeln überprüfen. Sie fügen (bzw. entfernen) die Eingabe in einer Java-Datenstruktur ein.

- **Klasse RefusalCensor**
  Diese Klasse klassifiziert Implikationsergebnisse und bestimmt in Abhängigkeit von denen die Antwort auf die Anfragen, siehe 12.6.

- **Klasse Modificator**
  Diese Klasse bringt die Antwort auf die Anfrage in letztendliche Form.

- **Klasse QueryParser**
  Diese Klasse ist für die semantische Überprüfung einer Anfrage und die anschließende Ausgabe in Form von einem Baum zuständig, siehe 12.1.3.

- **Klasse LogParser**
  Diese Klasse ist für die semantische Überprüfung eines Logs und die anschließende Ausgabe in Form von einem Baum zuständig, siehe 12.1.3.

- **Klasse Pot_secParser**
  Diese Klasse ist für die semantische Überprüfung eines Pot_sec und die anschließende Ausgabe in Form von einem Baum zuständig, siehe 12.1.3.

  Die Methode **ANF** in Parserklassen dient dazu, um einen Knoten vom Typ *SimpleNode* zu erzeugen und zurückzuliefern. Der Knoten wird beim Erzeugen eines Syntaxbaums benutzt.

- **Interface TPInterface**
  Dieses Interface ist für die Schnittstelle zum Testen von Implikationen.

  - **isImplied(log: LinkedList, query: Formula, pot_sec: LinkedList): boolean** Erhält als Übergabeparameter das Benutzerwissen (log), die Benutzeranfrage (query) und die Menge der potentiellen Geheimnisse (pot_sec)

und liefert den Wertt *true* aus, wenn ein potentielles Geheimnis aus der Anfrage und Benutzerwissen impliziert wird.

- **translate(log: LinkedList, query: Formula): String** Überführt das Benutzerwissen (log) und die Benutzeranfrage (query) in eine passende Eingabesprache des Theorembeweisers.
- **prove(): boolean** Testet jeweils eine Implikation auf ihren Wahrheitsgehalt.

- **Klasse ProverNine**
  Diese Klasse ist eine Unterklasse von TpInterface und implementiert den TheoremBeweiser ProverNine.

12.5.3 Anwendungsfall kontrollierte Anfrageauswertung

Dieser Abschnitt soll den internen Programmablauf der kontrollierten Anfrageauswerung beschreiben. Es soll vor allem gezeigt werden, wie die im Klassendiagramm festgehaltenen Komponenten interagieren, um die gewünschte Funktionalität umzusetzen. Hierfür werden die entsprechenden, in der Entwurfsphase erstellten, Aktivitäts- und Sequenzdiagramme herangezogen.

Abbildung 41 zeigt den Ablauf der kontrollierten Anfrageauswertung dargestellt als Aktivitätsdiagramm. Nachdem der Benutzer seine Eingabe gemacht hat, wird zunächst geprüft, ob es sich dabei um eine gültige Anfrage im Sinne der hierfür definierten Grammatik handelt. Liegt eine syntaktisch nicht korrekte Anfrage vor, so wird abgebrochen. Ansonsten muss die Anfrage darauf geprüft werden, ob sie die von uns festgelegten semantischen Nebenbedingungen einhält. Wenn dies nicht der Fall ist, so wird auch hier abgebrochen. Sind auch die semantischen Nebenbedingungen erfüllt, so muss die Eingabe in eine SQL-Anfrage übersetzt werden, für die das unterliegende DBMS – entsprechend interpretiert – den Wahrheitswert der Anfrage in der gegebenen DB-Instanz liefert. Die korrekte Antwort wird aber zunächst noch zurückgehalten und erst einmal die Anfrage auf ihre Gefährlichkeit hin überprüft. Hierzu werden sowohl die Anfrage selbst, als auch das Wissen des Benutzers und seine Sicherheitspolitik benötigt. Diese werden syntaktisch so umgeformt, dass der angebundene Theorembeweiser sie versteht. Dies wird mittels einer Tiefensuche durch den Syntaxbaum realisiert. Dabei müssen im Prinzip nur Negation, Konjunktion und Disjunktion anders dargestellt werden. Die hierdurch erzeugte Eingabe wird dem Theorembeweiser zugeführt und der Test $\{eval^*(\Phi)(db)\} \cup log \models \Psi$ für ein $\Psi \in pot\_sec$ durchgeführt. Dieser Test wird für alle $\Psi \in pot\_sec$ wiederholt. Danach wird die Anfrage negiert und die Schleife nochmals für $\{\neg eval^*(\Phi)(db)\} \cup log \models \Psi$ ausgeführt. Sobald sich eine Implikation als wahr herausstellt, kann die Prozedur abgebrochen werden und die Antwort an den Benutzer lautet mum.
Wurden alle Ψ abgearbeitet und keine Gefahren erkannt, so wird die korrekte Antwort dem Benutzerwissen hinzugefügt und ausgegeben.

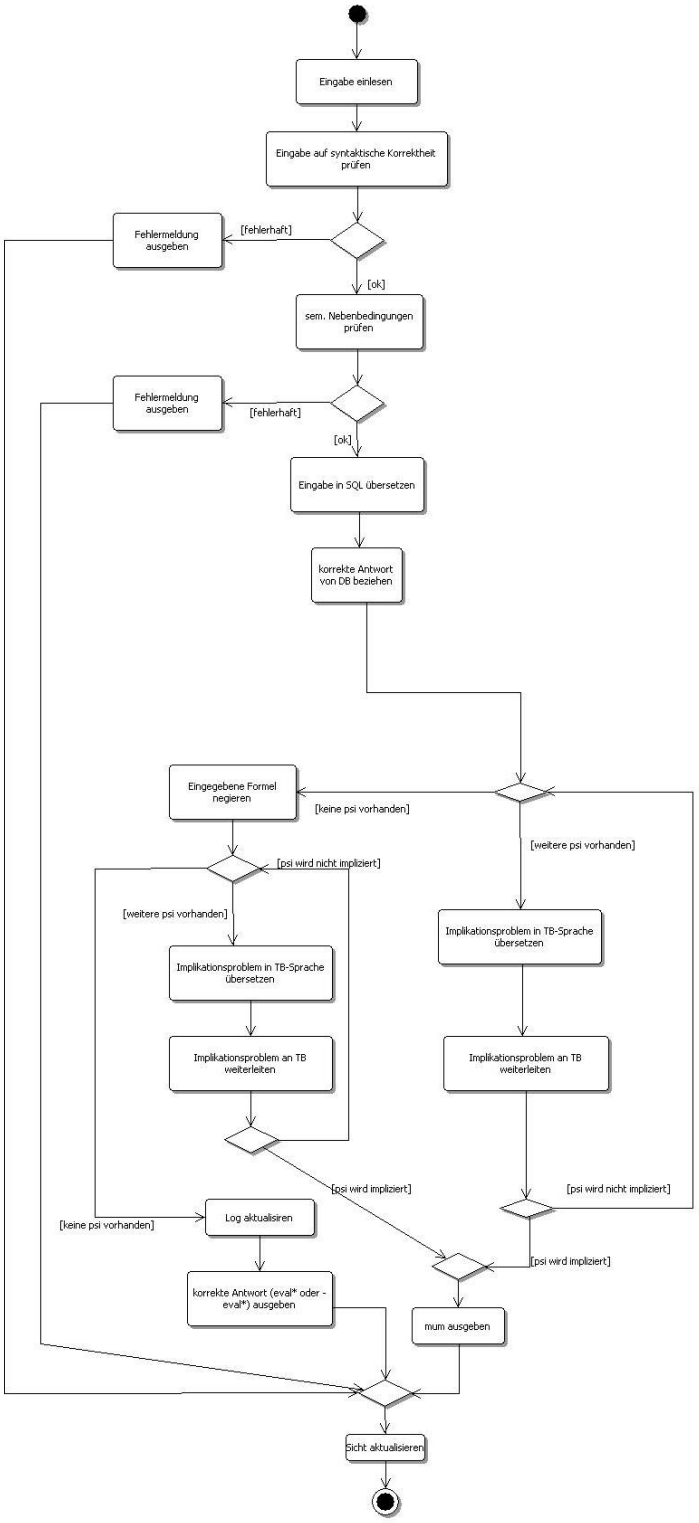

Abbildung 13: Aktivitätsdiagramm der kontrollierten Anfrageauswertung

Das Sequenzdiagramm aus Abbildung 14 zeigt einen möglichen Ablauf der kontrollierten Anfrageauswertung im Detail. Hierbei wird sichtbar, welche Methoden von welchen Klassen aus aufgerufen werden und welche Informationen dabei fließen. Das Sequenzdiagramm zeigt genau einen möglichen Ablauf des Aktivitätsdiagramms. Hierbei gelten folgende Voraussetzungen:

- Die Anfrage ist syntaktisch korrekt und erfüllt alle semantischen Nebenbedingungen.
- Die Anfrage ist wahr in db.
- Es gibt nur ein potentielles Geheimnis.
- Die Anfrage ist ungefährlich.

Wenn diese Voraussetzungen gelten, ergibt sich genau der dargestellte Ablauf.
Die Benutzeroberfläche ruft die Methode cqe im Objekt Application auf und übergibt ihr den vom Benutzer eingegeben String. Innerhalb der Methode cqe wird als nächstes unter Übergabe des eingegebenen Strings der QueryParser, ein Parser für die Eingabe, initialisiert. Durch den Aufruf der Methode ANF (die das Startsymbol ist) wird der Parser gestartet. Da die Eingabe eine gültige Anfrage war, wird ein Objekt vom Typ SimpleNode zurückgegeben, welches die Wurzel des entsprechenden Syntaxbaums als Java-Datenstruktur darstellt. Nun wird ein Objekt vom Typ TreeNode erzeugt. Der Konstruktor erhält als Parameter die vom Parser erzeugte SimpleNode Datenstruktur, aus welcher ein neuer Syntaxbaum, deren Knoten vom Typ TreeNode sind erzeugt. TreeNode ist eine von uns selbst implementierte Datenstruktur für einen Syntaxbaum und im Weiteren wird nur noch mit dieser Datenstruktur gearbeitet. Sie ist für unsere Zwecke besser geeignet, da sie auch Methoden zum Manipulieren des Syntaxbaumes enthält. Der Syntaxbaum wird nun mittels der Methode isValid auf seine Gültigkeit bezüglich der semantischen Nebenbedingungen geprüft. Nach Voraussetzung wird hier true zurückgegeben. Die Methode classify der Klasse RefusalCensor soll eine Klassifizierung der Anfrage zurückgeben. Der Rückgabewert ist vom selbstdefinierten Typ CType und soll für den modellierten Ablehnungszensor einer der Werte 'UNCRITICAL' oder 'MUM' sein. Um die Klassifizierung vorzunehmen, muss der Zensor prüfen, ob $\{eval^*(\Phi)(db)\} \cup log \models \Psi$ oder $\{\neg eval^*(\Phi)(db)\} \cup log \models \Psi$ für ein $\Psi \in pot\_sec$ gilt. Hierzu ruft dieser die Methode isImplied der Klasse ProverNine auf. Hier wird mittels der Methode translate die Formel, das Benutzerwissen und die potentiellen Geheimnisse in die Syntax des Theorembeweisers gebracht. Anschließend werden mit prove in einer Schleife alle vorhanden $\Psi \in pot\_sec$ durchlaufen und die Anfrage, das Benutzerwissen und das aktuelle Ψ als Eingaben für den Theorembeweiser in einer Datei gespeichert. Danach wird der Theorembeweiser als ein externer Prozess aufgerufen, der die zuvor erzeugte Datei als Eingabe benutzt. Das Ergebnis ist, dass das potentielle Geheimnis nicht ableitbar ist. Nun wird die gesamte Prozedur nocheinmal mit der negierten Anfrageformel ausgeführt. Auch hier wird das potentielle Geheimnis nicht impliziert. Das Ergebnis wird einem Modifikator mitgeteilt, der die entsprechden Handlungsvorschrift ausgibt. In diesem Fall „UNCRITICAL“, weshalb die Formel dem Benutzerwissen hinzugefügt und ausgegeben wird.

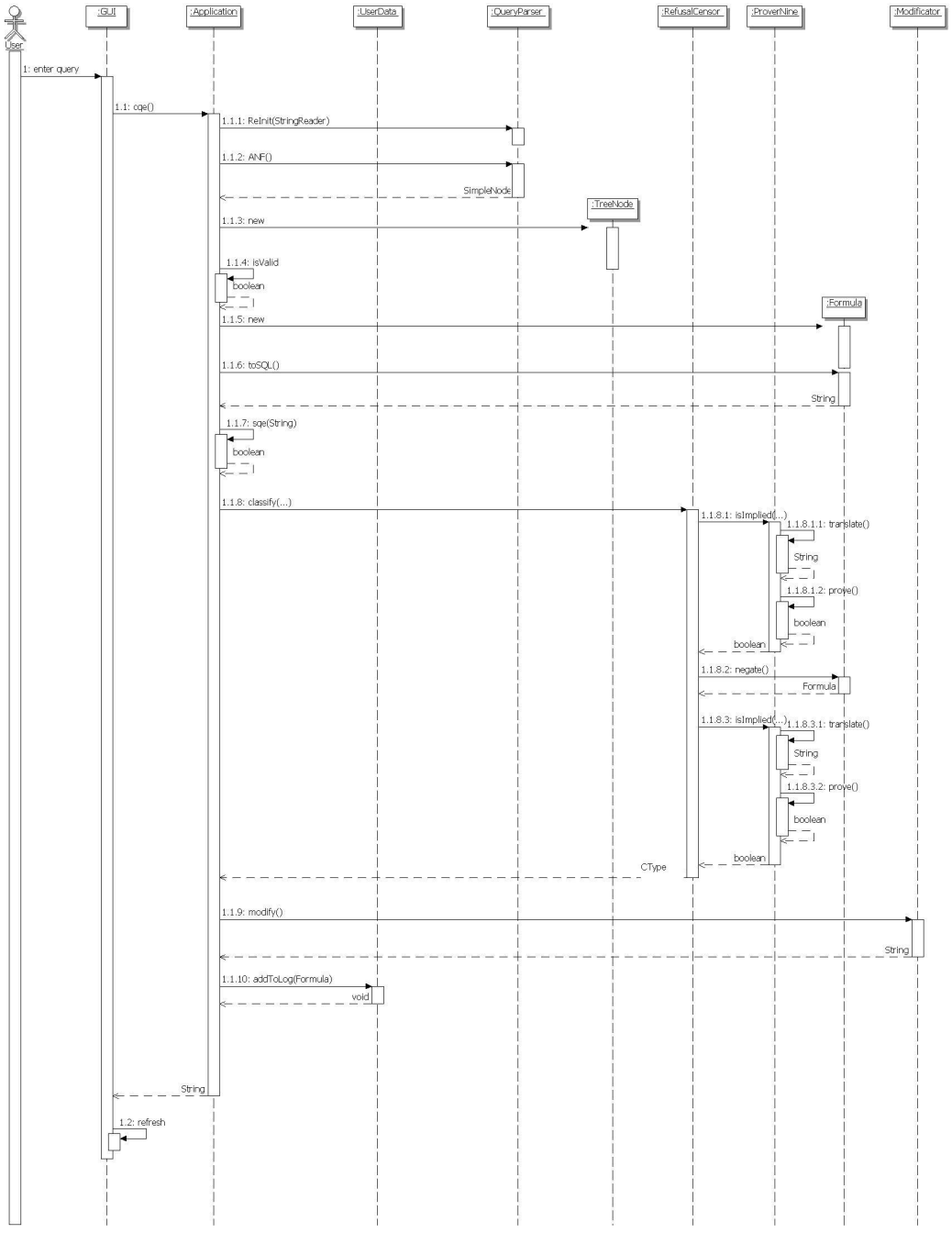

Abbildung 14: Sequenzdiagramm der kontrollierten Anfrageauswertung in der ersten Iteration

12.5.4 Anwendungsfall Benutzerdaten bearbeiten

Aktivitätsdiagramm AddToPot_sec Um ein neues potentielles Geheimnis in die pot_sec-Menge einzufügen, sollte zuerst die Eingabe eingelesen werden, die sowohl auf syntaktische Korrektheit als auch semantische Nebenbedingungen geprüft wird. In dem Fall, dass die Bedingungen nicht erfüllt sind, wird eine Fehlermeldung ausgegeben. Sonst wird die pot_sec-Menge nach einem eventuell mit der Eingabe syntaktisch gleichen potentiellen Geheimnis Ψ durchsucht. Falls das Element bereits vorhanden ist, wird eine Fehlermeldung ausgegeben. Sonst wird die Eingabe in die pot_sec-Menge hinzugefügt und die Menge wird aufgelistet.

Die Abbildung 15 stellt das Aktivitätsdiagramm von AddToPot_sec dar.

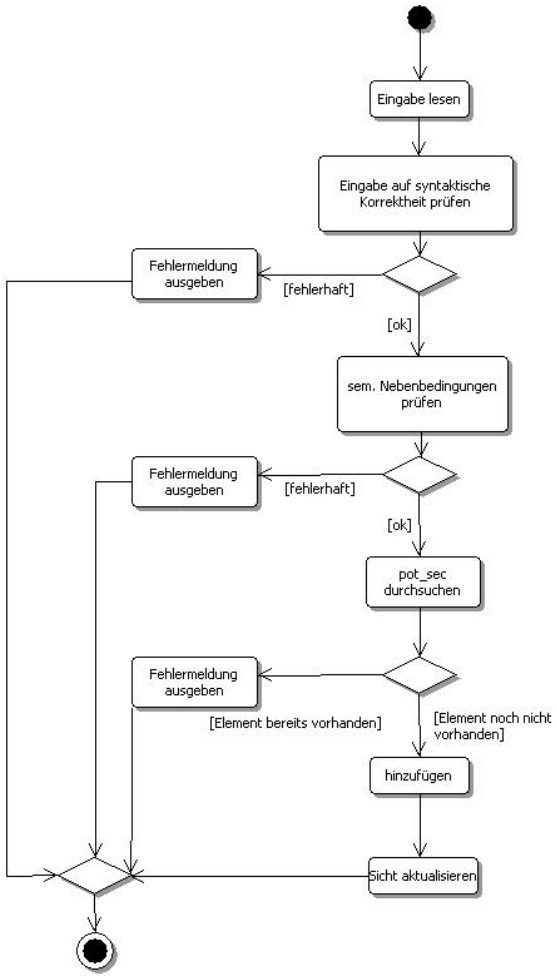

Abbildung 15: Aktivitätsdiagramm AddToPotsec

Sequenzdiagramm AddToPot_sec Der Benutzer ruft die Methode zum Hinzufügen des potentiellen Geheimnisses über die Oberfläche auf. Die Methode wird an das Application-Objekt weitergeleitet.
Application erzeugt ein neues Pot_secParser-Objekt, um anhand der definierten Grammatik einen Syntaxbaum vom Typ jjTree zu erzeugen. Der Baum wird benutzt um ein Formula-Objekt zu erzeugen, nachdem seine syntaktische Korrektheit geprüft ist. Die pot_sec-Menge wird nach einem eventuell mit der Eingabe gleichen Element durchgesucht, um doppeltes Vorkommen von Elementen zu vermeiden.
Es werden die Implikationen erzeugt und durch den Theorembeweiser überprüft, ob die Implikationen für die neue Eingabe wahr sind.
Am Ende wird die Eingabe hinzugefügt und die Sicht wird aktuallisiert.

Die Abbildung 16 auf Seite 136 stellt das Sequenzdiagramm von AddToPot_sec dar.

Aktivitätsdiagramm DeleteFromPot_sec Da ein potentielles Geheimnis nur dann gelöscht werden kann, wenn es bereits existiert, wählt der Benutzer das zu löschende potentielle Gehemnis aus der aufgelisteten pot_sec-Menge.
Nach dem Löschen wird den Sicht aktualisiert und angezeigt.

Die Abbildung 17 auf Seite 137 stellt das Aktivitätsdiagramm von DeleteFromPot_sec dar.

Sequenzdiagramm DeleteFromPot_sec Der Benutzer ruft die Methode zum Löschen des potentiellen Geheimnisses über die Oberfläche auf und wählt das zu löschende potentielle Geheimnis aus. Es wird an die Application weiter geleitet und anschließend gelöscht.
Am Ende wird die Sicht aktualisiert.

Die Abbildung 18 auf Seite 137 stellt das Sequenzdiagramm von DeleteFromPot_sec dar.

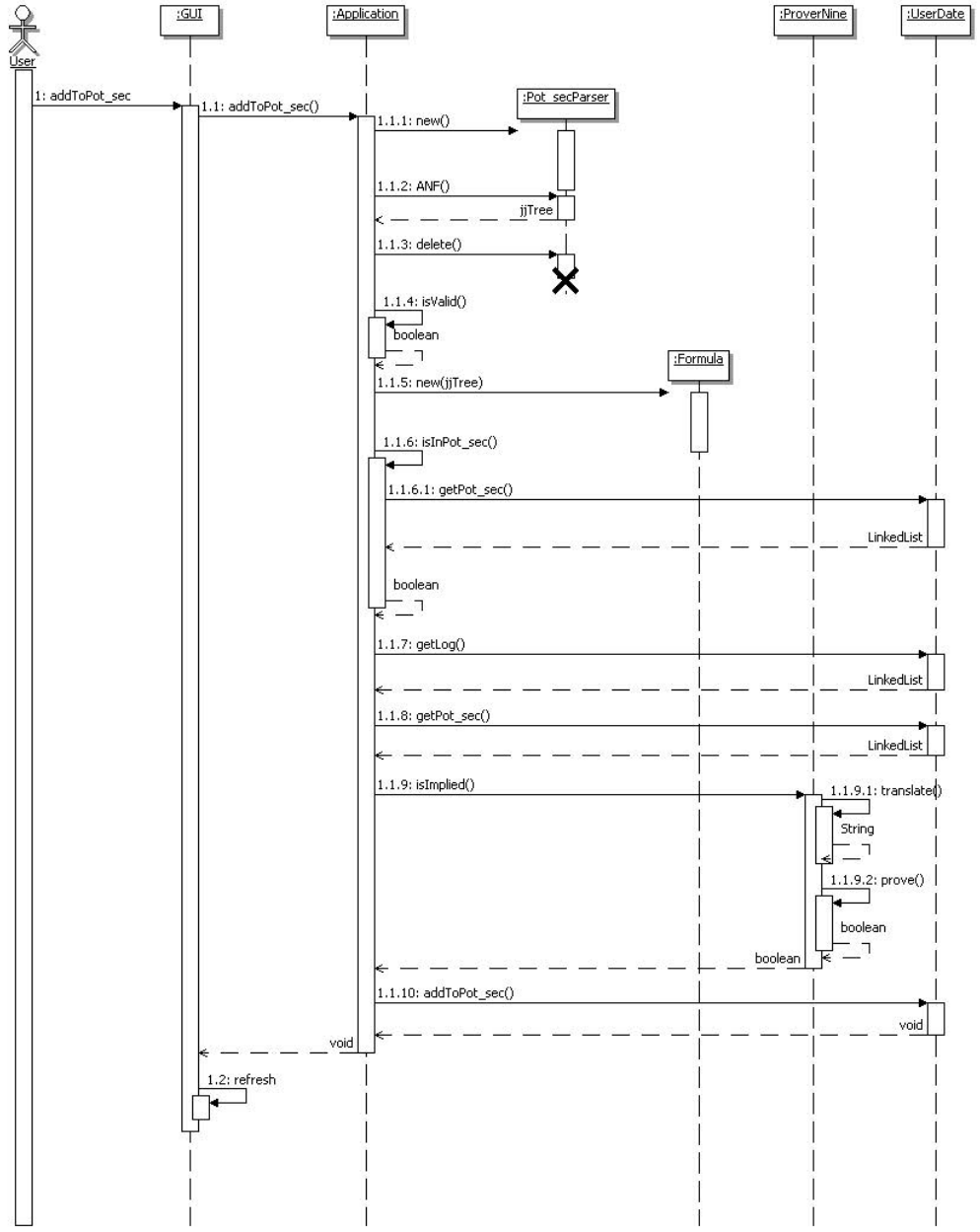

Abbildung 16: Sequenzdiagramm AddToPotsec

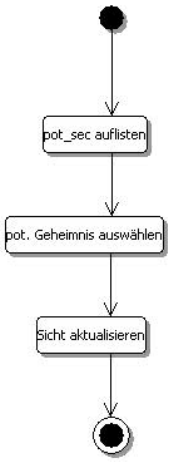

Abbildung 17: Aktivitätsdiagramm DeleteFromPotsec

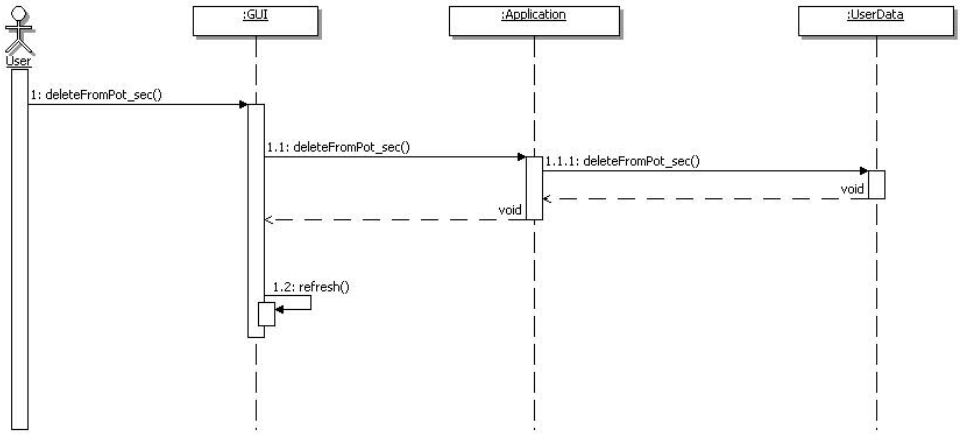

Abbildung 18: Sequenzdiagramm DeleteFromPotsec

12.6 Die Methode classify

Dieser Abschnitt dient der Beschreibung der Methode classify aus der Klasse *RefusalCensor*.
CType classify(LinkedList log, Formula query, LinkedList pot_sec)

12.6.1 Aufbau

Die Methode *classify* bekommt das Benutzerwissen *log*, die Anfrage *query* und die Menge der potentiellen Geheimnisse *pot_sec* übergeben. Sie liefert einen Wert des Typs *CType* zurück. Dabei ist *CType* ein eigens definierter Datentyp, der folgende Werte annehmen kann:

- UNCRITICAL
- MUM
- LIE

12.6.2 Ausführung

Beim Programmstart wird eine Instanz des *RefusalCensor* erzeugt, dieser wird zum ersten Mal benutzt, wenn aus der Methode cqe vom Hauptprogramm aus (d.h. aus der Klasse *Application*) die Methode *classify* des *RefusalCensors* aufgerufen wird.
Beim Aufruf erhält die Methode alle Übergabeparameter, welche dem Benutzer des Programms zugeordnet sind. Dies sind das Benutzerwissen *log*, die Benutzeranfrage *query* und die Menge der potentiellen Geheimnisse *pot_sec*.
Da es sich bei unserem Zensor der ersten Iteration um einen Verweigerungs-Zensor handelt, müssen wir sicherstellen, dass folgende Formel nicht wahr ist:
$\exists \Psi \in pot\_sec$: $log \cup \{eval^*(\Phi_i)(db)\} \models \Psi$ or $log \cup \{\neg eval^*(\Phi_i)(db)\} \models \Psi$
Um dies zu überprüfen, muss zunächst der linke Teil der Formel (bzgl. des or's) ausgewertet werden. Dazu wird die Methode *isImplied* der Klasse *ProverNine* mit den Übergabeparametern von *classify* aufgerufen. Diese durchläuft nun intern eine Schleife, in der für alle übergebenen potentiellen Geheimnisse folgende Schritte ausgeführt werden:

- Sie ruft als erstes die Methode *translate* des *ProverNine* auf, damit die Übergabeparameter in ein gültiges Eingabeformat des Theorembeweisers überführt werden. Diese Eingabe wird von der Methode als String zurückgegeben und anschließend als Inputdatei temporär gespeichert.
- Danach wird die Methode *prove* des *ProverNine* aufgerufen, in der der Theorembeweiser zum Überprüfen der Implikationen aufgerufen wird. Abhängig von dessen Ausgabe gibt diese Methode „true“ zurück, falls die Implikation wahr ist und im anderen Fall „false“.

Sobald eine Implikation wahr ist, wird die Schleife abgebrochen und die Methode *isImplied* liefert den Rückgabewert „true“, ansonsten wird die Schleife komplett durchlaufen und *isImplied* liefert „false“ zurück.
Wenn bis hierhin *isImplied* nicht den Wert „true“ geliefert hat, so wird auch noch der rechte Teil der Formel (bzgl. des or's) untersucht. Dazu wird die Anfrage *query* negiert und erneut mit *isImplied* ausgewertet.
Wenn auch dieser Durchlauf den Wert „false“ zurückliefert, ist die Anfrage für uns unbedenklich und wir können dessen Ergebnis ausgeben lassen. Daher wird *classify* den Rückgabewert „UNCRITICAL“ liefern.
Wenn nun aber beim ersten oder zweiten Durchlauf der Methode *isImplied* der Wert „true“ zurückgeliefert wurde, so ist die Anfrage für uns kritisch und *classify* muss den Rückgabewert „MUM“ zurückgeben.
Dieser Rückgabewert kann anschließend an den Modifikator weitergeleitet und dort weiter verarbeitet werden.
Der Rückgabewert „LIE“ des Datentyps *CType* wird voraussichtlich erst in der zweiten Iteration benötigt, wurde aber bereits mit integriert.

13 Implementierung

13.1 Was wurde implementiert?

Das Programm erlaubt die Übermittlung geschlossener Anfragen an die Datenbank, deren inferenzgestützte Auswertung und die Rückmeldung an den Nutzer, wobei die Anfragen in einer von der Projektgruppe festgelegten Sprache eingegeben werden. Eine genauere Spezifikation der Eingabesprache kann im Anforderungsdokument der 1. Iteration (Kapitel 11) nachgelesen werden. Nachdem die Anfrage auf syntaktische und semantische Korrektheit geprüft und in die SQL-Syntax übersetzt wurde, wird sie an den Theorembeweiser geschickt, um das Implikationsproblem zu lösen. Es wurden also diverse Parserfunktionalitäten und eine Schemaverwaltung implementiert. Die Programmierung der Theorembeweiser-Schnittstelle war wichtig, um die Einbindung der Theorembeweiser-Funktion zu gewährleisten. Ein Datenbankmanagement wurde umgesetzt. Das heisst, dass ein Verbindungsaufbau und das Anmelden bei der Datenbank programmintern administriert werden. Das Programm unterstützt die Verwaltung der potentiellen Geheimnisse (pot_sec) und des Benutzerwissens (log).

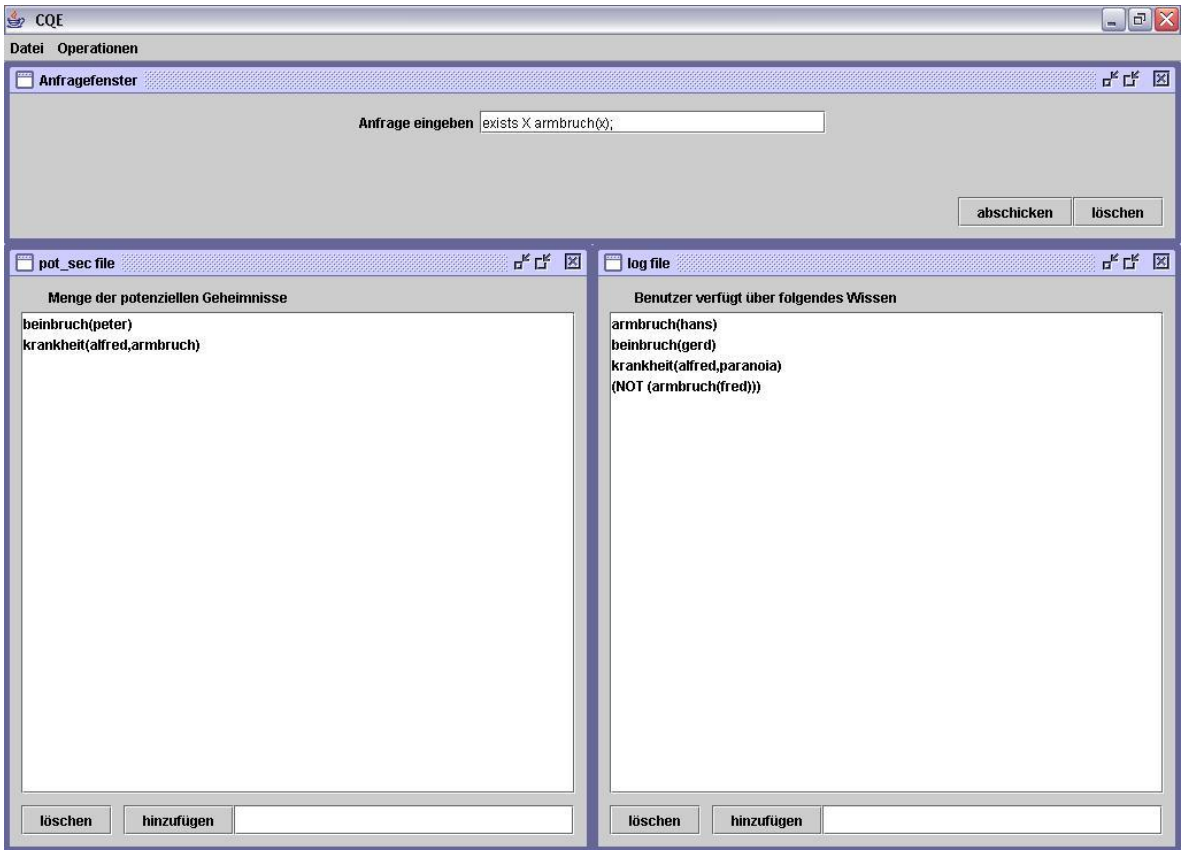

Abbildung 19: GUI

Das Programm ist modular aufgebaut und nach den objektorientierten Prinzipien

konstruiert. Das Datenmodell der Anwendung ist von der GUI unabhängig implementiert worden. Es ist unproblematisch, das Programm später an eine andere GUI anzubinden oder als Plugin zu verwenden. Weiterhin verfügt das Programm über die Schnittstelle zum Theorembeweiser, somit ist eine Anbindung eines anderen Theorembeweisers jederzeit möglich.

Die GUI ist für das Editieren der Anfragen und gleichzeitig für das Verwalten von Sicherheitspolitiken geeignet, da der Nutzer in der ersten Iteration konzeptionell als Administrator angesehen wird. Das heisst, dass in weiteren Iterationen ein Mehrbenutzerbetrieb implementiert wird. Die potentiellen Geheimnisse und das Benutzerwissen werden dann nur vom Administrator verwaltet.

Es geht in der ersten Iteration um die experimentelle Umsetzung des Konzeptes und um die Implementierung der Hauptfunktionalitäten des Programms. Der Aufbau von Syntaxbäumen für die eigentlichen Anfrageformeln, für die Formeln aus dem Benutzerwissen (log) und der Menge potentieller Geheimnisse (pot_sec) ist unterstützt, und somit ist das Eingeben und das Generieren nur syntaktisch korrekter Formeln gewährleistet. Die Anfrageformeln werden den Anforderungen entsprechend in die SQL-Syntax übersetzt. Falls die eingegebene Formel mit einer existierenden Relation der Datenbank assoziiert ist, findet ein Abgleich der Attributanzahl und der Attributnamen mit den Metadaten dieser Relation statt. So wird eine Anfrage der Form „*exists X armbruch(X);*“ als gültig angesehen, da eine Relation armbruch in der Datenbank enthalten ist und die Attributanzahl richtig eingegeben ist (Alle Beispielrelationen sind im Kapitel 13.4.4 aufgelistet und können dort nachgelesen werden). In diesem Fall wird dem Nutzer die Antwort der Datenbank „*Die Anfrage liefert true*“ angezeigt und die abgefragte Formel in das Benutzerwissen eingefügt und durch die GUI repräsentiert.

Bei dem Versuch eine Anfrage der Form „*exists X armbruch(X, paranoia);*“ abzuschicken, wird der Nutzer eine Fehlermeldung bekommen, da die Attributanzahl der Relation armbruch falsch ist. Bei der Eingabe eines ungültigen Relationennamens wird ebenfalls eine Fehlermeldung geworfen.

Die Anbindung des Theorembeweisers an das Programm und die Erstellung des Input-Files ist erfolgreich implementiert worden. Die Input-Datei besteht aus den Eingabeformeln, die der Theorembeweiser zum Prüfen der Implikation benötigt, und muss in der Syntax editiert werden, die der jeweilige Theorembeweiser akzeptiert. Sie ist an jedem Rechner im Workspace enthalten, wobei jeder User Besitzer der Datei ist und sie somit auch beliebig überschreiben kann. Damit kann das Programm an jedem Rechner gleichzeitig gestartet und korrekt ausgeführt werden.

Falls die Menge potentieller Geheimnisse die Formel „*armbruch(alfred)*“ enthält und die Anfrage der Form „*armbruch(alfred);*“ gestellt wird, wird die Anfrage durch den Ablehnungszensor verworfen. Somit ist eine kontrollierte Anfrageauswertung für die geschlossenen Anfragen im Programm gewährleistet. Daten, die in die Menge Benutzerwissen und in die Menge potentieller Geheimnisse eingefügt werden, werden beim Beenden des Programms in einem externen File gespeichert und beim neuen Start des Programms aus diesem File ausgelesen und dem Nutzer angezeigt. Es ist auch das Vorkommen von nur gebundenen Variablen in einer Formel gewähleistet.

13.2 Was ist noch zu tun?

Der Nutzer soll über die Fehler mit einer genauen Angabe des Grundes benachrichtigt werden. Weiterhin soll die Menge der potentiellen Geheimnisse auf redundante Formeln geprüft werden, dies wird wesentlich zur Effizienz des gesamten Programms beitragen. Die Fehlerberichtserstattung soll verbessert werden. Damit ist gemeint, dass alle Fehlermeldungen mehr Informationen über die Ursache der Fehler und den genauen Ort des Fehlerauftretens enthalten. Beim Verbindungsaufbau mit der Datenbank soll die Verwaltung des Benutzernamens und des Passwortes anders implementiert werden. Benutzername und Passwort kommen dann nicht mehr im Programmcode vor, da es eine Verwundbarkeit des Codes darstellt. Solche Daten sollen extern gespeichert und beim Verbindungsaufbau mit der Datenbank verifiziert werden. So könnte man eine globale Verbindungsaufbau implementieren und die Verbindungsdaten in globalen Variablen ablegen. Es soll noch ein Vergleich für Formeln ermöglicht werden, der angibt, ob zwei verschiedene Formula Objekte inhaltlich gleich sind. Das heisst, dass die Formeln der Form „*exists X armbruch(X)*“ und „*exists Y armbruch(Y)*“ als äquivalent angesehen werden.

13.3 Was wurde verworfen? Gründe!

Die Konsistenzprüfung des Benutzerwissens wurde auf die 2. Iteration verschoben. Die Formeln der Datei „log“ werden dann alle von der Datenbankinstanz impliziert sein (Falls die Verweigerung-Methode benutzt wird). Für das Benutzerwissen wurden negierte Formeln erlaubt (dies ist eine Abweichung von den Anforderungen an die Menge Benutzerwissen). Die negierten Formeln werden nicht danach geprüft, ob sie von der Datenbank tatsächlich impliziert sind. Dies hängt mit der Translation der negierten Formeln in SQL-Syntax zusammen, für die die Projektgruppe noch keine Lösung gefunden hat. Es soll noch die Frage geklärt werden, ob bei der behandelnen Implikation es sich um eine DB-Implikation handelt.

13.4 Auswahl der Plattform und benutzte Software

13.4.1 Programmierungsumgebung

Auf den Rechnern im PG-Pool ist das Betriebssystem Debian Linux verfügbar. Die Projektgruppe hat die Programmiersprache Java(JDK 1.5) benutzt. Als Modellierungs- und Programmierumgebung wurden Borland Together Architect 2006 und Eclipse 3.1 ausgewählt.

13.4.2 JavaCC

Als Parser-Generator benutzt die Projekt-Gruppe Java Compiler Compiler. Mehr dazu im Kapitel 12.1.3.

13.4.3 Theorembeweiser

An das Programm ist der Theorembeweiser Prover9, des Otters designierter Nachfolger, angebunden. Als Auswahlkriterien waren die gute Performance, das Operieren mit der Prädikatenlogik und eine leichte Bedienbarkeit dieser Software entscheidend.

13.4.4 Datenbank

Es wird eine durch den Lehrstuhl 6 zur Verfügung gestellte Oracle-Datenbank benutzt. Die Datenbank-Verbindung passiert mit Hilfe von JDBC-Technologie. Alle Informationen zu der JDBC-Technologie können im Kapitel 6 und im Kapitel 7 nachgelesen werden.
Für experimentelle Zwecke wurden folgende Relationen angelegt:

1. Relation *armbruch*

armbruch	NAME
	alfred
	hans

2. Relation *beinbruch*

beinbruch	NAME
	gerd
	peter

3. Relation *krankheit*

krankheit	NAME	KRANKHEIT
	fred	cold
	alfred	paranoia
	frank	death
	alfred	armbruch
	hans	armbruch

Anhand dieser Relationen konnte man eine logische Beziehung zwischen den Tabellen *armbruch* und *krankheit* modellieren. Bestimmte Informationen der Relation *armbruch* sind aus der Kenntnis der Relation *krankheit* ableitbar. Und zwar lautet die semantische Bedingung: $\forall X : krankheit(X, armbruch) \Rightarrow armbruch(X)$.

13.5 Änderungen

Laut dem Aktivitätsdiagramm für addToLog sollen beim Einfügen einer Formel zu der Menge „Benutzerwissen“ die Preconditions geprüft werden. Dieser Schritt wird nicht durchgeführt und somit die Konsistenzprüfung der Menge „Benutzerwissen“ ausgelassen. Es gibt keine weiteren Änderungen gegenüber den Entwurfs- und Anforderungsdokumenten.

13.6 Programmmodule

13.6.1 Theorembeweiser

Um den Theorembeweiser in unserem Projekt benutzen zu können, sollten bestimmte Softwarepackete im Package jCQE positioniert werden.
In dem Verzeichnis jCQE/src/Prover9 befinden sich Dateien, die für das Ausführen des Theorembeweisers zuständig sind. Die binäre prover9-Datei ist für den Aufruf unter Linux notwendig. cygwin1.dll und prover9.exe sind die Dateien für die Ausführung von Prover9 unter Windows. Input.in ist die Inputdatei. Der Theorembeweiser wird in der Klasse ProverNine in der Methode prove() zum Überprüfen der Implikationen aufgerufen.

```
public boolean prove() {

int exitCode = 0;

try{
ProcessBuilder builder =
  new ProcessBuilder( "src/Prover9/prover9", "-f",
   "src/Prover9/Input.in" );
    Process p = builder.start();
    p.waitFor();
    exitCode = p.exitValue();
}
catch (IOException e) {
        System.out.println("Fehler aufgetreten bei prove:
         "+e.getMessage());
    }
catch (InterruptedException e) {
System.out.println("Fehler aufgetreten bei prove: "+e.getMessage());
}

//Implikation nicht wahr
if(exitCode == 2) return false;

//Implikation wahr
if(exitCode == 0) return true;

//Sonst gab es einen anderen Fehler
System.out.println("Fehler in prove,
                  Prover9 beendet mit fatalem Error! " + exitCode);

return true;
}
```

Das Benutzerwissen „log“, die Benutzeranfrage und die Menge der potentiellen Geheimnisse werden in eine passende Eingabesprache des Theorembeweisers überführt. Anschließend wird die Eingabe in eine Inputdatei des Theorembeweisers geschrieben. Die Inputdatei des Theorembeweisers kann z.B. folgende Informationen enthalten.

```
clear(auto_denials).

formulas(sos).
(-(armbruch(fred)) | armbruch(horst)).
(armbruch(fred)).
end_of_list.

formulas(goals).
(armbruch(horst)).
end_of_list.
```

13.6.2 Parser

Alle Informationen zu den Parserfunktionalitäten können im Beschreibungsdokument zu JavaCC (Kapitel 12.1.3) nachgelesen werden.

13.6.3 Verwaltung der Formeln

Ein Objekt Formula baut auf der Datenstruktur TreeNode von JavaCC auf, in der Syntaxbäume generiert und verwaltet werden. Hier ist die Methode dfs() der Klasse Formula von großer Relevanz, da sie für geschlossene Anfragen mit einem dfs-Durchlauf durch den Syntaxbaum der Formel den entsprechenden Befehl generiert. Hier wird eine Verbindung mit der Datenbank aufgebaut und eine Anfrage an die Datenbank abgeschickt.
Eine Formel soll negierbar sein, dafür ist die Methode negate() der Klasse Formula verantwortlich.

```
public void negate() {
if (syntaxtree.getNumChildren() > 1) {
for (int i=0; i<=(syntaxtree.getNumChildren()-1)/2; i+=2) {
if ((syntaxtree.getChild(i).toString()).equals("EXISTS"))
syntaxtree.getChild(i).setString("FORALL");
else
syntaxtree.getChild(i).setString("EXISTS");
}
}
TreeNode n = syntaxtree.getChild(syntaxtree.getNumChildren()-1);
switch (n.getNumChildren()) {
```

```
case 2:
syntaxtree.addChild(syntaxtree.getNumChildren(), n.getChild(1));
syntaxtree.removeChild(syntaxtree.getNumChildren()-2);
break;
default:
TreeNode exp2 = new TreeNode("EXP");
exp2.addChild(0, new TreeNode("NOT"));
exp2.addChild(1, n);
syntaxtree.addChild(syntaxtree.getNumChildren(), exp2);
syntaxtree.removeChild(syntaxtree.getNumChildren()-2);
}
string = createString(syntaxtree);
}
```

13.6.4 Ablehnungszensor und Modifikator

Nachdem dem Theorembeweiser alle wichtigen Daten für die Auswertung des Implikationsproblems („log“, „pot_sec“, „query“) zur Verfügung gestellt wurden und diese Auswertung abgeschlossen ist, muss das Ergebniss bewertet werden. Anschließend bekommt man eine Anweisung des Modifikators entweder zur Ablehnung der Anfrage oder zur Ausgabe der einfachen Anfrageauswertung. Dies ist in der Klasse RefusalCensor in der Methode classify() realisiert worden.

```
CType  classify(LinkedList log, Formula query,
LinkedList pot_sec) {

tpInterface = new ProverNine();
boolean ergebnis = false;

//Prüfe, ob die Implikation log u query |= psi für ein psi aus pot_sec
//wahr ist
ergebnis = tpInterface.isImplied(log, query, pot_sec);

//Abfrage ob wir bereits eine wahre Implikation gefunden haben,
//falls ja, können wir schon "true" zurückgeben
if(ergebnis == true) return CType.MUM;

//Wir haben es mit dem RefusalZensor zu tun, daher query negieren
query.negate();

//Prüfe, ob die Implikation log u -query |= psi für ein psi aus
//pot_sec wahr ist
ergebnis = tpInterface.isImplied(log, query, pot_sec);
```

```
query.negate();

if(ergebnis == true) return CType.MUM;

//Die Implikation log u (-)query |= psi ist für kein psi aus pot_sec
//wahr. Die Anfrage ist daher unbedenklich.
return CType.UNCRITICAL;
}
```

Wobei CType eine Datenstruktur darstellt und die folgenden Konstanten enthält:

```
// die Anfrage ist unbedenklich
public static final CType UNCRITICAL = new CType();
// die Anfrage soll abgelehnt werden
public static final CType MUM  = new CType();
// Bezüglich der Anfrage soll gelogen werden
public static final CType LIE  = new CType();
```

Die Methode modify() der Klasse Modificator gibt in Abhängigkeit vom Rückgabewert der Methode classify() entweder „true“ oder „false“ oder „mum“ zurück. Falls die Anfrage als kritisch bewertet wurde und abgelehnt werden soll, wird „mum“ an den Nutzer weitergeleitet. Falls die Anfrageauswertung dem Nutzer zugänglich sein darf, wird „true“ oder „false“ zurückgegeben, da es sich um eine geschlossene Anfrage handelt.

13.6.5 Verwaltung des Benutzerwissens und der potentiellen Geheimnisse

Die Formeln, die zu der Menge Benutzerwissen und zu der Menge potentieller Geheimnisse gehören, werden in der Klasse UserData verwaltet. Die Formula Objekte werden in einer verketteten Liste gehalten, wodurch das Einfügen und das Löschen der Formeln effizient unterstützt wird. Das Nutzerwissen und die potentiellen Geheimnisse, also die Informationen, die vor dem Nutzer geheim gehalten werden sollen, werden getrennt in zwei Listen verwaltet.
Es ist das Serialisieren und das Deserialisieren der Formula Objekte implementiert worden. Das Serialisieren ist eine sehr komfortable Möglichkeit, ein ganzes Geflecht von Objekten in eine „serielle Form“ einer Folge von Bytes zu bringen, die sich zum Speichern auf einer sequentiellen Datei oder zum Transfer über die Zwischenablage eignet. Auf diese Weise läßt sich auch die Persistenz von Objekten implementieren, das ist das „Fortleben“ von Objekten über einen einzigen Programmlauf hinaus. Diese Funktionalität ist in der Klasse SerializeDeserialize programmiert worden.

14 Test

14.1 Einleitung

Die Projektgruppe hat sich in der ersten Iteration dafür entschieden, alle implementierten Methoden, die einen Rückgabewert liefern, zu testen. Für diesen Zweck wurden verschiedene Testklassen implementiert, in denen die zu testenden Methoden auf Korrektheit überprüft wurden. Dabei wurden möglichst anhand kritische Eingaben die Ausgaben der Methoden betrachtet und mit den korrekten erwarteten Ausgaben verglichen.

14.2 Test der Klasse Formula

14.2.1 Wichtige Tests für die Klasse Formula

Die wichtigsten und zugleich kritischen Funktionalitäten der Klasse *Formula* sind das Übersetzen der Formel in eine entsprechende SQL-Anfrage, die Negation der Formel und die Ausgabe der Formel als String. Diese drei Funktionen werden in den Methoden *toSQL, negate* und *toString* implementiert. Die Klasse *xxFormulaTest* implementiert entsprechende Tests für diese drei Methoden, wobei insbesondere kritische Eingaben betrachtet werden sollten. Hierzu gehören theoretisch Sonderfälle und unzulässige Eingaben. Unzulässige Eingaben sind nach Voraussetzung allerdings ausgeschlossen, da eine syntaktische sowie eine semantische Prüfung der vom Benutzer gemachten Eingaben erfolgt, bevor das entsprechende Formelobjekt erzeugt wird. Es existieren auch keine nennenswerten Sonderfälle. Darum wird hier ein simples und ein komplexes Beispiel für eine Formel getestet.

14.2.2 Durchgeführte Tests

Folgende Testmethoden wurden implementiert:

1. *testNegate()*
2. *testSQL()*
3. *testToString()*

Setup Das Setup der Tests beinhaltet die Erzeugung von drei Formeln mit denen die Tests durchgeführt werden sollen. Zwei der Formeln sind identisch und relativ simpel. Die dritte Formel ist etwas komplexer. Die Formeln wurden folgendermaßen erzeugt:

```
queryParser.ReInit(new StringReader(
        "exists X (armbruch(X) and beinbruch(gerd));"));
```

```
f = new Formula(new TreeNode(queryParser.ANF()));
queryParser.ReInit(new StringReader(
        "exists X (armbruch(X) and beinbruch(gerd));"));
g = new Formula(new TreeNode(queryParser.ANF()));
queryParser.ReInit(new StringReader("exists X exists Y
        (armbruch(X) and (beinbruch(gerd) or krankheit(horst, Y)));"));
h = new Formula(new TreeNode(queryParser.ANF()));
```

Negation Zunächst wurde die Negation getestet. Dazu wurde die erste Formel doppelt negiert. Erwartungsgemäß sollte das Ergebnis wieder die Ausgangsformel sein. Folgende Assertion wurde durchgeführt:

```
g.negate();
g.negate();
Assert.assertTrue(f.equals(g));
```

Zu diesem Zweck wurde die von Object geerbte Methode *equals()* in *Formula* überschrieben. Die Methode *equals()* liefert nun *true*, wenn die übergebene Formel die gleiche Stringrepräsentation besitzt, wie die aufrufende Formel.

Ergebnis Dieser Test wurde problemlos durchgeführt.

Übersetzung in SQL Um diesen Test durchzuführen, wurde in einer Assertion das Ergebnis der Methode *toSQL()* mit dem laut Theorie korrekten String verglichen. Das sah so aus:

```
Assert.assertEquals(f.toSQL(), "SELECT DISTINCT 'c0' FROM
        ((SELECT NAME AS X FROM armbruch) NATURAL JOIN
        (SELECT DISTINCT 'c1' FROM (SELECT * FROM beinbruch
        WHERE name='gerd')))");
Assert.assertEquals(h.toSQL(), "SELECT DISTINCT 'c0' FROM
        ((SELECT * FROM (SELECT NAME AS X FROM armbruch)
        NATURAL JOIN ((SELECT * FROM (SELECT DISTINCT 'c1'
        FROM (SELECT * FROM beinbruch WHERE NAME='gerd'))
        NATURAL FULL OUTER JOIN (SELECT KRANKHEIT AS Y FROM
        krankheit WHERE NAME='horst')))))");
```

Ergebnis Auch dieser Test wurde erfolgreich durchgeführt.

Ausgabe als String Zuletzt wurde die Methode *toString()* getestet. Dazu wird wiederum das Ergebnis der Methode mit dem zu erwartenden String verglichen:

```
Assert.assertEquals(f.toString(), "EXISTS X (armbruch(X)
        AND beinbruch(gerd))");
Assert.assertEquals(h.toString(), "EXISTS X EXISTS Y
        (armbruch(X) AND (beinbruch(gerd) OR krankheit(horst,Y)))");
```

Ergebnis Wiederum ein fehlerfreier Durchlauf des Tests.

14.3 Test der Klasse Application

Die beiden Methoden **sqe()** und **isValid()** von der Klasse **Application** werden innerhalb der Klasse **xxApplicationTest** getestet. Da Die anderen Methoden von der Klasse **xxApplicationTest** keine Rückgabewert liefern verzichten wir auf deren Tests.
Die Testklasse enthält enthält folgenden Testmethoden:

- testSqe()
- testIsValid()

Jede diese Methoden testet die entsprechende Methode von der Klasse Application anhand der verschiedenen Übergabewerte.

14.3.1 Testmethode testSqe()

Hierbei wird die Methode sqe() von der Klasse Application anhand der folgenden Werte mit erwarteten Datenbank-Ausgaben getestet:

- SELECT * FROM ARMBRUCH WHERE NAME = 'alfred' (erwarteter Ausgabewert: true)
- SELECT * FROM ARMBRUCH WHERE NAME = 'peiman' (erwarteter Ausgabewert: false)
- SELECT * FROM KRANKHEIT WHERE NAME = 'pedram' OR KRANKHEIT = 'paranoia' (erwarteter Ausgabewert: true)
- SELECT * FROM KRANKHEIT WHERE NAME = 'pedram' AND KRANKHEIT = 'paranoia' (erwarteter Ausgabewert: false)
- SELECT * FROM KRANKHEIT WHERE NAME = 'frank' AND KRANKHEIT = 'death' (erwarteter Ausgabewert: true)

Mit der Methode **Assert.assertEquals()** werden die jeweiligen Werten überprüft. Kein Fehler trat auf.

14.3.2 Testmethode testIsValid()

Hierbei wird die Methode isValid() von der Klasse Application anhand der folgenden Werte mit erwarteten Datenbank-Ausgaben getestet:

- exists X armbruch(X); (erwarteter Ausgabewert: true)
- exists X krankheit(fred, X); (erwarteter Ausgabewert: true)

- exists X armbruch(alfred, X); (erwarteter Ausgabewert: false)
- exists X nasenbruch(X); (erwarteter Ausgabewert: false)

Die Eingaben werden erstmal innerhalb der Methode setUp() bearbeitet und die zugehörigen Syntaxbäume erzeugt.
Mit der Methode **Assert.assertEquals()** werden die jeweiligen Werten überprüft.
Es stellt sich hinaus, dass bei den ersten beiden Eingaben, wo wir true als Testergebnis erwarten, ein Fehler auftritt.

14.4 Test der Klasse ProverNine

Dieser Abschnitt dient der Beschreibung der JUnit-Testklasse *xxProverNineTest*.
Die Testklasse *xxProverNineTest* besitzt die folgenden Testmethoden:

- public void testIsImplied()
- public void testFormulaToString()
- public void testTranslate()
- public void testWriteToFile()

Jede dieser Methoden ruft die entsprechende Methode der Klasse *ProverNine* auf und testet diese mit verschiedenen Übergabewerten. Tritt bei der Ausführung von der Testklasse mit JUnit ein Fehler auf, so wird dies von JUnit protokolliert und direkt in der Entwicklungsumgebung angezeigt.
Um die Tests besser nachzuvollziehen, sollte man die Klasse *xxProverNineTest* vor sich geöffnet haben. Im Folgenden betrachten wir jede Testmethode einzeln.

14.4.1 Testmethode testIsImplied

Test 1 Zunächst werden die drei Parser (*LogParser*, *Pot_ secParser* und *QueryParser*) initialisiert. Dazu übergeben wir Ihnen die Eingaben log=$\{armbruch(fred)\}$ als Benutzerwissen, Ψ=$\{armbruch(horst)\}$ als potentielles Geheimnis und die Benutzeranfrage $query$=$\{armbruch(horst)\}$.
Als erstes wird nun mit dem Aufruf der Methode *isImplied* getestet, ob die Methode *isImplied* erkennt, dass die Implikation $log \cup \{\text{query}\} \models \Psi$ „wahr“ ist.
Dies geschieht durch diesen Aufruf:

```
assertEquals(proverNine.isImplied(lLog,fquery,lpot_sec), true);
```

Ergebnis Test 1 Unser Theorembeweiser ProverNine liefert uns in diesem Fall das Ergebnis, dass die Implikation „wahr“ ist, daher stimmt dieser Test.

Test 2 Der nächste Test

```
assertEquals(proverNine.isImplied(lpot_sec,fquery,lLog), false);
```

vertauscht einfach das potentielle Geheimnis und das Benutzerwissen.

Ergebnis Test 2 Dieser Aufruf liefert daher das erwartete Ergebnis „falsch“.

Test 3 Im letzten Test dieser Methode ändern wir unsere Eingaben wie folgt:

- log=$\{((not\ armbruch(fred))\ or\ armbruch(horst))\}$
- Ψ=$\{armbruch(horst)\}$
- $query$=$\{armbruch(fred)\}$

Wir haben es hier mit einer Implikation im Benutzerwissen zu tun, denn die Formel $((not\ armbruch(fred))\ or\ armbruch(horst))$ ist äquivalent zu der Formel $armbruch(fred)) \implies armbruch(horst)$. D.h. wenn wir wissen, dass Fred einen Armbruch hat, so hat auch Horst einen Armbruch. Wir rufen den Test auf mit

```
assertEquals(proverNine.isImplied(lLog,fquery,lpot_sec), true);.
```

Ergebnis Test 3 Der Benutzer fragt hier ob Fred einen Armbruch hat, da durch die Implikation im Benutzerwissen die Implikation $log \cup \{query\} \models \Psi$ „wahr“ ist, wird auch dieser Test erfolgreich beendet.

14.4.2 Testmethode testFormulaToString

Test 1 Die Testmethode *testFormulaToString* soll überprüfen, ob die Methode *FormulaToString* der Klasse *ProverNine* eine Formel korrekt in einen String übersetzt, welcher später als Eingabe für den Prover9 weiterverwendet werden soll. Als erstes wird der *QueryParser* mit der Formel $exists\ X\ armbruch(X)$; initialisiert.
Der erste Test der Methode wird mit

```
assertEquals(proverNine.FormulaToString(fquery.getSyntaxTree()),
 "exists X (armbruch(X)).");
```

aufgerufen.

Ergebnis Test 1 Das Ergebnis stimmt mit dem erwartetem Ergebnis überein, daher werden noch zwei weitere Tests ausgeführt.

Test 2 Es wird der *QueryParser* mit der Formel $exists\ X\ (krankheit(X, paranoia)\ and\ armbruch(X))$; initialisiert und mit

```
assertEquals(proverNine.FormulaToString(fquery.getSyntaxTree()),
 "exists X (krankheit(X, paranoia) & armbruch(X)).");
```

getestet.

Ergebnis Test 2 Auch dieser Test ist erfolgreich.

Test 3 Es wird der *QueryParser* mit der Formel *exists X exists Y ((krankheit(X, paranoia) and armbruch(X)) or krankheit(fred, Y))*; initialisiert und mit

```
assertEquals(proverNine.FormulaToString(fquery.getSyntaxTree()),
"exists X exists Y (krankheit(X, paranoia) & armbruch(X) |
 krankheit(fred, Y)).");
```

getestet.

Ergebnis Test 3 Auch dieser Test verläuft planmäßig.

14.4.3 Testmethode testTranslate

In der Testmethode *testTranslate* wird überpüft, ob anhand der Übergabeparameter *lLog* und *fpot_ sec* ein korrektes Inputfile bzw. der zugehörige String für den Prover9 erzeugt wird. Die zu testende Methode wird mit

```
proverNine.translate(lLog, fpot_sec)
```

aufgerufen und liefert den entstandenen String zurück, der mit dem erwarteten String verglichen wird.

```
assertEquals(proverNine.translate(lLog, fpot_sec), "clear(auto_denials).
\n\nformulas(sos).\n(armbruch(fred)).\n(armbruch(horst)).\n
end_of_list.\n\nformulas(goals).\n");
```

Ergebnis Da beide Strings übereinstimmen, ist der Test erfolgreich.

14.4.4 Testmethode testWriteToFile

In der Testmethode *testWriteToFile* wird überprüft, ob ein Text richtig in eine .txt Datei geschrieben werden kann.

Durch den Aufruf

```
proverNine.writeToFile(writetext ,fileName);
```

wird zuerst eine neue Datei *fileName* erstellt und dann der übergebene Text *writetext* in diese Datei geschrieben. Zum Ende wird der Test

```
assertEquals(writetext, readtext.toString());
```

aufgerufen und überpüft, ob der eingelesene Text vollständig geschrieben wurde.

Ergebnis Da die beiden Texte identisch sind, ist auch dieser Test erfolgreich.

14.5 Test der Klasse Modificator

In diesem Dokument wird die JUnit-Testklasse *xxModificatorTest* beschrieben.
Die Testklasse *xxModificatorTest* hat die folgende Testmethode:

- *public void testmodificator()*

Diese Methode ruft die entsprechende Methode der Klasse *Modificator* auf und testet diese mit verschiedenen Übergabewerten. Alle bei der Ausführung von der Testklasse mit JUnit autretenden Fehler, werden von JUnit protokolliert und direkt in der Entwicklungsumgebung angezeigt.
Im Folgenden wird die Testmethode *testmodify* näher dargestellt.

14.5.1 Testmethode testmodify

Die Testmethode testmodify überprüft, ob die Methode modify der Klasse Modificator in Abhängigkeit von Übergabewerten die korrekten Ergebnisse liefert. Es wird z.B. der Fall mit den Übergabeparametern $eval = true$ und $classification = CType.UNCRITICAL$ getestet. Dies geschieht durch diesen Aufruf:

assertEquals(„*Die wahre Antwort ist true.*“, „*Die Anfrage liefert true*“, *modificator.modify*($eval, classification$));

Der Modificator wird das Ergebnis „Die Anfrage liefert true“ ausgeben, falls die Anfrage in unserer Datenbank enthalten ist und falls RefusalCensor diese Anfrage für *„unkritisch“* hält (d.h., aus dieser Anfrage können keine potentiellen Geheimnisse abgeleitet werden).
Dieser Test ist erfolgreich.

14.6 Test der Klasse UserData

Dieser Abschnitt dient der Beschreibung der JUnit-Testklasse xxUserDataTest. Die Testklasse xxUserDataTest besitzt die folgenden Testmethoden:

- public void xxUserDatatest()
- public void testGetLog()
- public void testGetPot_sec()
- public void testAddToLog()
- public void testAddToPot_sec()
- public void testDeleteFromLog()
- public void testDeleteFromPot_sec()

Die Klasse UserData verwaltet Benutzerwissen durch LinkedList. Zum Test der Klasse UserData wollen wir wissen, ob Benutzerwissen fehlerfrei durch LinkedList verwaltet wird.

14.6.1 Testmethoden

Setup Das setup der Tests beinhaltet die Erzeugung von UserData mit denen die Tests durchgeführt werden sollen. Die Instanz wurde folgendermaßen erzeugt:

$$userData = new \quad UserData().$$

xxUserDataTest In der Methode xxUserDataTest wird eine Anfrage erzeugt, die ein Element der log- oder pot_sec Menge sein kann. Außerdem wird durch die beiden Bedingungen:

$$assert(userData.log! = null)$$

$$assert(userData.pot\_sec! = null).$$

getestet, ob LinkedList log und pot_sec erzeugt werden.

Ergebnis von xxUserDataTest Die beiden LinkedLists werden erzeugt.

testGetLog und testGetPot_sec In den beiden Methoden werden getLog und getPot_sec von der Klasse UserData mit den folgenden Befehlen:

$$assert(userData.getLog() == userData.log)$$

$$assert(userData.getPot\_sec() == userData.pot\_sec)$$

getestet.

Ergebnis von testGetLog und testGetPot_sec Die beide Methoden sind fehlerfrei.

testAddToLog und testAddToPot_sec In den beiden Methoden werden addToLog und addToPot_sec von der Klasse UserData mit den folgenden Befehlen:

$$Assert.assertEquals(-1, userData.log.indexOf(f))$$

$$Assert.assertEquals(-1, userData.pot_sec.indexOf(f))$$

getestet, indem es überprüft wird, ob das Element in der log- oder pot_secListe vorhanden ist. Wenn das Element vorhanden ist, wird eine Fehlermeldung ausgegeben. Sonst wird das Element hinzugefügt.

$$userData.addToLog(f)$$

$$userData.addToPot\_sec(f).$$

Die beiden Bedingungen

$$assert(f == (Formula)userData.log.getLast())$$

$$assert(f == (Formula)userData.pot_sec.getLast())$$

überprüfen, ob das Element hinzugefügt wurde.

Ergebnis von testAddToLog und testAddToPot_sec Die beiden Methoden sind fehlerfrei.

testDeleteFromLog und testDeleteFromPot_sec In den beiden Methoden testDeleteFromLog und testDeleteFromPot_sec von Klasse UserData wird jeweils ein Element hinzugefügt. Dann wird dieses Element gelöscht. Zum Schlußwird überprüft, ob genau dieses Element gelöscht wurde.
Für log:

$userData.addToLog(f);$
$LinkedListcopyVonLog = newLinkedList();$
$for(inti = 0; i < userData.log.size() - 1; i++)$
$copyVonLog.add(userData.log.get(i));$
$userData.deleteFromLog(userData.log.size() - 1);$
$for(inti = 0; i < copyVonLog.size(); i++)$
$assert(copyVonLog.get(i) == userData.log.get(i));$

Für pot_sec:

$userData.addToPot\_sec(f);$
$LinkedListcopyVonPot_sec = newLinkedList();$
$for(inti = 0; i < userData.pot_sec.size() - 1; i++);$
$copyVonPot_sec.add(userData.pot_sec.get(i));$
$userData.deleteFromPot\_sec(userData.pot\_sec.size() - 1);$
$for(inti = 0; i < copyVonPot_sec.size(); i++)$
$assert(copyVonPot_sec.get(i) == userData.pot_sec.get(i));$

Ergebnis von testDeleteFromLog und testDeleteFromPot_sec Die beiden Methoden sind fehlerfrei.

15 Fazit

Insgesamt ist die Projektgruppe mit dem Ergebnis der in der ersten Iteration entworfenen und implementierten Software sehr zufrieden. Lediglich einige kleinere Schwächen des Programms haben sich herausgestellt. Diese sollen natürlich in der nächsten Iteration behoben werden. Es soll kurz festgehalten werden welche Verbesserungen für den nächsten Durchlauf vorgesehen sind (diese zählen nicht zu den Erweiterungen in der nächsten Iteration):

1. Es wurde, aufgrund fehlender Kenntnis über die technische Umsetzung der Erstellung entsprechender SQL-Anfragen, auf Negationen in Anfragen verzichtet. Für das Benutzerwissen waren Negationen explizit erlaubt, da die Projektgruppe davon ausging, dass für diese Formeln keine SQL-Anfragen erzeugt werden müssen. Dies stellte sich aber später als falsch heraus. Beim Hinzufügen einer Formel zum Benutzerwissen, muss diese dahingehend überprüft werden, ob sie in der gegebenen DB-Instanz wahr ist. Und eben hierzu muss eine SQL-Anfrage erzeugt werden. Das Benutzerwissen muss immer wahr in der gegebenen DB-Instanz sein. Dies hatte zur Folge, dass auf eine solche Prüfung letztlich verzichtet wurde, was allerdings eine Verletzung der Voraussetzungen für eine kontrollierte Anfrageauswertung darstellt. Darum soll eine Negation in der nächsten Iteration unbedingt implementiert werden.

2. Es fehlt eine allgemeine Konsistenzprüfung für das Benutzerwissen. Formeln, die dem Benutzerwissen neu hinzugefügt werden, werden nicht darauf geprüft, ob sie mit dem bereits vorhandenen Wissen in Konflikt stehen. Folge eines Konflikts wäre eine unerfüllbare Formelmenge. Dies wiederum führt dazu, dass **alle** Anfragen abgelehnt werden. Somit ist eine Konsistenzprüfung für die nächste Iteration vorgesehen.

3. Bislang ist die Fehlerberichterstattung des Programms sehr dürftig ausgefallen. Bei auftretenden Fehlern, wird der Benutzer nur in seltenen Fällen darüber informiert, was genau Ursache des Fehlers war. In der nächsten Iteration soll sich eingehend mit dem Konzept von *Exceptions* beschäftigt werden. Diese machen für praktisch jeden Fall eine relativ genaue Fehlerberichterstattung möglich.

4. Im Moment sind die Verbindungsdaten (Benutzer/Passwort) für das unterliegende Oracle-DBMS im Klartext im Quellcode abgespeichert. Diese sollen in der nächsten Iteration serialisiert werden.

Teil IV
Zweites Release

16 Anforderungen an das zweite Release

16.1 Einleitung

Im Folgenden sollen die Anforderungen für die zweite Iteration spezifiziert werden. Dabei sind speziell die Punkte *Negation in Formeln*, *Benutzertrennung*, *Optimierung und Konsistenzsicherung*, *Fehlerbehandlung* und *Lügen- und kombinierte Methode* von Interesse.

16.2 Negation in Formeln

In diesem Durchlauf soll das Programm dahingehend erweitert werden, dass auch Negationen in Formeln möglich sind. Somit wird unsere sehr begrenzte Sprache aus der ersten Iteration erheblich erweitert. Insbesondere ist dadurch die Darstellung von Implikationen möglich. Durch diese Erweiterung wird die in der ersten Iteration noch fehlende Grundvoraussetzung für eine Konsistenzprüfung des Benutzerwissens erfüllt. Rein syntaktisch waren Negationen auch in der ersten Iteration schon implementiert, nur wurden Anfragen, die Negationen enthalten, vom Programm zurückgewiesen. Es müssen also keine Änderungen an den Parsern vorgenommen werden.

16.3 Benutzertrennung

Die Benutzertrennung ist gemäß der folgenden Rollen zu erreichen:

- Administrator (diese Rolle beinhaltet den funktionalen Administrator und den Sicherheitsadministrator).
- Anfrager(-gruppe).

Dabei soll das Interesse der Authentizität im Sinne von Richtigkeit des Teilnehmers und seiner Benutzergruppenzugehörigkeit gewährleistet werden, aber auch eine Festlegung der diversen Sicherheitspolitiken für mehrere Benutzer und weitergehend auch das Interesse der Autorisierung behandelt werden. Um dies zu erreichen, sollen vor allem die Zugriffsrechte und funktionale Aufgaben für jede der Benutzergruppen festgelegt werden.
Falls es sich um einen Administrator handelt, kann man seine Aufgaben in zwei große Bereiche aufteilen. Zum einen soll er eine Benutzerverwaltung im klassischen Sinne durchführen können, was solche Teilbereiche wie *Benutzer anlegen*, *Benutzer löschen* und *Benutzer zu einer Gruppe hinzufügen* einschließt. Dabei soll insbesondere darauf geachtet werden, dass jedem Benutzer eine eindeutige Rolle zuzuordnen ist. Da in unserer Anwendung ein Administrator auch als ein Sicherheitsadministrator angesehen wird, ergibt sich daraus seine Zuständigkeit für die Verwaltung der Sicherheitspolitiken und des Benutzerwissens. Jedes Mal, wenn ein neuer Benutzer initialisiert wird, und falls es sich um einen Anfrager handelt, soll ihm ein eindeutiges Benutzerwissen

und eine eindeutige Sicherheitspolitik zugeordnet werden. Somit wird es ermöglicht eine Mehrbenutzervariante für CQE zu realisieren und unterschiedliche Sicherheitspolitiken für mehrere Benutzer zu verwalten, um auf diesen Grundmengen eine Inferenzkontrolle durchzuführen.
Weiterhin ist noch ein genauerer Mechanismus zur Verwaltung der Benutzerdaten zu erarbeiten. Es soll also die Entscheidung getroffen werden, ob man die Oracle-Datenbank oder die Technologien von Java dafür benutzt. In der weiteren Arbeit soll noch ein Konzept für die Datensynchronisierung aufgestellt werden. Da es in Mehrbenutzersystemen oft zu gleichzeitigen Objekt- und Datenzugriffen (z.B. ein gleichzeitiges Editieren des Benutzerwissens oder eine gleichzeitige Änderung der Zugangsdaten) kommt, ist die Behandlung solcher Fälle von großer Bedeutung, um mögliche Inkonsistenzen zu vermeiden.

16.4 Optimierung und Konsistenzsicherung

In dieser Iteration soll das Programm so optimiert werden, dass die eventuelle Inkonsistenz der Benutzerwissen- und der Pot_sec-Mengen nicht vorkommt. Außerdem sollen bei der Suche innerhalb der Log- oder Pot_sec-Mengen die semantischen Gleichheiten beachtet werden und nicht nur die syntaktischen.
Um den Aufwand zu reduzieren, soll eine Klasse von Situationen definiert werden, die beim Implikationstest den Theorembeweiser überspringen können. Ein einfacher Fall dafür ist z. B., wenn die zu überprüfenden Formeln nur aus Grundfakten bestehen.

16.5 Fehlerbehandlung

Folgende Anforderungen an die Fehlerberichterstattung werden gestellt:
Bei kontextbezogenen Fehlern innerhalb einer Anfrage sollen sinnvolle Fehlermeldungen ausgegeben werden. D.h. vor allem, dass bei einer nicht existierenden Relation oder einer falschen Stelligkeit innerhalb einer Anfrage der Benutzer genau darüber in Kenntnis gesetzt wird.
Die Fehler, die bei Prover9 und beim Parser auftreten, sollen behandelt werden. Es sollen alle Fehlermeldungen an den kritischen Stellen abgefangen und an die GUI weitergeleitet werden, wo sie dem Beutzer in einer informativen Form dargestellt werden. Dies soll mittels des Exception-Mechanismus von Java realisiert werden. Fehler werden also auf unterster Ebene innerhalb der Algorithmen geworfen und auf höheren Ebenen gefangen, um die Fehler in einer verständlichen Form, unter möglichst genauer Angabe der Fehlerart und -quelle, auszugeben.

16.6 Lügen- und kombinierte Methode

16.6.1 Lügenmethode

Bei Verwendung der Lügenmethode prüft der Lying_Censor, ob die korrekte Antwort auf die vom Benutzer gestellte Anfrage zusammen mit dem Benutzerwissen die Dis-

junktion aller potentiellen Geheimnisse abzuleiten ermöglicht. Sobald das der Fall ist, wird die gelogene Antwort ausgegeben. Die gelogene Antwort wird zum Benutzerwissen hinzugefügt. Sonst erhält der Benutzer die korrekte Antwort und diese wird zum Benutzerwissen hinzugefügt.

16.6.2 Kombinierte Methode

Bei Verwendung der kombinierten Methode prüft der Combined_Censor, ob die Antwort auf die vom Benutzer gestellte Anfrage zusammen mit dem Benutzerwissen ein potentielles Geheimnis aufdeckt und falls ja, dann muss der Combined_Censor unterscheiden, ob die Antwort gelogen oder verweigert werden muss. Im Falle, dass die Antwort gelogen wird, wird die gelogene Antwort ausgegeben und zum Benutzerwissen hinzugefügt. Falls die Antwort verweigert wird, wird *mum* ausgegeben und das Benutzerwissen bleibt unverändert. Sonst erhält der Benutzer die korrekte Antwort und diese wird zum Benutzerwissen hinzugefügt.

17 Entwurf

17.1 Negation in Anfragen

17.1.1 Problemstellung

Durch die Hinzunahme von Negationen in Anfragen wird die Mächtigkeit der Anfragesprache enorm gesteigert. Doch ist dies auch mit erheblichen neuen Schwierigkeiten verbunden. Durch eine Negation wird es möglich, *unsichere* bzw. *domain-abhängige* Anfragen zu formulieren. Die Teilergebnisse können dann, wenn man eine unendliche Domäne voraussetzt, unendlich groß werden. Im Allgemeinen will man keine unendlich großen (Zwischen-) Ergebnisse haben, da diese nicht verarbeitet werden können. Deshalb muss hier durch geeignete Einschränkungen Abhilfe verschafft werden.

17.1.2 Safe-Range Queries

Zur Lösung des oben beschriebenen Problems schlägt [AHV95] eine syntaktische Lösung vor. Es werden nur sog. *Safe-Range Queries* zugelassen. Diese sind immer domain-unabhängig und somit auch sicher. Im Folgenden soll ein Algorithmus vorgestellt werden, der Formeln, die in sog. *safe-range normal form (SRNF)* vorliegen, dahingehend überprüft, ob diese *safe range* und damit auch garantiert sicher sind. Dazu muss zunächst die SRNF erklärt werden.

Safe-Range normal form Damit sich eine Formel in SRNF befindet, müssen einige syntaktische Bedingungen erfüllt sein:

1. Keine Variable kommt sowohl frei als auch gebunden vor.
2. Es kommen keine Allquantoren vor.
3. Jede Implikationen $\psi \Rightarrow \xi$ muss durch $\neg\psi \vee \xi$ dargestellt werden. Analoges gilt für $\psi \Leftrightarrow \xi$.
4. Negationen dürfen nur vor Atomen oder vor Existenzquantoren stehen.

Die ersten drei Einschränkungen sind bereits durch die von uns definierten Sprachen erfüllt. Der Punkt 4 kann durch Anwendung der De Morgan'schen Regeln erfüllt werden.

Algorithmus (Range restriction (rr)) Eingabe: Eine Formel φ des Relationenkalküls in SRNF
Ausgabe: Eine Teilmenge der freien Variablen oder $false$

```
begin
case φ of
```

```
R(e_1,...,e_n)   : rr(φ) = Die Menge der Variablen in {e_1,...,e_n};
φ_1 ∧ φ_2        : rr(φ) = rr(φ_1) ∪ rr(φ_2);
φ_1 ∨ φ_2        : rr(φ) = rr(φ_1) ∩ rr(φ_2);
¬φ_1             : if rr(φ_1) ≠ false
                   then rr(φ) = ∅;
                   else return false
∃x⃗φ_1            : if x⃗ ⊆ rr(φ_1)
                   then rr(φ) = rr(φ_1) − x⃗
                   else return false
end case
end
```

Der Algorithmus liefert für Formeln in SRNF die Menge der freien Variablen zurück, die range restricted sind. Ist diese Menge gleich der Menge der freien Variablen, so ist die gesamte Formel *safe range*. Falls eine gebundene Variable nicht range restricted ist, so liefert der Algorithmus $false$.
Da wir in dieser Iteration nur geschlossene Anfragen betrachten, müssen wir lediglich zwischen den Ausgaben $false$ und $\emptyset$ unterscheiden. Wenn der Algorithmus für eine geschlossene Formel die leere Menge zurückgibt, so ist die Anfrage safe range. Falls die Negation vor einem Existenzquantor vorkommt, so muss in der Teilformel überprüft werden, ob die gebundenen Variablen *range restricted* sind.
Für unsere Anfragen soll eine etwas stärkere Einschränkung gelten, um die Übersetzung in SQL zu erleichtern. Diese Einschränkung sieht vor, dass negative Literale, die Variablen enthalten, immer unmittelbar konjugiert werden müssen mit positiven Literalen, die mindestens die entsprechenden Variablen ebenfalls enthalten.
Beispiel: $\exists X krankheit(X, armbruch) \wedge (\neg beinbruch(X))$

17.1.3 Syntaktische Umformungen

Rein syntaktisch waren Negationen auch in der vorherigen Iteration schon erlaubt. Entsprechende Anfragen wurden allerdings noch vom Programm zurückgewiesen. Die BNFs für die zulässigen Sprachen (und damit auch die Parser) bleiben also unberührt. Die Klasse Formula benötigt nun eine Methode, die die Formeln so umformt, dass Negationen nur noch unmittelbar vor Relationen oder Existenzquantoren stehen. Diese Methode kann relativ leicht implementiert werden. Über eine Tiefensuche durch den Syntaxbaum werden alle Knoten vom Typ NOT gesucht. Es können nun drei Fälle auftreten:

1. Der benachbarte EXP Knoten hat ein Kind: In diesem Fall handelt es sich um eine Relation, es muss daher nichts getan werden.

2. Der benachbarte EXP Knoten hat zwei Kinder: Es handelt sich um eine doppelte Negation. Die zwei EXP Knoten werden aus dem Baum entfernt (siehe Abbildung 20).

3. Der benachbarte EXP Knoten hat drei Kinder: Es handelt sich dabei um zwei

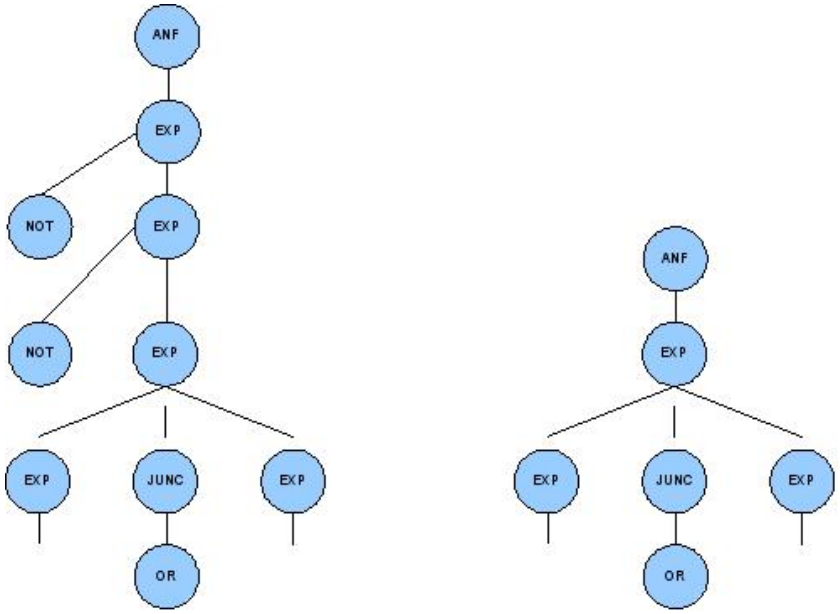

Abbildung 20: Syntaxbaum vor und nach Entfernung einer doppelten Negation

weitere EXP Knoten und einen Junktor. Der Junktor muss nach De Morgan in sein Komplement umgeformt werden und es werden jeweils zwischen dem linken und dem rechten Knoten vom Typ EXP ein neuer Knoten vom Typ EXP eingefügt. Diese erhalten zusätzlich als linkes Kind jeweils einen neuen Knoten vom Typ NOT. Danach können der alte NOT Knoten und sein Vater gelöscht werden (siehe Abbildung 21).

4. Sonst wird die Tiefensuche fortgesetzt. Eventuell vor Existenzquantoren auftretende Negationen werden hierbei ignoriert.

Benutzeranfragen Eine Benutzeranfrage muss nun folgende Stationen durchlaufen:

1. Syntaktische Prüfung.
2. Prüfung der semantischen Nebenbedingungen.
3. Umformung in SRNF (also lediglich Anwendung der De Morgan'schen Regeln).
4. Prüfung auf safe range.
5. Prüfung, ob alle negierten Literale mit Variablen unmittelbar konjunktivisch verknüpft sind mit einem passenden positiven Literal, das mindestens die gleichen Variablen enthält.
6. Umformung in SQL (s.u.).

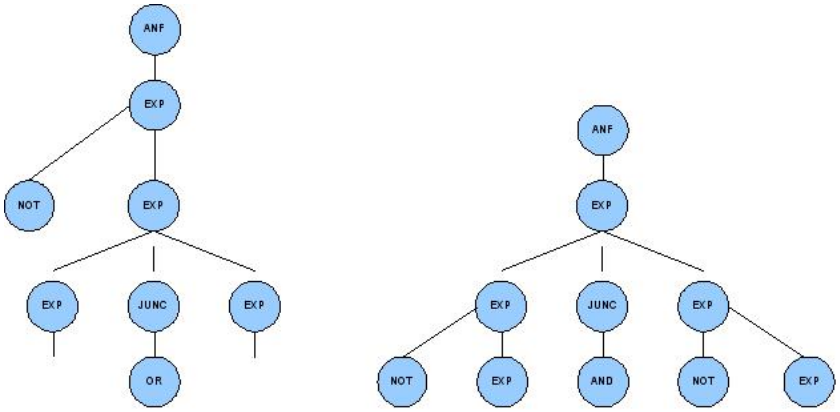

Abbildung 21: Syntaxbaum vor und nach Anwendung der De Morgan'schen Regeln

7. Weitere Behandlung gemäß Ablauf der kontrollierten Anfrageauswertung (siehe Kapitel 17.6.3).

Formeln aus dem Benutzerwissen Formeln aus dem Benutzerwissen können allquantifiziert sein. Um Formeln der Form $\forall x_1 \forall x_2 \ldots \forall x_n \varphi$ dahingehend überprüfen zu können, ob sie safe range sind, müssen diese zu $\neg\exists x_1 \exists x_2 \ldots \exists x_n \neg\varphi$ umgeformt und der oben genannte Algorithmus zum nach innen Ziehen der Negationen angewendet werden. Zur Sicherstellung der Entscheidbarkeit der Implikation der Formeln untereinander werden diese später wieder in ihrer ursprünglichen Form abgespeichert. Demnach durchlaufen Formeln aus dem Benutzerwissen nach Eingabe durch den Administrator folgende Stationen:

1. Syntaktische Prüfung.
2. Umformung in SRNF (u.U. Umkehrung der Quantoren und De Morgan'sche Regeln).
3. Prüfung auf safe range.
4. Prüfung, ob alle negierten Literale mit Variablen unmittelbar konjunktivisch verknüpft sind mit einem passenden positiven Literal, das mindestens die gleichen Variablen enthält.
5. Umformung in SQL (s.u.).
6. Weitere Behandlung gemäß Ablauf AddToLog (siehe Kapitel 17.6.4).

17.1.4 Übersetzung der Formeln in SQL-Anfragen

Nachdem nun die entscheidenden Einschränkungen bezüglich der zulässigen Formeln vorgestellt wurden, stellt sich die Frage, wie aus den zulässigen Formeln SQL-Anfragen

erzeugt werden. Bislang ging der Algorithmus zur Übersetzung der Formeln zu SQL implizit davon aus, dass alle vorkommenden Variablen (positiv) existenzquantifiziert sind. Nun sind grundsätzlich zwei Fälle voneinander zu unterscheiden:

- Alle Variablen sind positiv existenzquantifiziert, die Formeln haben also die Form $\exists x_1 \exists x_2 \ldots \exists x_n \varphi$.
  Dies kann bei Benutzeranfragen und bei Formeln aus dem Benutzerwissens auftreten.
- Alle Variablen sind negativ existenzquantifiziert, die Formeln haben also die Form $\neg\exists x_1 \exists x_2 \ldots \exists x_n \varphi$.
  Dies kann nur bei Formeln aus dem Benutzerwissen auftreten.

Es gilt wie bisher, dass eine Anfrage genau dann wahr ist, wenn es mindestens ein Tupel gibt, das die gewünschten Bedingungen erfüllt. Von daher reicht als Rahmen für Anfragen der ersten Art folgendes Konstrukt:

```
SELECT 'c0' FROM DUAL WHERE EXISTS (<zu konstruierender Ausdruck>);
```

Es wird eine Tabelle mit einem einzigen Eintrag $c0$ erzeugt, wenn der zu konstruierende Ausdruck ein nicht leeres Ergebnis liefert. Dabei muss $c0$ ein Konstantensymbol sein, das in der betrachteten Formel nicht auftaucht. Umgekehrt erfüllt der Rahmen

```
SELECT 'c0' FROM DUAL WHERE NOT EXISTS (<zu konstruierender Ausdruck>);
```

genau den Zweck für den zweiten Fall. Hier ist das Verhalten genau umgekehrt zu dem im ersten Fall. Exakt auf die gleiche Art und Weise können auch Literale behandelt werden, die keine Variablen enthalten. Diese werden, je nachdem, ob sie negiert oder nicht negiert sind, mit dem entsprechenden Rahmen versehen. Im Inneren dieses Rahmens steht eine Anfrage der Form

```
SELECT * FROM <Relationenname> WHERE <attribut1>=<Konst1> ...;
```

Hier werden über die WHERE-Klausel die Attribute an die gewünschten Konstanten gebunden. Bei Literalen, in denen Variablen auftreten, muss man sich die Tupel merken, die das Literal wahr machen. Ist das Literal nicht negiert, so wird es genau so behandelt, wie aus der ersten Iteration bekannt. Literale, die negiert sind und Variablen enthalten, müssen, wie oben beschrieben, immer unmittelbar mit einem positiven Literal konjugiert sein, das mindestens die gleichen Variablen enthält. Dieser Fall wird also als Einheit behandelt. Für das positive Literal wird ein äußerer und für das negative Literal ein innerer SQL-Ausdruck generiert. Das Ergebnis soll die Differenz zwischen dem positiven Literal und dem Verbund beider Literale sein.

Im äußeren Ausdruck werden über die WHERE-Klausel alle Attribute, die im Literal einen konstanten Wert haben, an die entsprechende Konstante gebunden und eine Projektion aller mit Variablen belegten Attribute ausgeführt. Ebenfalls über die WHERE-Klausel wird der Verbund abgezogen. Das sieht dann so aus:

SELECT <var_att_1> AS <var_1> ... <var_att_n> AS <var_n>
FROM <Relationenname außen> A
WHERE <$konst\_att_1$>=<$konst_1$> ... <$konst\_att_n$>=<$konst_n$> AND <var_1> NOT IN
(<innerer Ausdruck>);

Der innere Ausdruck sieht demnach so aus:

SELECT <var_1> FROM <Relationenname innen> B
WHERE B.<var_1>=A.<var_1> AND ... AND
B.<var_m>=A.<var_m> AND B.<$konst\_att_1$>=<$konst_1$> AND ...
AND <$konst\_att_n$>=<$konst_n$>)

Für den Teil „<var_i> NOT IN“ kann $<var_i>$ eine beliebige Variable aus dem negierten Literal sein. Wichtig ist dann, dass man im Teil „SELECT <var_i> FROM“ des inneren Ausdrucks für $<var_i>$ genau dieselbe Variable für die Projektion auswählt.

Beispiel Betrachte folgende (Teil-) Formel:

$$krankheit(X, Y) \wedge (\neg reha(X, dortmund, kurz, Y))$$

und die entsprechenden Relationenschemas:

```
krankheit:
Name        Type
----------  ----------
NAME        VARCHAR(20)
KRANKHEIT   VARCHAR(20)

reha:
Name        Type
----------  ----------
NAME        VARCHAR(20)
ORT         VARCHAR(20)
DAUER       VARCHAR(20)
KRANKHEIT   VARCHAR(20)
```

Der entsprechende SQL-Ausdruck würde folgendermaßen aussehen:

```
SELECT name AS X, krankheit AS Y
FROM krankheit k
```

```
WHERE k.name NOT IN
(SELECT name
FROM reha r
WHERE r.name=k.name AND r.krankheit=k.krankheit
AND r.ort='dortmund' AND r.dauer='kurz');
```

17.2 Lügenzensor und kombinierter Zensor

Bislang wurde nur ein Zensor – der Verweigerungszensor – implementiert. Jetzt sollen im Programm alle drei bekannten Methoden zur Zensur zur Verfügung stehen, also auch der Lügen- und der kombinierte Zensor. Dazu soll es fortan eine abstrakte Klasse *Censor* geben, die von drei Klassen *RefusalCensor, LyingCensor* und *CombinedCensor* jeweils passend implementiert wird. Da für jeden Zensor andere Vorbedingungen gelten, muss jeder Zensor eine Methode *isPrecondition* implementieren, die die entsprechenden Vorbedingungen prüft. Also sind die Methoden *classify* und *isPrecondition* jeweils, gemäß den Anforderungen an den jeweiligen Zensor, anzupassen. Die Eigenschaften der beiden neu hinzugekommenen Zensoren werden im folgenden erläutert.

17.2.1 Lügenmethode

Für die Lügenmethode muss die folgende Bedingung stets erfüllt sein:

$$log \not\models \bigvee_{\Psi \in\ pot\_sec} \Psi.$$

pot_sec bezeichnet die Menge der potentiellen Geheimnisse und *log* ist das Benutzervorwissen.
Wenn der Benutzer eine Anfrage stellt, prüft der Lügenzensor, ob die entsprechende Antwort zusammen mit dem Benutzerwissen die Disjunktion der potentiellen Geheimnisse abzuleiten ermöglicht. Sobald das der Fall ist, wird die Negation der korrekten Antwort ausgegeben. Die Negation der korrekten Antwort wird zum Benutzerwissen hinzugefügt. Sonst erhält der Benutzer die korrekte Antwort und diese wird zum Benutzerwissen hinzugefügt. Um sicherzustellen, dass die Vorbedingung immer erfüllt ist, wird bei jedem Hinzufügen zur Menge des Benutzerwissens oder der potentiellen Geheimnisse die Methode *isPrecondition* aus der Klasse *LyingCensor* aufgerufen. In der Klasse *LyingCensor* (siehe Kapitel 17.6.1) wird *isPrecondition* implementiert und für die Überprüfung der Vorbedingung im Rahmen des Lügenzensors angepasst. Beim Aufruf erhält *isPrecondition* als Übergabeparameter das Benutzerwissen und die Menge der potentiellen Geheimnisse. Innerhalb von *isPrecondition* wird die Methode *isImplied* der Klasse *Prover9* mit den Übergabeparametern von *isPrecondition* aufgerufen.

Falls *isImplied* den Rückgabewert *true* liefert, wird *isPrecondition false* zurückliefern, d.h., dass man potentielle Geheimnisse aus dem vorhandenen Benutzerwissen ableiten kann, was unerwünscht ist. Deswegen wird an den Administrator eine Fehlermeldung ausgegeben. Falls *isImplied* den Rückgabewert *false* liefert, wird *isPrecondition* den Rückgabewert *true* zurückliefern. Das heißt, es wird kein potentielles Geheimnis vom Benutzerwissen impliziert.

Wenn der Benutzer Anfragen an das System stellt, wird die Methode *classify* (die die entsprechende Methode aus der Klasse *Censor* implementiert) des *LyingCensor* aufgerufen. Beim Aufruf erhält die Methode alle Übergabeparameter, welche dem Benutzer des Programms zugeordnet sind. Dies sind das Benutzerwissen *log*, die Menge

der potentiellen Geheimnisse *pot_sec* und die korrekte Antwort $eval^*(\Phi)(db)$. Da es sich um einen Lügenzensor handelt, müssen wir sicherstellen, dass die folgende Bedingung erhalten bleibt:

$$log \cup \{eval^*(\Phi)(db)\} \not\models \bigvee_{\Psi \in pot\_sec} \Psi.$$

Um dies zu überprüfen, wird die Methode *isImplied* der Klasse *Prover9* mit den Übergabeparametern von *classify* aufgerufen. In dieser Methode werden folgende Schritte ausgeführt:

- Sie ruft als erstes die Methode *translate* des *Prover9* auf, damit die Übergabeparameter in ein gültiges Eingabeformat des Theorembeweisers überführt werden. Diese Eingabe wird von der Methode als String zurückgegeben und anschließend als Inputdatei abgelegt.
- Danach wird die Methode *prove* des *Prover9* aufgerufen, in der der Theorembeweiser zum Überprüfen der Implikation aufgerufen wird. Abhängig von dessen Ausgabe gibt diese Methode *true* zurück, falls die Implikation wahr ist, und im anderen Fall *false*.

Wenn die Methode *isImplied* den Wert *true* zurückliefert, so ist die Anfrage für uns kritisch und *classify* muss den Rückgabewert LIE zurückgeben. Dann wird die Methode *modify* der Klasse *Modificator* aufgerufen, um die Antwort, gemäß dem Rückgabewert LIE, zu modifizieren. Nach der erfolgten Modifikation wird die Methode *AddToLog* der Klasse *UserData* aufgerufen und die Negation der korrekten Antwort zum Benutzerwissen *log* hinzugefügt. Außerdem wird auch die Methode *cqe* der Klasse *Application* aufgerufen und die entsprechend modifizierte Antwort (die Negation der korrekten Antwort) an den Benutzer ausgegeben.
Wenn die Methode *isImplied* den Wert *false* zurückliefert, so ist die Anfrage für uns unkritisch und *classify* muss den Rückgabewert $UNCRITICAL$ zurückgeben. Dann wird die Methode *modify* der Klasse *Modificator* aufgerufen, um die Antwort gemäß dem Rückgabewert $UNCRITICAL$ zu modifizieren. Nach der erfolgten Modifikation wird die Methode *AddToLog* der Klasse *UserData* aufgerufen, und die korrekte Antwort zum Benutzerwissen *log* hinzugefügt. Die Methode *cqe* der Klasse *Application* wird die entsprechend modifizierte Antwort (die korrekte Antwort) an den Benutzer ausgeben.

17.2.2 Kombinierte Methode

Für die kombinierte Methode muss die folgende Bedingung stets erfüllt sein:

$$\forall \Psi \in pot\_sec : log \not\models \Psi\,.$$

wobei $\Psi \in pot\_sec$, und *pot_sec* die Menge der potentiellen Geheimnisse bezeichnet und *log* das Benutzervorwissen ist.

Wenn der Benutzer eine Anfrage stellt, prüft der kombinierte Zensor, ob die entsprechende Antwort zusammen mit dem Benutzerwissen ein potentielles Geheimnis aufdeckt, und falls ja, dann muss der kombinierte Zensor unterscheiden, ob die Antwort gelogen oder abgelehnt werden muss. Im Falle, dass die Antwort gelogen wird, wird die Negation der korrekten Antwort ausgegeben. Die Negation der korrekten Antwort wird zum Benutzerwissen hinzugefügt. Falls die Antwort abgelehnt wird, wird die Anfrage abgelehnt und das Benutzerwissen bleibt unverändert. Sonst erhält der Benutzer die korrekte Antwort und diese wird zum Benutzerwissen hinzugefügt.

Um sicherzustellen, dass die Vorbedingung immer erfüllt ist, wird beim Hinzufügen zur Menge des Benutzerwissens oder der potentiellen Geheimnisse die Methode *isPrecondition* aus der Klasse *CombinedCensor* aufgerufen. In der Klasse *CombinedCensor* (siehe Kapitel 17.6.1) wird *isPrecondition* implementiert und für die Überprüfung der Vorbedingung im Rahmen des kombinierten Zensors angepasst. Beim Aufruf erhält *isPrecondition* als Übergabeparameter das Benutzerwissen und die Menge der potentiellen Geheimnisse. Innerhalb von *isPrecondition* wird die Methode *isImplied* der Klasse *Prover9* mit den Übergabeparametern von *isPrecondition* aufgerufen.

Falls *isImplied* den Rückgabewert *true* liefert, wird *isPrecondition false* zurückliefern, d.h., dass man potentielle Geheimnisse aus dem vorhandenen Benutzerwissen ableiten kann, was unerwünscht ist. Deswegen wird an den Administrator eine Fehlermeldung ausgegeben. Falls *isImplied* den Rückgabewert *false* liefert, wird *isPrecondition* den Rückgabewert *true* zurückliefern. Das heißt, es wird kein potentielles Geheimnis vom Benutzerwissen impliziert.

Wenn der Benutzer Anfragen an das System stellt, wird die Methode *classify* (die die entsprechende Methode aus der Klasse *Censor* implementiert) des *CombinedCensor* aufgerufen. Beim Aufruf erhält die Methode alle Übergabeparameter, welche dem Benutzer des Programms zugeordnet sind. Dies sind das Benutzerwissen log, die Menge der potentiellen Geheimnisse pot_sec und die korrekte Antwort $eval^*(\Phi)(db)$. Da es sich um einen kombinierten Zensor handelt, müssen wir sicherstellen, dass die folgende Bedingung nicht wahr ist:

$$\exists \Psi_1 \in pot\_sec : log \cup \{eval^*(\Phi)(db)\} \models \Psi_1 \text{ und}$$
$$\exists \Psi_2 \in pot\_sec : log \cup \{\neg eval^*(\Phi)(db)\} \models \Psi_2.$$

Um dies sicherzustellen, muss zunächst der linke Teil der Bedingung (bzgl. des und's) überprüft werden. Dazu wird die Methode *isImplied* der Klasse *Prover9* mit den Übergabeparametern von *classify* aufgerufen. Diese durchläuft nun intern eine Schleife, in der für alle übergebenen potentiellen Geheimnisse folgende Schritte ausgeführt werden:

- Sie ruft als erstes die Methode *translate* des *Prover9* auf, damit die Übergabeparameter in ein gültiges Eingabeformat des Theorembeweisers überführt werden. Diese Eingabe wird von der Methode als String zurückgegeben und anschließend als Inputdatei abgelegt.
- Danach wird die Methode *prove* des *Prover9* aufgerufen, in der der Theorembe-

weiser zum Überprüfen der Implikation aufgerufen wird. Abhängig von dessen Ausgabe gibt diese Methode *true* zurück, falls die Implikation wahr ist, und im anderen Fall *false*.

Wenn die Methode *isImplied* für den linken Teil den Wert *true* zurückliefert, so ist die Anfrage für uns kritisch und wir müssen den rechten Teil der Bedingung überprüfen.

Falls die Methode *isImplied* auch für den rechten Teil *true* zurückliefert, dann muss *classify* den Rückgabewert MUM zurückgeben. Dann wird die Methode *modify* der Klasse *Modificator* aufgerufen, um die Antwort gemäß dem Rückgabewert MUM zu modifizieren. Nach der erfolgten Modifikation an den Benutzer wird ausgegeben, dass seine Anfrage abgelehnt wurde. Es wird nichts zum Benutzerwissen hinzugefügt.

Falls *isImplied* für den rechten Teil der Bedingung *false* zurückliefert, dann muss *classify* den Rückgabewert LIE zurückgeben. Dann wird die Methode *modify* der Klasse *Modificator* aufgerufen, um die Antwort gemäß dem Rückgabewert LIE zu modifizieren. Nach der erfolgten wird Modifikation die Methode *AddToLog* der Klasse *UserData* aufgerufen, und die Negation der korrekten Antwort zum Benutzerwissen *log* hinzugefügt. Die Methode *cqe* der Klasse *Application* wird die modifizierte Antwort an den Benutzer ausgegeben.
Ansonsten liefert die Methode *isImplied* für den linken Teil den Wert *false*, d.h., dass die Anfrage für uns unkritisch ist. Die Methode *classify* wird den Rückgabewert $UNCRITICAL$ zurückgeben. Dann wird die Methode *modify* der Klasse *Modificator* aufgerufen, um die Antwort gemäß dem Rückgabewert $UNCRITICAL$ zu modifizieren. Nach der erfolgten Modifikation wird die Methode *AddToLog* der Klasse *UserData* aufgerufen und die korrekte Antwort zum Benutzerwissen *log* hinzugefügt. Die Methode *cqe* der Klasse *Application* wird die entsprechend modifizierte Antwort (die korrekte Antwort) an den Benutzer ausgeben.

17.3 Optimierung

17.3.1 Einleitung

Das Ziel der Optimierung ist es, die Redundanzfreiheit und die Minimalität der Daten innerhalb der log- und pot_sec-Menge zu sichern. Die Minimalität, die wir hier meinen, heißt die log- und pot_sec-Menge **so klein wie möglich** zu halten, damit kein einziges Element von der Menge entfernt werden kann, ohne dass die ganze Menge ihre semantischen Sinn verliert. Besser gesagt, wenn man von einer minimalen log- bzw. pot_sec-Menge ein einziges Element entfernt, wird das Programm bei manchen Anfragen unerwünschte Ausgaben liefern.
Wie diese Minimalität gesichert wird, erklären wir getrennt für log und pot_sec.

17.3.2 Minimalisierung bei der log-Menge

Es ist zu beachten, dass wir die Konjunktionen innerhalb der log-Menge auflösen und auf diese Weise die ganze Menge normalisieren. D. h., wenn wir in der nicht optimierten log-Menge zum Beispiel die Formeln $\alpha_1 \wedge \alpha_2$ bzw. $(\forall x)[\alpha_1(x) \wedge \alpha_2(x)]$ haben, wird in der optimierten Menge $\{\alpha_1,\ \alpha_2\}$ bzw. $\{\forall x[\alpha_1(x)],\ \forall x[\alpha_1(x)]\}$ gespeichert.
Im Folgenden werden wir die optimierte log-Menge mit $\widetilde{log}$ bezeichnen und das neue Wissen, das wir in die Menge (log) einfügen möchten, mit α. Die optimierte log-Menge wird so minimalisiert, dass die folgenden beiden Eigenschaften immer gelten:

1) $\widetilde{log} \models log \cup \{\alpha\}$ und $log \cup \{\alpha\} \models \widetilde{log}$ [Äquivalenz]

2) Für alle $\varphi \in \widetilde{log}$: $\widetilde{log} \backslash \{\varphi\} \not\models \widetilde{log}$ [Minimalität]

Beim Hinzufügen des neuen Wissens (α) testen wir erstmal, ob $log \models \alpha$ gilt. Wenn das der Fall ist, wird α nicht eingefügt. D. h. es gilt:

$$\widetilde{log} := log$$

Andernfalls wird α in die log-Menge eingefügt ($\widetilde{log} := log \cup \{\alpha\}$), aber danach muss noch überprüft werden, ob die oben genannte Minimalitäts-Eigenschaft für $\widetilde{log}$ gilt. Dazu muss für alle Elemente (φ) aus $\widetilde{log}$ getestet werden, ob $\widetilde{log} \backslash \{\varphi\} \models \varphi$ gilt. Falls dies der Fall ist, dann wird dieses φ aus $\widetilde{log}$ entfernt. D.h. es gilt:

$$\widetilde{log} := \widetilde{log} \backslash \{\varphi\}$$

Und falls so ein φ nicht existiert, bleibt $\widetilde{log}$ unverändert.
Es ist offensichtlich, dass für alle genannten Fälle die beiden Eigenschaften (Äquivalenz und Minimalität) gelten.

17.3.3 Minimalisierung der pot_sec-Menge für den Lügenzensor

Die pot_sec-Menge wird auch normalisiert, und zwar so, dass wir die Disjunktionen innerhalb der Menge auflösen. D. h. die Formeln der Form $\psi_1 \vee \psi_2$ bzw. $(\exists x)[\psi_1(x) \vee$

$\psi_2(x)]$ werden innerhalb der pot_sec-Menge als $\{\psi_1, \psi_2\}$ bzw. $\{\exists x[\psi_1(x)], \exists x[\psi_2(x)]\}$ gespeichert.
Wir bezeichnen die optimierte pot_sec-Menge mit $\widetilde{pot\_sec}$ und das neue Geheimnis, das wir in pot_sec-Menge einfügen möchten, mit ψ_{neu}. Die pot_sec-Menge muss so minimalisiert werden, dass die beiden unteren Eigenschaften gelten:

1) $\bigvee \widetilde{pot\_sec} \models \bigvee pot\_sec \vee \psi_{neu}$ und $\bigvee pot\_sec \vee \psi_{neu} \models \bigvee \widetilde{pot\_sec}$
[Disjunktionsäquivalenz]

2) Für alle $\psi \in \widetilde{pot\_sec}: \bigvee \widetilde{pot\_sec} \not\models \bigvee \widetilde{pot\_sec} \backslash \{\psi\}$ [Minimalität]

Beim Hinzufügen des neuen Geheimnisses in die pot_sec-Menge wird erstmal überprüft, ob $\bigvee pot\_sec \vee \psi_{neu} \models \bigvee pot\_sec$ gilt. Falls es der Fall ist, wird das neue Geheimnis nicht eingefügt. Es gilt nämlich:

$$\widetilde{pot\_sec} := pot\_sec$$

Andernfalls wird das neue Geheimnis eingefügt ($\widetilde{pot\_sec} := pot\_sec \cup \{\psi_{neu}\}$). Es muss aber noch überprüft werden, ob eine Minimalisierung der Menge notwendig ist. Dazu muss für alle Elemente ψ aus $\widetilde{pot\_sec}$ überprüft werden, ob $\bigvee \widetilde{pot\_sec} \models \bigvee \widetilde{pot\_sec} \backslash \{\psi\}$ gilt. Falls es der Fall ist, wird das betroffene Element aus $\widetilde{pot\_sec}$ entfernt. D. h. es gilt:

$$\widetilde{pot\_sec} := \widetilde{pot\_sec} \backslash \{\psi\}$$

Wenn es nicht der Fall ist, bleibt $\widetilde{pot\_sec}$ unverändert.
Für alle drei obigen Fälle gelten die beiden genannten Eigenschaften (Disjunktionsäquivalenz und Minimalität).

17.3.4 Minimalisierung der pot_sec-Menge für den Verweigerungszensor und den kombinierten Zensor

Wir bezeichnen weiterhin die optimierte pot_sec-Menge mit $\widetilde{pot\_sec}$ und das neue Geheimnis mit ψ_{neu}. Da diese Zensoren im Gegensatz zum Lügenzensor die Disjunktionen nicht berücksichtigen, ist die Äquivalenz-Eigenschaft anders als beim letzten Fall.
Die pot_sec-Menge muss in diesem Fall so minimalisiert werden, dass außer einer Minimalitätseigenschaft auch die folgende Bedingung für ein beliebiges $\chi \in log \cup \{eval^*(\Phi)(db)\}$ oder $\chi \in log \cup \{\neg eval^*(\Phi)(db)\}$ gilt:

$\exists \psi \in pot\_sec: \chi \models \psi$ gdw. $\exists \psi \in \widetilde{pot\_sec}: \chi \models \psi$ [Semantiktreue]

Beim Hinzufügen des neuen Geheimnisses in die pot_sec-Menge wird erstmal überprüft, ob $\psi_{neu} \models \psi$, für ein $\psi \in pot\_sec$ gilt. Falls es der Fall ist, wird das neue Geheimnis nicht eingefügt (*). Es gilt nämlich:

$$\widetilde{pot\_sec} := pot\_sec$$

Andernfalls wird das neue Geheimnis eingefügt ($\widetilde{pot\_sec} := pot\_sec \cup \{\psi_{neu}\}$). Es muss aber überprüft werden, ob eine Minimalisierung der pot_sec-Menge notwendig ist. Dazu muss für alle Elemente ψ aus $\widetilde{pot\_sec}$ überprüft werden, ob $\psi \models \psi_{neu}$ gilt. Falls es der Fall ist, wird das betroffene Element aus $\widetilde{pot\_sec}$ entfernt (**). D. h. es gilt:

$$\widetilde{pot\_sec} := \widetilde{pot\_sec} \backslash \{\psi\}$$

Und wenn es nicht der Fall ist, bleibt $\widetilde{pot\_sec}$ unverändert.

Wir wollen jetzt zeigen, dass die Semantiktreue-Eigenschaft für unsere optimierte pot_sec-Menge ($\widetilde{pot\_sec}$) gültig ist. Wir bezeichnen die nicht optimierte pot_sec-Menge mit pot_sec'.

Für den Fall (*) gilt:

$$\widetilde{pot\_sec} = pot\_sec$$

und

$$pot\_sec' = pot\_sec \cup \{\psi_{neu}\}$$

Wir wollen zeigen:
$\exists \psi \in pot\_sec' : \ \chi \models \psi$ gdw. $\exists \psi \in \widetilde{pot\_sec} : \ \chi \models \psi$

" $\Rightarrow$ "
Wenn $\psi \in \widetilde{pot\_sec}$ gilt, dann gilt auch die Behauptung ganz offensichtlich. (Da $\widetilde{pot\_sec} \subseteq pot\_sec'$ gilt.) Ansonsten nehmen wir an, dass $\chi \models \psi_{neu}$. Wir wissen, dass ψ_{neu} ein bereits existierendes ψ aus $\widetilde{pot\_sec}$ impliziert. Daraus folgt, dass $\chi \models \psi$ für ein ψ aus $\widetilde{pot\_sec}$ gilt.

" $\Leftarrow$ "
trivial

Für den Fall (**) gilt:

$$\exists \psi' \in pot\_sec \ : \ \psi' \models \psi_{neu}$$

$$\widetilde{pot\_sec} = (pot\_sec \setminus \{\psi'\}) \cup \{\psi_{neu}\}$$

$$pot\_sec' = pot\_sec \cup \{\psi_{neu}\}$$

Wenn $\psi \in \widetilde{pot\_sec}$, dann gilt die Behauptung offensichtlich. Ansonsten:

" $\Rightarrow$ "
Wir nehmen an, dass $\chi \models \psi'$.
Da $\psi' \models \psi_{neu}$ gilt, gilt auch $\chi \models \psi_{neu}$.

" $\Leftarrow$ "
trivial

17.4 Fehlerbehandlungskonzept

17.4.1 Einführung in Java Exceptions

Exception Handling bedeutet die Behandlung von Ausnahmen in Java. Mit Ausnahmen sind Fehlersituationen gemeint, die in einem Programm während der Ausführung entstehen können und auf die geeignet reagiert werden muss. Es gibt tatsächlich Methoden in Java, die fehlschlagen können. Es kann zu Fehlersituationen kommen, die nicht vorhersehbar sind.

- *try - catch* Anweisung
  Die *try-catch* Anweisung besteht aus zwei Anweisungsblöcken. Der erste Anweisungsblock folgt dem Schlüsselwort *try*, der zweite dem Schlüsselwort *catch*. In den ersten Anweisungsblock hinter *try* fügt man die eigentlichen Anweisungen ein, die ausgeführt werden sollen. Sie erstellen Objekte, greifen über Referenzvariablen auf diese zu, rufen Methoden auf und so weiter. Die Standard-Reaktion, wenn Ausnahmesituationen eintreten, sieht so aus, dass die Anwendung abgebrochen wird. Also wird die Ausnahme von der *catch* Anweisung gefangen.

- *benutzerdefinierte Exceptions*
  Wenn man selber Exceptions erstellen möchte, um zum Beispiel eine spezielle Fehlerbehandlung für seine eigenen Klassen zu implementieren, muss man seine neuen Exceptions in die Java-Hierarchie einordnen. Diese sollte allerdings entweder von Exception oder einer ihrer Subklassen abgeleitet werden. Damit werden ihre Klassen ebenfalls als Fehlerobjekte für benutzerdefinierte Ausnahmen gekennzeichnet. Das folgende Beispiel geht diesen Weg und definiert eine neue Ausnahme. Diese leiten wir gleich von Exception ab und überschreiben sowohl den Standardkonstruktor als auch die Variante, welche einen String übernimmt.

```
class MyException extends Exception
{
  public MyException()
  {
}
  public MyException(String s)
  {
    super(s);
  }
}
public class MyClass
{
  public static void main(String[] args)
  {
    try
    {
      throw new MyException("Fehler aufgetreten...");
```

```
    }
    catch(MyException e)
    {
      System.out.print(e.getMessage());
    }
  }
}
```

Ausgabe:

```
Fehler aufgetreten...
```

- Exceptions auslösen
  Um nun eine Exception auslösen zu können, verwendet man den *throw*-Befehl. In dessen syntaktischer Ausführung wird mit dem *new* Befehl eine neue Instanz der Fehlerklasse erzeugt. Dieser kann man dann, je nach Konstruktor, bestimmte Parameter übergeben. Nachdem die Ausnahme ausgelöst wurde, wird sie in einer passenden Klausel gefangen.

  Als Beispiel sei hier eine Exceptionklasse *CqeSQLException* definiert, die von der Oberklasse *Exception* erbt. Diese besitzt zwei Methoden zur Fehlerbehandlung, hier dient die erste dazu, dem Entwickler einen Hinweis über die Java-Konsole auszugeben, und die zweite gibt dem Benutzer eine Fehlermeldung über die GUI aus.

```
class CqeSQLException extends Exception
{
  public CqeSQLException()
  {
}
  public CqeSQLException(boolean s)
  {
  System.out.print("Stelligkeit ist falsch!")
 }
  public CqeSQLException(String s)
  {
  JOptionPane.showMessageDialog(application.cqe(
      txtQuery.getText()), "Ergebnis",
          JOptionPane.INFORMATION_MESSAGE);
 }
}

public static void main(String[] args) throws CqeSQLException
{
....
try{
```

```
...
}catch(SQLException s)
{
throw new CqeSQLException(1);
}
}
```

17.4.2 Fehlerbehandlung

Die folgenden Abschnitte in unserem Programmcode werden mit der Behandlung oder der Weitergabe von Ausnahmen versehen:

Stellen mit umfangreichen Fehlerbehandlungen Die Klassen stellen vorhandene Methoden bereit, in denen unterschiedliche Fehler auftreten können, die mit verschiedenen Exceptionklassen behandelt werden. Beispielhaft könnte eine Fehlerbehandlung durch die Exceptionklassen *IllegalanfrageException*, *CqeSQLException* und *IOException* in den jeweiligen Methoden vorgenommen werden.

- Methode *cqe(query:String):String*:
  Diese Methode wird von der GUI aufgerufen, hier könnte die Exception IllegalanfrageException eingesetzt werden.
- Methode *isValid(Syntaxbaum: TreeNode): boolean*:
  Diese Methode wird von cqe aufgerufen, hier könnte die Exception CqeSQLException eingesetzt werden.
- Methode *sqe(query: String): boolean*:
  Diese Methode wird von cqe aufgerufen, hier könnte die Exception IllegalanfrageException eingesetzt werden.
- Methode *addToLog(expr: String):Integer*
  Diese Methode wird von der GUI aufgerufen, hier könnte die Exception CqeSQLException eingesetzt werden.
- Methode *addToPot_Sec(expr: String): Integer*
  Diese Methode wird von der GUI aufgerufen, hier könnte die Exception CqeSQLException eingesetzt werden.
- Methode *deleteFromLog(index: Integer)*
  Diese Methode wird von der GUI aufgerufen, hier könnte die Exception CqeSQLException eingesetzt werden.
- Methode *deleteFromPot_Sec(index: Integer)*
  Diese Methode wird von der GUI aufgerufen, hier könnte die Exception CqeSQLException eingesetzt werden.

- Die Methode *isImlied(log:LinkedList, query:Formula, pot_ sec: LinkedList): boolean*
  Diese Methode wird von RefusalCensor aufgerufen, hier könnte die Exception IOException eingesetzt werden.

17.5 Benutzertrennung

17.5.1 Benutzerdatenverwaltung

In der zweiten Iteration des Projektes ist die Speicherung und Verwaltung der Benutzerdaten modifiziert worden. Alle benutzerrelevanten Daten werden in der Datenbank gespeichert. Das heißt Login, Passwort, Zensor und Rolle werden je Benutzer in der DB festgehalten. Dies bietet den Vorteil der Synchronisation der Administratorzugriffe auf die benutzerbezogenen Daten. Jedem Benutzer ist außerdem eine Tabelle für *log*-Formeln und eine Tabelle für *pot_sec* zuzuordnen (konkrete Schemas für diese Tabellen sind im Abschnitt 3.1 dieses Dokumentes angegeben). In diesen Tabellen sind also die Java Objekte vom Typ Formula zu speichern. Falls es nicht möglich ist, in die Oracle-Datenbank die nicht primitiven Datentypen von Java einzubringen, soll evtl. eine Serialisierung dieser Objekte vorgenommen werden. In diesem Fall sollen vor einer Einfügeoperation die Formula-Objekte in eine serialisierte Form gebracht werden und anschließend in die Tabelle eingefügt werden. Bei einer Datenbankanfrage werden die serialisierten Formula-Objekte aus der Tabelle ausgelesen und wiederhergestellt, was eine weitere Bearbeitung dieser Daten im Programm ermöglicht. Die Klasse UserData verfügt über die Attribute *login*, *password*, *role*, *censor*, *log* und *pot_sec*. Die Attribute *log* und *pot_sec*, die die verketteten Listen von Formula-Objekten für die Mengen *log* und *pot_sec* darstellen, bilden eine lokale Kopie der in der DB enthaltenen Daten. Die lokale Kopie wird benötigt, um einen Vergleich durchführen zu können, ob diese Daten sich verändert haben. Solcher Vergleich wird z.B. beim Aufruf der Methode *lock* (genauere Beschreibung der Methode *lock* siehe Abschnitt 17.5.3) stattfinden. Der Abruf der Benutzerdaten von der DB wird bei jedem Aufruf der Methoden *getFromLog* und *getFromPotSec* stattfinden. Dabei wird gesichert, dass im Programm ein aktueller Stand der Daten und die Konsistenz zwischen Datenbank und Anwendung gewährleistet sind.

17.5.2 Sperrmechanismus

Die Bearbeitung der Benutzerdaten erfolgt mit Hilfe der graphischen Benutzeroberfläche und schließt unmittelbar direkte Datenbankzugriffe mit ein, die auch von mehreren Administratoren gleichzeitig erfolgen können. Wir haben also mit dem gleichzeitigen Zugriff auf gemeinsame Ressourcen (DB-Tabellen) zu tun und wollen Dateninkonsistenzen vermeiden, die ohne eine explizite Behandlung entstehen können. Dafür nutzen wir das Konzept der Oracle-Datenbank, die einige Sperrmechanismen zur Verfügung stellt.
Mit Hilfe von solchen Sperrmechanismen (Lockingmethoden) dürfen Transaktionen Objekte für sich reservieren und vor konkurrierenden Zugriffen schützen. Mit Hilfe von Element-Sperren kann Konflikt-Serialisierbarkeit erzwungen werden. Transaktionen dürfen bei diesem Verfahren Datenbankobjekte für eine bestimmte Zeit für sich reservieren. Vor dem Zugriff auf ein Objekt wird ein Lock (Sperre) für dieses angefordert. Nach Beendigung des Zugriffs wird der Lock wieder freigegeben. Ein jedes Datenbanksystem hat eine selbstständige Lockverwaltung, die das Setzen und das

Freigeben der Locks für beliebig viele Transaktionen steuert. In diesem Beispiel wird ein Datenverlust dargestellt.

	B_1	B_2
1	SELECT * FROM LOG	
2		SELECT * FROM LOG
3	INSERT,UPDATE oder DELETE	
4		INSERT,UPDATE oder DELETE
5	COMMIT	
6		COMMIT

Seien B1 und B2 zwei Benutzer. Beide Benutzer wollen die Log-Tabelle öffnen und editieren. B1 öffnet die Log-Tabelle zum Zeitpunkt t=1, B2 zum Zeitpunkt t=2 und sehen die Daten der folgenden Tabelle.

Name	Krankheit
Hans	Armbruch
Peter	Beinbruch

B1 ersetzt den ersten Eintrag der Spalte Name mit dem Wert *Hans* durch den Wert *Müller*. B2 editiert nun auch die gleiche Tabelle. B2 will den ersten Eintrag der Spalte Name mit dem Wert *Hans* durch den Wert *Kim* ersetzen. Führt B2 nun die Transaktion durch, so wird dieser Vorgang zurückgewiesen. Auf diese Weise kann bei gleichzeitigem Datenzugriff ein Datenverlust folgen. Die Art der entstandenen Dateninkonsistenz wird hier als Datenverlustproblem bezeichnet.

Erweiterter Sperrmechanismus Das Datenverlustproblem besteht darin, dass B2 die Änderung von der Log-Tabelle nicht kennt. Wenn wir vor der Sperre überprüfen, ob die Tabelle von einem anderem Benutzer geändert worden ist, dann verfahren wir wie folgt: Wenn es nicht der Fall ist, sperren wir sofort die Tabelle, so wird das Datenverlustproblem gelöst. Ein Beispiel sieht so aus:

	B_1	B_2
1	SELECT * FROM LOG	
2		SELECT * FROM LOG
3	SELECT * FROM LOG WHERE Name=Hans FOR UPDATE NOWAIT	
4	INSERT, UPDATE oder DELETE	
5	COMMIT	
6		SELECT * FROM LOG WHERE Name=Hans FOR UPDATE NOWAIT
7		Zwei Alternativen

SELECT...FOR UPDATE ermöglicht, die Sperre direkt nach der Überprüfung aus-

zuführen. Die zwei Alternativen sind:

- Wenn die von B2 benötigten Daten nicht geändert werden, also die Schritte 3 und 4 nicht ausgeführt werden, werden alle Zeilen in Log ausgegeben, welche Name Hans sind.
- Wenn ein anderer Benutzer die von B2 benötigten Daten reserviert, also der Schritt 5 nicht ausgeführt wird, wird eine Fehlermeldung ausgegeben.

17.5.3 Methode lock

Die Methode *lock* dient zum Reservieren von Tabellen und gehört zur Klasse Application. Wenn man eine Zeile von der LOG-Tabelle editieren möchte, soll zuerst die aktuelle Zeile mit der in der Datenbank gespeicherten Zeile verglichen werden. Die Aufgaben der Methode sind das Überprüfen der Daten und das Reservieren der Tabelle.

Struktur der Tabellen Benutzertabellen

Formel
armbruch(Hans);
beinbruch(Peter);

Username	Password	Rolle
zhang	123	user
yu	123	admin

Die entsprechende Methode ist:

public boolean lock(String tabellename);

Die Methode führt den SQL-Befehl
LOCK TABLE *tabellenname*
aus. Wenn die Tabelle reserviert werden kann, wird *true* ausgegeben, ansonsten $false$.

17.5.4 Methode unlock

public void unlock()
Nach der Durchführung der DML-Befehle soll die verarbeitete Tabelle freigegeben

werden. In der Methode soll nur ein SQL-Befehl COMMIT an die Datenbank übergeben werden, damit andere Benutzer die Tabelle verarbeiten können.

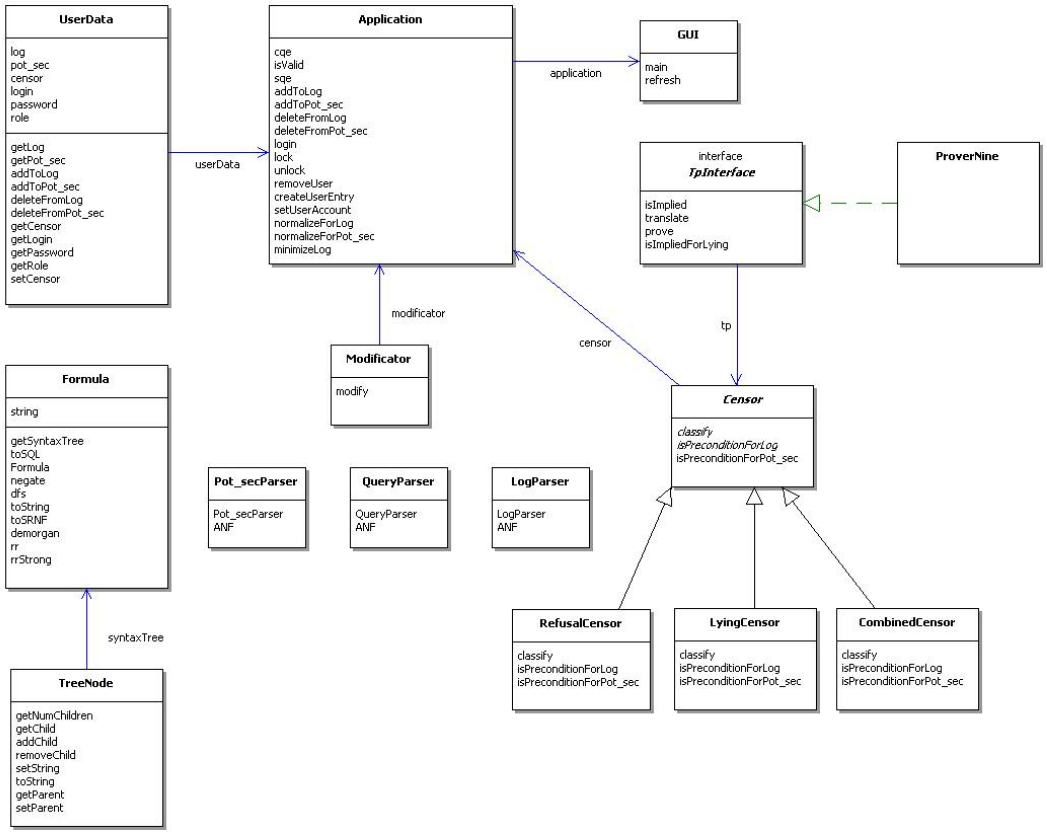

Abbildung 22: Klassendiagramm der zweiten Iteration

17.6 UML-Modellierung

17.6.1 Klassendiagramm

Es wird im Folgenden ein kurzer Einblick in die Struktur des Klassendiagramms gegeben. Dabei liegt das Hauptaugenmerk auf den Unterschieden und Erweiterungen gegenüber dem ersten Entwurf.
Es ist nun die Negation in Anfragen zugelassen. Dafür werden die Anfragen in *SRNF* (siehe Kapitel 17.1) überführt und daraufhin dahingehend überprüft, ob sie safe range sind. Hierfür wurden neue Methoden hinzugefügt.
Für jeden Benutzer werden seine Daten (Benutzername, Benutzerrolle, Benutzerpasswort, Zensor) ebenfalls in *UserData* verwaltet. Hierfür sind neue Attribute hinzugefügt worden.
Ebenfalls neu ist die Unterscheidung verschiedener Zensoren. Es gibt nun eine abstrakte Klasse *Censor*, die von den jeweiligen Zensoren *LyingCensor*, *RefusalCensor* und *CombinedCensor* implementiert wird. Unten werden die Klassen, die Methoden und die Attribute erläutert, die in der zweiten Iteration neu entstanden oder verändert worden sind.

- **Klasse Formula**
  Neu hinzugekommen sind die folgenden Methoden:
  - **rr(syntaxTree: TreeNode): LinkedList<String>** Diese Methode überprüft, ob eine Formel, gegeben in SRNF, safe range ist.
  - **toSRNF(): Formula** Diese Methode überführt die Formel in SRNF.
  - **demorgan(syntaxTree: TreeNode): TreeNode** Es werden auf die Formel solange die De Morgan'schen Regeln angewendet, bis Negationen nur noch unmittelbar vor Atomen auftreten.

- **Klasse UserData**
  Hier sind die in Kapitel 17.5 genannten benutzerspezifischen Daten als Attribute neu hinzugekommen:
  - **char censor** Der dem Benutzer zugeordnete Zensor, identifiziert als char ('l','r','c').
  - **String login** Benutzername, gespeichert als String.
  - **String passwort** Passwort, gespeichert als String.
  - **String role** Rolle (Art des Benutzers, z.B. Administrator).

  Die neuen Methoden der Klasse *UserData*:
  - **getCensor(): char** Diese Methode gibt den Zensor (Art des Zensors) zurück, der dem Benutzer zugeordnet ist.
  - **getCensor(loginKey: String): char** Die Methode liefert den Zensor des mit dem Parameter *loginKey* bezeichneten Benutzers zurück.
  - **getUserName(): String** Diese Methode gibt den Benutzername des Benutzers zurück.
  - **getRole(): String** Diese Methode gibt die Rolle des Benutzers zurück, die dem Benutzer zugeordnet ist.
  - **getLog(loginKey: String): LinkedList<Formula>** Die Methode liefert die Liste mit den Teilformeln aus dem Benutzerwissen des mit dem Parameter *loginKey* bezeichneten Benutzers zurück.
  - **getLog(): LinkedList<Formula>** Die Methode liefert die Liste mit den Teilformeln aus dem Benutzerwissen des eingeloggten Benutzers zurück.
  - **getPot_sec(loginKey: String): LinkedList<Formula>** Die Methode liefert die Liste mit den Teilformeln aus dem pot_sec des mit dem Parameter *loginKey* bezeichneten Benutzers zurück.
  - **getPot_sec(): LinkedList<Formula>** Die Methode liefert die Liste mit den Teilformeln aus dem pot_sec des eingeloggten Benutzers zurück.
  - **addToLog(loginKey: String, item: Formula)** Die Methode fügt eine Formel zum Benutzerwissen des mit dem Parameter *loginKey* bezeichneten Benutzers hinzu.

- **addToLog(item: Formula)** Die Methode fügt eine Formel zum Benutzerwissen des eingeloggten Benutzers hinzu.
- **addToPot_sec(loginKey: String, item: Formula)** Die Methode fügt eine Formel zu den potentiellen Geheimnissen des mit dem Parameter *loginKey* bezeichneten Benutzers hinzu.
- **addToPot_sec(item: Formula)** Die Methode fügt eine Formel zu den potentiellen Geheimnissen des eingeloggten Benutzers hinzu.
- **deleteFromLog(loginKey: String, index: int)** Diese Methode löscht die Formel am angegebenen Index im Benutzerwissen des mit dem Parameter *loginKey* bezeichneten Benutzers.
- **deleteFromLog(index: int)** Diese Methode löscht die Formel am angegebenen Index im Benutzerwissen.
- **deleteFromPot_sec(loginKey: String, index: int)** Diese Methode löscht die Formel am angegebenen Index im *pot_sec* des mit dem Parameter *loginKey* bezeichneten Benutzers.
- **deleteFromPot_sec(index: int)** Diese Methode löscht die Formel am angegebenen Index im *pot_sec*.

- **Klasse Application**
  An dieser Stelle kommen mehrere neue Methoden hinzu.
  - **lock(tableName: String)** Diese Methode überprüft, ob die Tabelle mit dem Namen *tableName* reserviert ist. Ist dies nicht der Fall, so wird sie reserviert.
  - **unlock(tableName: String)** Diese Methode gibt die reservierte Tabelle mit dem Namen *tableName* wieder frei.
  - **normalizeForLog(syntaxTree: TreeNode): LinkedList<TreeNode>** Diese Methode wendet auf den Syntaxbaum *syntaxTree* eine Minimalisierung gemäß der Menge *log* an. Das Ergebnis ist eine *LinkedList* mit den Teilformeln.
  - **normalizeForPot_sec(syntaxTree: TreeNode): LinkedList<TreeNode>** Diese Methode wendet auf den Syntaxbaum *syntaxTree* eine Minimalisierung gemäß der Menge *pot_sec* an. Das Ergebnis ist eine *LinkedList* mit den Teilformeln.
  - **minimizeLog()** Entfernt Redundanzen aus der Menge *log*.
  - **createUserEntry(newLogin: String, newRolle: String, newCensor: String)** Diese Methode fügt den neuen Benutzer in die Datenbanktabelle *benutzertabelle* ein. Der neue Benutzeraccount wird erstellt.
  - **removeUser(login: String)** Diese Methode löscht den Benutzer mit dem Benutzernamen *login* aus der Datenbank.
  - **setUserAccount(newLogin: String, newRolle: String, newCensor: String)** Mit dieser Methode können Änderungen an den Account-Daten eines Benutzers durchgeführt werden.

- **addToLog(loginKey: String, item: Formula)** Die Methode fügt eine Formel zum Benutzerwissen des mit dem Parameter *loginKey* bezeichneten Benutzers hinzu.
- **addToLog(item: Formula)** Die Methode fügt eine Formel zum Benutzerwissen des eingeloggten Benutzers hinzu.
- **addToPot_sec(loginKey: String, item: Formula)** Die Methode fügt eine Formel zu den potentiellen Geheimnissen des mit dem Parameter *loginKey* bezeichneten Benutzers hinzu.
- **addToPot_sec(item: Formula)** Die Methode fügt eine Formel zu den potentiellen Geheimnissen des eingeloggten Benutzers hinzu.
- **deleteFromLog(loginKey: String, index: int)** Diese Methode löscht die Formel am angegebenen Index im Benutzerwissen des mit dem Parameter *loginKey* bezeichneten Benutzers.
- **deleteFromLog(index: int)** Diese Methode löscht die Formel am angegebenen Index im Benutzerwissen des mit dem Parameter *loginKey* bezeichneten Benutzers.
- **deleteFromPot_sec(loginKey: String, index: int)** Diese Methode löscht die Formel am angegebenen Index im *pot_sec* des mit dem Parameter *loginKey* bezeichneten Benutzers.
- **deleteFromPot_sec(index: int)** Diese Methode löscht die Formel am angegebenen Index im *pot_sec*.

- **Klasse Censor**
  In dieser abstrakten Klasse werden die Methoden und die Attribute definiert, die der Klassifikation der Anfragen dienen. Jeder abgeleitete Zensor muss diese Methoden geeignet implementieren.

  - **isPreconditionForLog(log: LinkedList<Formula>, lognew: Formula, potsec: LinkedList<Formula>): boolean** Diese Methode prüft, ob die Vorbedingungen eines Zensors erfüllt sind. Dazu wird dieser das Benutzerwissen, die potentiellen Geheimnisse und das neu hinzuzufügende Wissen übergeben.
  - **isPreconditionForPot_sec(log: LinkedList<Formula>, psinew: Formula, potsec: LinkedList<Formula>): boolean** Diese Methode prüft, ob die Vorbedingungen eines Zensors erfüllt sind. Dazu wird dieser das Benutzerwissen, die potentiellen Geheimnisse und das neu hinzuzufügende potentielle Geheimnis übergeben. Diese Methode wird beim Hinzufügen zur Menge *pot_sec* angewendet. Der Unterschied zu *isPreconditionForLog* ist, dass innerhalb dieser Methode *isImplied* anders angewendet werden muss.
  - **classify(log: LinkedList<Formula>, query: Formula, potsec: LinkedList<Formula>): CType** Die Methode klassifiziert die vom Benutzer gestellte Anfrage gemäß den möglichen Rückgabewerten *REFUSE, LIE* und *UNCRITICAL*.

- **Klassen RefusalCensor, LyingCensor, CombinedCensor**
  In diesen Klassen werden die abstrakten Methoden der Oberklasse *Censor* implementiert und im Rahmen des entsprechenden Zensors angepasst.

17.6.2 Zensoren

Lügenzensor Im Folgenden wird der interne Programmablauf des Lügenzensors beschrieben.
Hierzu wurde ein Aktivitätsdiagramm erstellt (siehe Abbildung 23), welches den genauen Ablauf spezifiziert. Im ersten Schritt wird die ausgewertete Anfrage in Form von $eval^*(\Phi)(db)$ eingelesen. Es muss nun überprüft werden, ob die Bedingung $log \cup \{eval^*(\Phi)(db)\} \not\models pot\_sec\_disj$ eingehalten werden kann, dazu wird in Schritt zwei zunächst die Disjunktion der potentiellen Geheimnissen pot_sec_disj erzeugt als $\bigvee_{\Psi \in pot\_sec} \Psi$.
Danach wird der Ausdruck $log \cup \{eval^*(\Phi)(db)\} \models pot\_sec\_disj$ in die Sprache des Theorembeweisers übersetzt und anschließend mit dem Theorembeweiser überprüft. Abhängig vom Rückgabeergebnis des Theorembeweisers wird die ursprüngliche Anfrage oder die Negation dieser Anfrage ausgegeben. Wird die Bedingung $log \cup \{eval^*(\Phi)(db)\} \models pot\_sec\_disj$ erfüllt, so wird $\neg eval^*(\Phi)(db)$ ausgegeben und ansonsten $eval^*(\Phi)(db)$.

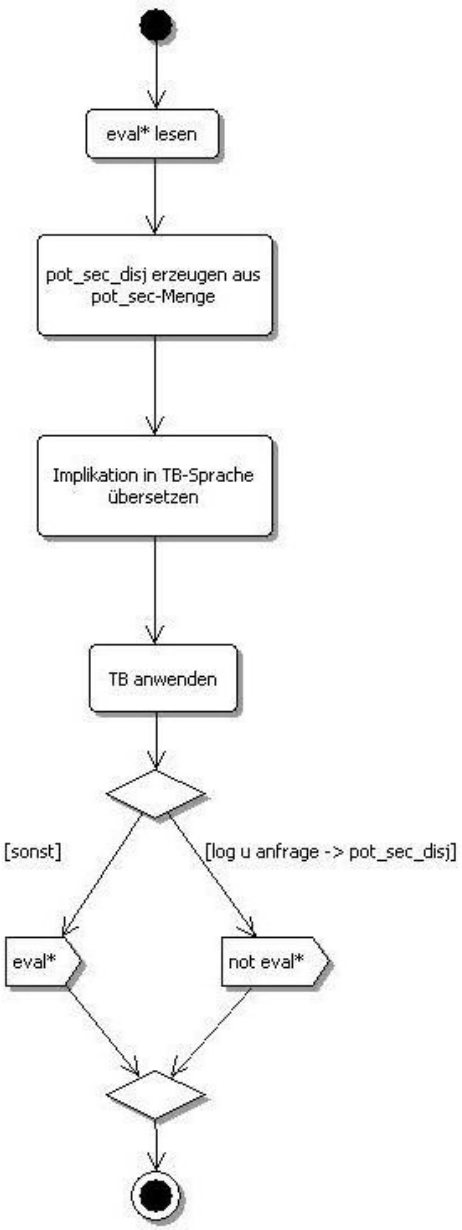

Abbildung 23: Interner Ablauf des Lügenzensors

Kombinierter Zensor Für den kombinierten Zensor wurde ebenfalls ein Aktivitätsdiagramm erstellt (siehe Abbildung 24), welches den genauen Ablauf spezifiziert.
Im ersten Schritt wird die ausgewertete Anfrage in Form von $eval^*(\Phi)(db)$ eingelesen. Die zu überprüfende Bedingung lautet

$$\exists \Psi_1 \in pot\_sec : log \cup \{eval^*(\Phi)(db)\} \models \Psi_1 \text{ und}$$
$$\exists \Psi_2 \in pot\_sec : log \cup \{\neg eval^*(\Phi)(db)\} \models \Psi_2.$$

Als erstes wird eine Schleife durchlaufen, in der der erste Teil der Bedingung in die Sprache des Theorembeweisers übersetzt wird und anschließend dem Theorembeweiser zum Lösen des Implikationsproblems übergeben wird. Wird nun kein passendes Ψ_1 gefunden, welches den ersten Teil der Bedingung erfüllt, so wird die Anfrage $eval^*(\Phi)(db)$ ausgegeben. Wird aber ein solches Ψ_1 gefunden, das den ersten Teil der Bedingung erfüllt, so wird die Anfrage $eval^*(\Phi)(db)$ negiert und eine zweite Schleife durchlaufen. In dieser Schleife wird der zweite Teil der Bedingung in die Sprache des Theorembeweisers übersetzt und mit dem Theorembeweiser überprüft. Ist ein entsprechendes Ψ_2 gefunden worden, so dass die gesamte Bedingung erfüllt wird, so wird die Anfrage abgelehnt. Konnte kein entsprechendes Ψ_2 gefunden werden, so wird gelogen, d.h. $\neg eval^*(\Phi)(db)$ ausgegeben.

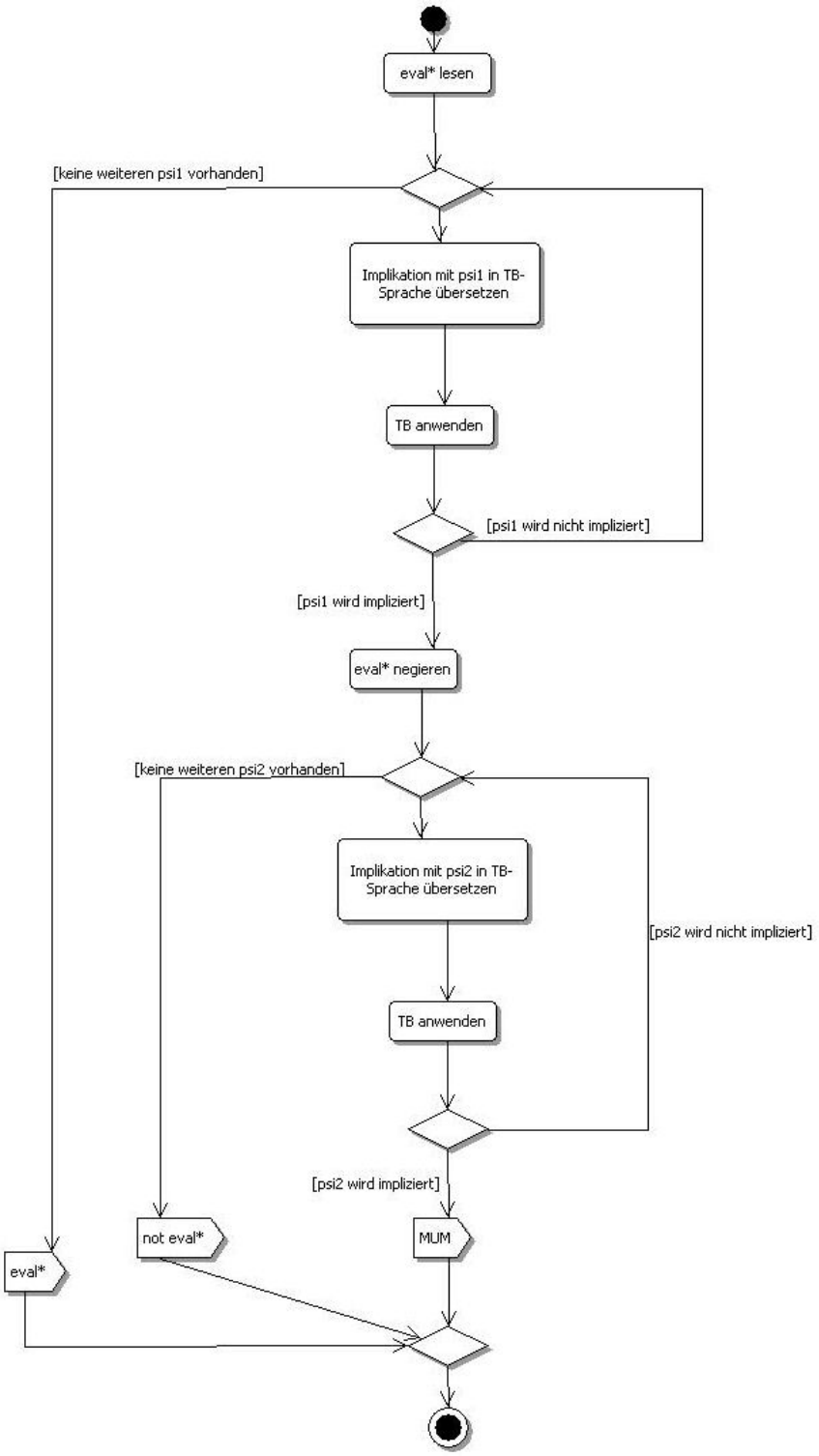

Abbildung 24: Interner Ablauf des kombinierten Zensors

17.6.3 Interner Ablauf einer kontrollierten Anfrage

Dieser Abschnitt soll den internen Programmablauf der kontrollierten Anfrageauswertung beschreiben. Es sollen vor allem die Unterschiede zum ersten Release herausgestellt werden. Hierfür werden die entsprechenden, in der Entwurfsphase erstellten, Aktivitäts- und Sequenzdiagramme herangezogen.

Abbildung 25 zeigt den Ablauf der kontrollierten Anfrageauswertung dargestellt als Aktivitätsdiagramm. Der Ablauf unterscheidet sich von dem im ersten Release durch die zusätzliche Umformung in SRNF und der anschließenden Prüfung auf safe rangeness. Zudem ist das Diagramm im Bereich des Zensors verallgemeinert worden. Es wird nun nur noch eine Aktivität durchlaufen, die sich „Eingabe von Zensor bewerten lassen“ nennt. Das Innere dieser Aktivität wird, je nach gewähltem Zensor, von einem aus drei weiteren Aktivitätsdiagrammen (siehe Kapitel 17.6.2) dargestellt. Auch muss nun vor jeder Auswertung einer gültigen Anfrage ein Exklusivzugriff auf die Tabellen, die das Benutzerwissen und die potentiellen Geheimnisse verwalten, angefordert werden. Nachdem der Benutzer seine Eingabe gemacht hat, wird zunächst geprüft, ob es sich dabei um eine gültige Anfrage im Sinne der hierfür definierten Grammatik handelt. Liegt eine grammatikalisch nicht korrekte Anfrage vor, so wird abgebrochen. Ansonsten muss die Anfrage darauf geprüft werden, ob sie die von uns festgelegten semantischen Nebenbedingungen einhält. Wenn dies nicht der Fall ist, so wird auch hier abgebrochen. Sind auch die semantischen Nebenbedingungen erfüllt, so muss nun noch die Umformung in SRNF und eine anschließende Prüfung auf safe rangeness durchgeführt werden. Besteht die Anfrage auch diesen Test, so wird versucht, ein Lock auf die benötigten Tabellen zu bekommen. Gelingt dies, dann wird die Eingabe in eine SQL-Anfrage übersetzt, für die das unterliegende DBMS – entsprechend interpretiert – den Wahrheitswert der Anfrage in der gegebenen DB-Instanz liefert. Die korrekte Antwort wird aber zunächst noch zurückgehalten und erst einmal die Anfrage durch den Zensor auf ihre Gefährlichkeit hin überprüft. Danach wird entsprechend der Ausgabe des Zensors geantwortet und bei Bedarf das Benutzerwissen normalisiert, aktualisiert und dann minimalisiert. Nun können die Benutzertabellen wieder freigegeben werden.

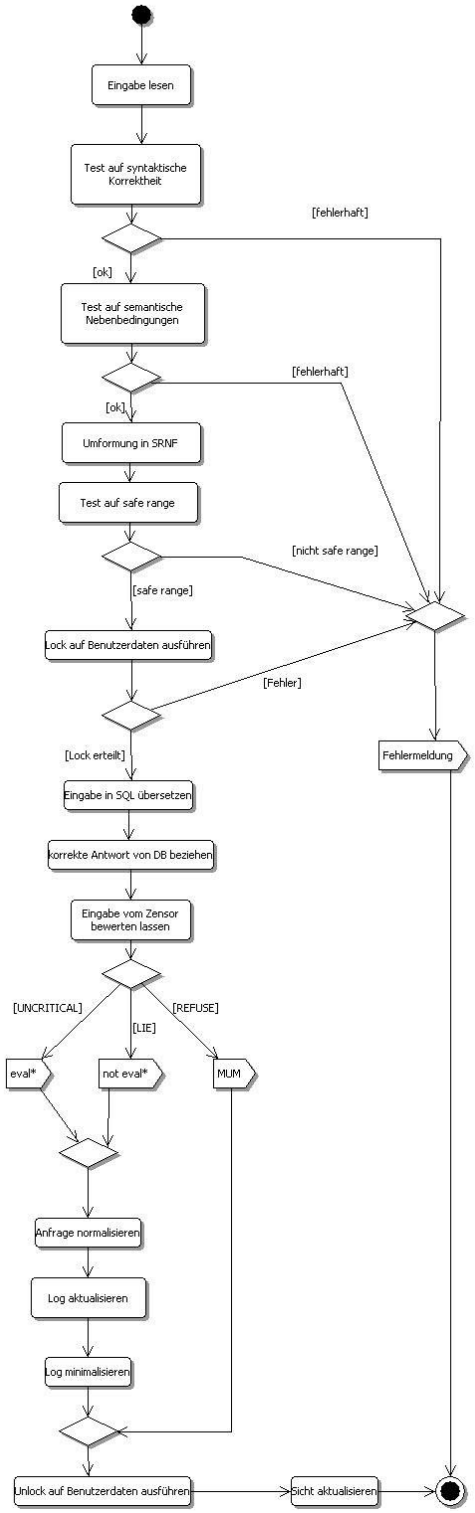

Abbildung 25: Aktivitätsdiagramm der kontrollierten Anfrageauswertung in der zweiten Iteration

Das Sequenzdiagramm aus Abbildung 26 zeigt einen möglichen Ablauf der kontrollierten Anfrageauswertung mit kombiniertem Zensor im Detail. Hierbei wird sichtbar, welche Methoden von welchen Klassen aus aufgerufen werden und welche Informationen dabei fließen. Das Sequenzdiagramm zeigt genau einen möglichen Ablauf des Aktivitätsdiagramms. Hierbei gelten folgende Voraussetzungen:

- Die Anfrage ist syntaktisch korrekt und erfüllt alle semantischen Nebenbedingungen.
- Die Anfrage besteht den Test auf safe rangeness.
- Der Exklusivzugriff auf die Tabellen wird erteilt.
- Es gibt zwei potentielle Geheimnisse.
- Die Anfrage ist gefährlich.
- Die negierte Anfrage kann durch Normalisierung in zwei Teilformeln aufgespalten werden.

Wenn diese Voraussetzungen gelten, ergibt sich genau der dargestellte Ablauf.
Die Benutzeroberfläche ruft die Methode *cqe* im Objekt *Application* auf und übergibt ihr den vom Benutzer eingegebenen String. Innerhalb der Methode *cqe* wird als nächstes unter Übergabe des eingegebenen Strings der *QueryParser*, ein Parser für die Eingabe, initialisiert. Durch den Aufruf der Methode *ANF* (die das Startsymbol ist, siehe Kapitel 12.1.3) wird der Parser gestartet. Da die Eingabe eine gültige Anfrage war, wird ein Objekt vom Typ *SimpleNode* zurückgegeben, welches die Wurzel des entsprechenden Syntaxbaums als Java-Datenstruktur darstellt.
Nun wird ein Objekt vom Typ *TreeNode* erzeugt. Der Konstruktor erhält als Parameter die vom Parser erzeugte *SimpleNode* Datenstruktur, aus welcher ein neuer Syntaxbaum, deren Knoten vom Typ *TreeNode* sind, erzeugt wird. Der Syntaxbaum wird nun mittels der Methode *isValid* auf seine Gültigkeit bezüglich der semantischen Nebenbedingungen geprüft. Nach Voraussetzung wird hier *true* zurückgegeben.
Anschließend wird die Anfrage in SRNF überführt und mittels der Methode *rr* auf safe rangeness überprüft.
Danach wird mittels der Methode *lock* der benötigte Exklusivzugriff auf die entsprechenden Tabellen angefordert. Wie schon bekannt, wird dann mittels *toSQL* die passende SQL-Anfrage erzeugt und der Wahrheitswert der Anfrage in der gegebenen DB-Instanz ermittelt.
Die Methode *classify* der Klasse *CombinedCensor* soll eine Klassifizierung der Anfrage zurückgeben. Der Rückgabewert ist vom selbstdefinierten Typ *CType* und soll für den modellierten kombinierten Zensor einer der Werte 'UNCRITICAL', 'MUM' oder 'LIE' sein.
Um die Klassifizierung vorzunehmen, muss der Zensor prüfen, ob $\exists \Psi_1 \in pot\_sec : log \cup \{eval^*(\Phi)(db)\} \models \Psi_1$ und $\exists \Psi_2 \in pot\_sec : log \cup \{\neg eval^*(\Phi)(db)\} \models \Psi_2$ gelten. Hierzu ruft dieser die Methode *isImplied* der Klasse *ProverNine* auf. Hier werden mittels der Methode *translate* die Formel, das Benutzerwissen und die potentiellen

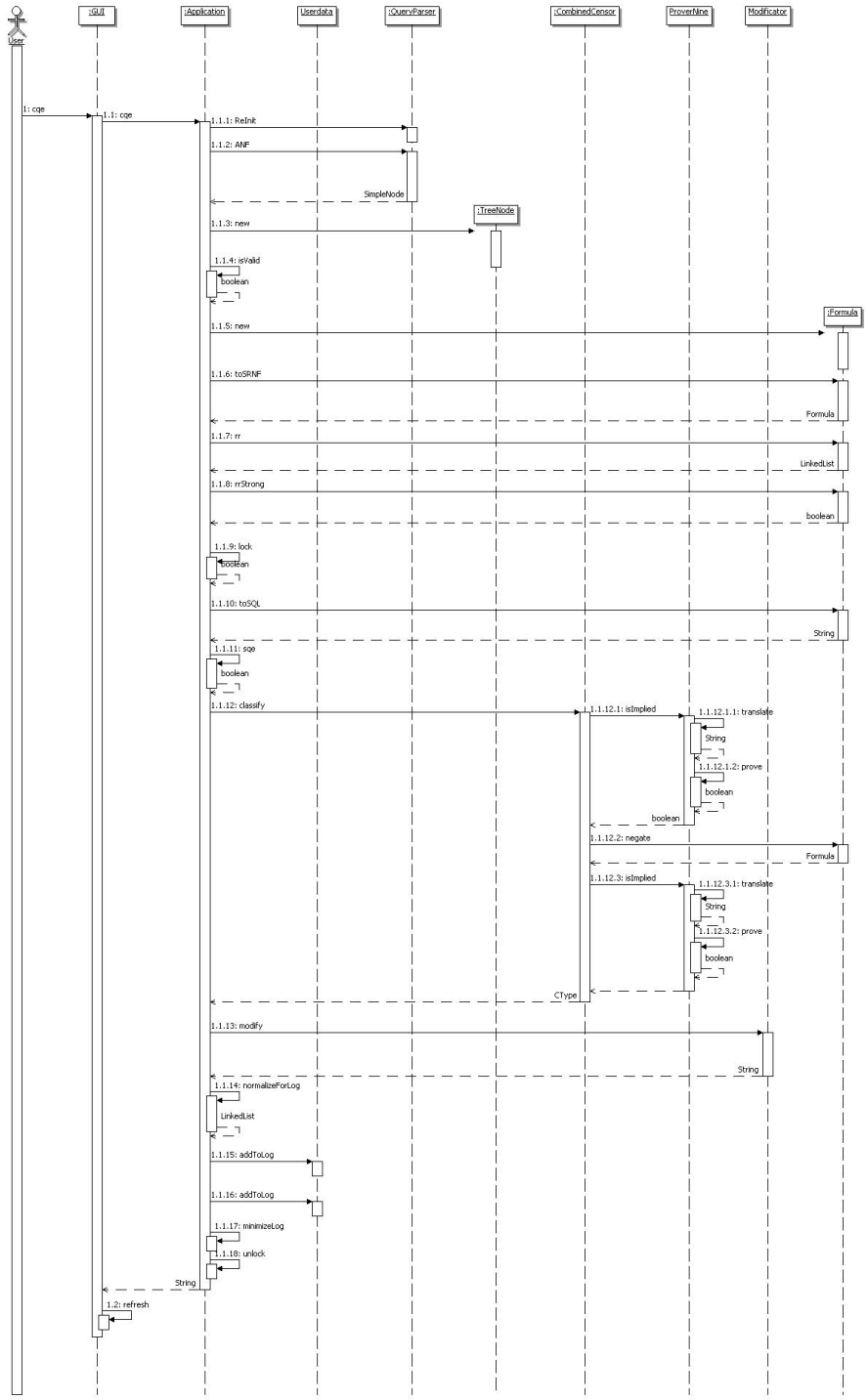

Abbildung 26: Sequenzdiagramm der kontrollierten Anfrageauswertung in der zweiten Iteration

Geheimnisse in die Syntax des Theorembeweisers gebracht. Anschließend werden mit *prove* in einer Schleife alle vorhandenen $\Psi_1 \in pot\_sec$ durchlaufen und die Anfrage, das Benutzerwissen und das aktuelle Ψ_1 als Eingaben für den Theorembeweiser in einer Datei gespeichert.
Danach wird der Theorembeweiser als ein externer Prozess aufgerufen, der die zuvor erzeugte Datei als Eingabe benutzt. Der Theorembeweiser kommt zu dem Ergebnis, dass das erste potentielle Geheimnis ableitbar ist. Nun wird geprüft, ob das zweite potentielle Geheimnis (das erste wurde bereits überprüft) mit der negierten Anfrageformel eine wahre Implikation liefert. Das potentielle Geheimnis Ψ_2 wird in diesem Fall nicht impliziert. Das Ergebnis wird dem Modifikator mitgeteilt, der die entsprechende Handlungsvorschrift ausgibt. In diesem Fall „LIE“, weshalb die gelogene Formel (d.h. die negierte Anfrage) normalisiert (nach Abschnitt 17.3.4) und dem Benutzerwissen hinzugefügt wird.
Nachdem dann das Benutzerwissen minimalisiert wurde, kann mittels der Methode *unlock* der Exklusivzugriff auf die Tabellen wieder abgegeben werden. Dem Benutzer wird dann die Antwort gezeigt.

17.6.4 Benutzerauthentifizierung und -trennung

In diesem Abschnitt sollen die internen Abläufe der Benutzerauthentifizierung und -trennung vorgestellt und beschrieben werden. Zur Erläuterung werden die in der Entwurfsphase entstandenen Aktivitätsdiagramme zur Benutzertrennung herangezogen.
Abbildung 27 stellt den Ablauf des Authentifizierungsvorganges zur Schau. Die Authentifizierung wird gestartet, indem dem Nutzer zuerst ein Login-Fenster angezeigt wird. Nach der Eingabe eines Benutzernamens und Passwortes werden diese Daten auf Korrektheit überprüft. Ein Nutzer kann sich nur dann erfolgreich authentifizieren, wenn für ihn ein gültiges Konto existiert und alle von ihm eingegebenen Daten korrekt sind. Nach erfolgreicher Authentifizierung wird eine der Rolle des Benutzers entsprechende GUI geladen. Ein Administrator bekommt also eine GUI mit administrativen Funktionalitäten zur Benutzerverwaltung angezeigt, während ein Anfragesteller diese Funktionalitäten nicht zur Verfügung gestellt bekommt. Das Stellen einer Anfrage oder die Bearbeitung der Benutzerdaten sowie alle anderen Funktionalitäten sind erst nach einem erfolgreichen Authentifizierungsvorgang möglich. Nach einem Authentifizierungsvorgang werden *log* und *potsec* aus den diesem Nutzer zugeordneten Tabellen ausgelesen und programmintern zur weiteren Bearbeitung gespeichert. Diese lokale Kopie der Benutzerdaten wird benötigt, um bei einem Aufruf der Methode *lock* den Vergleich durchführen zu können, ob diese Daten sich verändert haben.

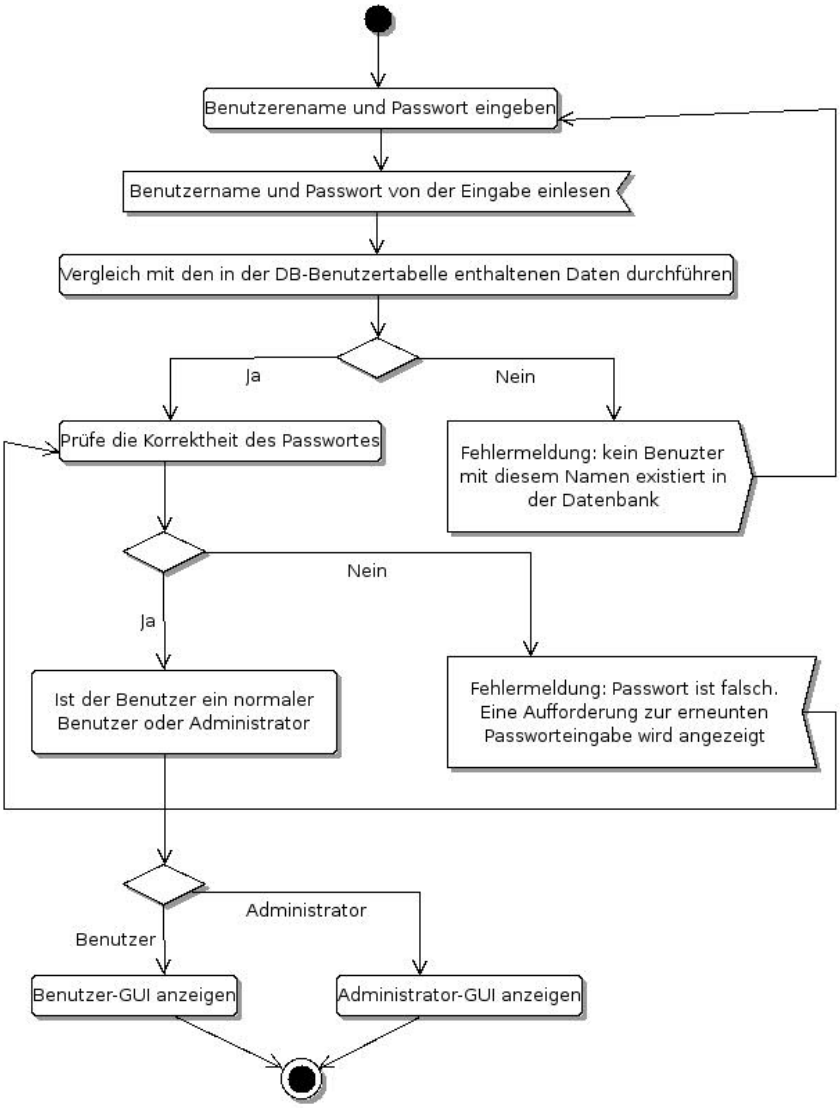

Abbildung 27: Ablauf der Benutzerauthentisierung

Abbildung 28 zeigt den Initialisierungsprozess eines Benutzerkontos. Bei der Festlegung eines Benutzerkontos sollen Benutzername, -passwort und -rolle festgelegt werden. Alle benutzerrelevanten Daten werden in einer DB-Tabelle gespeichert. Die Passwörter sind als MD5-Hashwerte zu speichern. Die Rolle kann einen Administrator oder einen einfachen Anfrager bezeichnen. Somit werden mit der Zuordnung der Rolle *Administrator* Lese- und Schreibrechte für die Benutzertabellen freigegeben. Falls einem Benutzer die Rolle *Anfrager* zugeordnet ist, kann er nur Anfragen stellen. Weiterhin dürfen keine Zugriffsrechte für die Benutzertabellen für solche Benutzer vergeben werden, d.h. er darf nicht die Informationen über Logins, Passwörter etc. anderer Benutzer erfolgreich anfordern und modifizieren können.
In der Implementierungsphase soll in der Datenbank eine Tabelle angelegt werden, die alle Benutzer umfasst. Diese Benutzertabelle enthält pro Benutzer Login, Passwort und Rolle. Bei der Erstellung eines Benutzerkontos werden also die Benutzerdaten in die Benutzertabelle eingefügt. Jedem Teilnehmer ist dann in der Datenbank eine Tabelle für die Formeln aus *log* und *potsec* zu erstellen und zuzuordnen. Die Aufgabe des Administrators bei der Initialisierung ist also auch die Festlegung der Mengen *log* und *potsec* für jeden erstellten Benutzer. Da es sich dabei um die *Formula*-Javaobjekte handelt, soll eine serialisierte Form dieser Objekte in die Datenbank eingebracht werden. Diesem Konzept zufolge soll eine Erweiterung und Trennung der graphischen Benutzeroberfläche erfolgen. Für einen Administrator soll eine Darstellung implementiert werden, die nicht nur eine Übersicht über die Benutzerdaten liefert, sondern auch die Bearbeitung dieser Daten funktionell unterstützt. Alle Datenbanktransaktionen sollen programmintern umgesetzt werden. Ein Administrator erbt die Eigenschaften eines Anfragers und kann daher auch die Anfragen stellen. Falls also ein Benutzer mit der Rolle *Administrator* angemeldet wird, sollen für diesen Nutzer auch Benutzerwissen und Sicherheitspolitik festgelegt werden. Diese Festlegung führt der Administrator, der das Benutzerkonto erstellt hat. Das heißt, dem Administrator ist die eigene Sicherheitspolitik während der normalen Anfrageauswertung bekannt.
Weiterhin soll jedem Benutzer ein Zensor hinzugefügt werden, der frei auswählbar, aber nach der Festlegung nicht mehr änderbar ist. Dies ist notwendig, da das Programm mehrere Zensormethoden (Lügenzensor, Ablehnungszensor und kombinierter Zensor) zur Verfügung stellt und diese in die Gesamtfunktionalität integriert.
Es ist zu beachten, dass bei jedem Schreibzugriff auf die Tabelle aller Benutzer eine Reservierung dieser Tabelle erfolgen soll. Falls die Tabelle zu diesem Zeitpunkt in Benutzung ist und deswegen nicht reserviert werden kann, wird eine Fehlermeldung ausgegeben. Der Administrator soll den Bearbeitungsvorgang neu starten. Nachdem alle Änderungen durchgeführt worden sind, soll die Tabelle freigegeben werden, damit ein anderer Administrator auf die Tabelle zugreifen und schreiben kann.

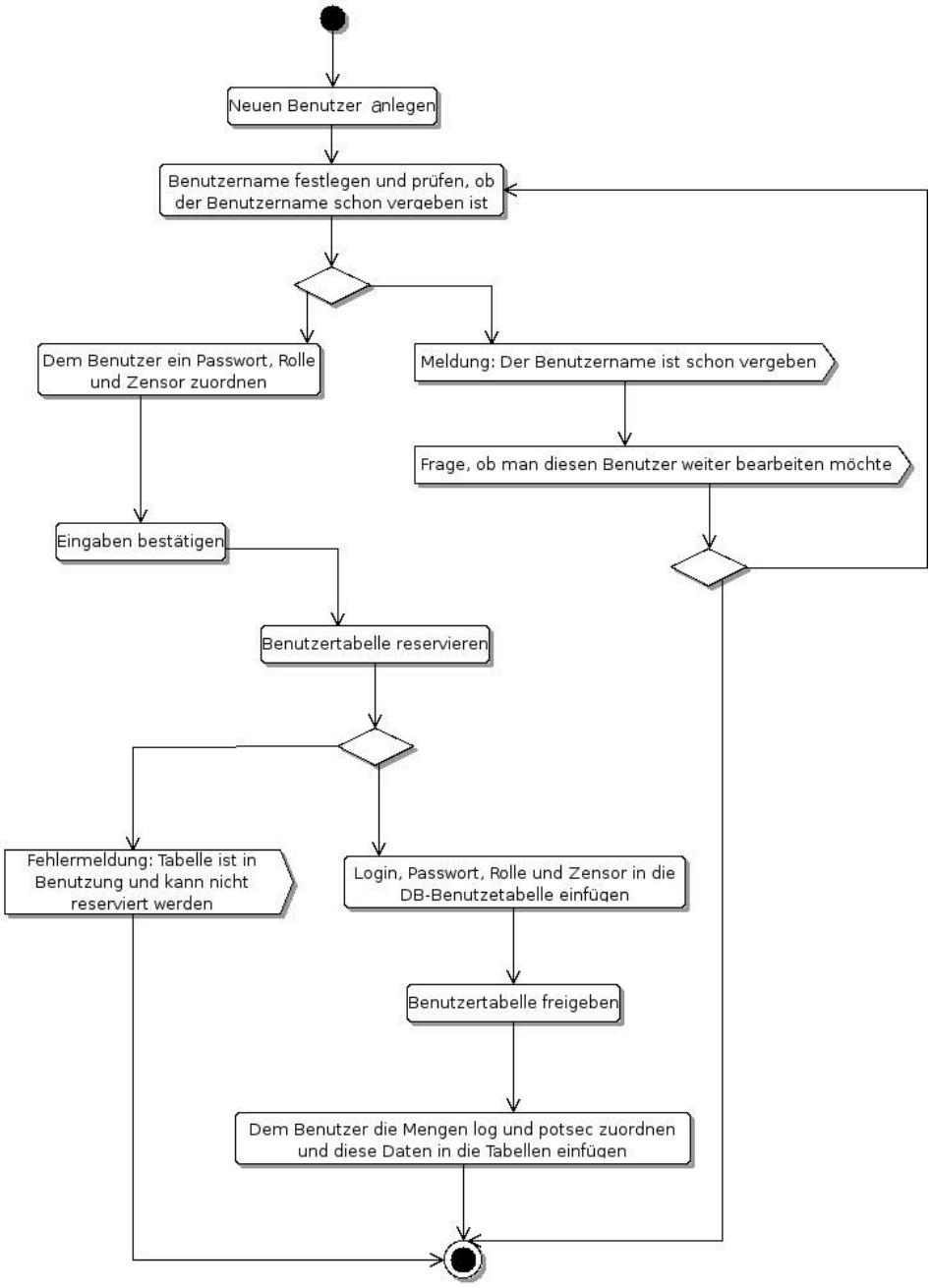

Abbildung 28: Initialisierung eines Benutzerkontos

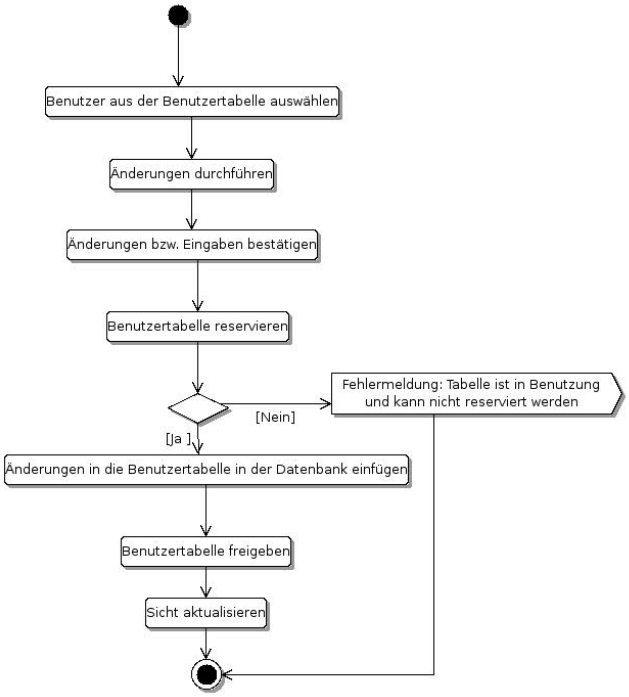

Abbildung 29: Bearbeitung eines Benutzerkontos

In der Abbildung 29 wird der Ablauf der Bearbeitung der Benutzerdaten dargestellt. Dabei wählt ein Administrator den Benutzer aus der Nutzerliste aus, führt die Änderungen durch und fügt diese in die Benutzertabelle ein. Die Sicht der Administrator-GUI wird dann sofort aktualisiert. Allerdings kann eine Änderung des Passwortes, der Rolle oder des Benutzernames erst in der nächsten Sitzung in Kraft treten. D.h. der Administrator benachrichtigt den Nutzer, für den die Änderungen durchgeführt worden sind, und teilt ihm die neuen Zugangsdaten mit. Diese Daten sollen vom Nutzer bei der nächsten Authentifizierung verwendet werden. Falls ein Administrator das Benutzerwissen modifiziert hat und die veränderten Daten zur Benutzertabelle hinzugefügt wurden, kann es dem Nutzer erst nach einer refresh-Operation der GUI angezeigt werden.

17.6.5 Anwendungsfalldiagramm

Aus den Anforderungen heraus ergeben sich unmittelbar die relevanten Anwendungsfälle. Diese sind Abbildung 30 zu entnehmen:

- $KontrollierteAnfrageauswertung$: Die Hauptprozedur. Es wird eine Anfrage ausgewertet.
- $Benutzerauthentifizierung$: Beschreibt den Authentifizierungsvorgang. In der zweiten Iteration findet eine Rollentrennung zwischen dem Administrator und dem Anfragesteller statt. Die Authentifizierung bewirkt also Identifizierung des Nutzers und es werden je nach Rolle, die jedem Nutzer zugeordnet ist, die Zugriffsrechte und die erlaubten Funktionalitäten festgelegt.
- *Benutzer bearbeiten*: Dieser Anwendungsfall beschreibt alle Operationen, die ein Administrator auf die Benutzerdaten anwenden kann. Bei der Initialisierung eines Benutzerkontos sollen vor allem Benutzername, -passwort und Rolle festgelegt werden. Der Anwendungsfall *Benutzer bearbeiten* ist erst vollständig abgelaufen, wenn mindestens einer der Anwendungsfälle $addToLog$, $addToPotsec$, $deleteFromLog$, $deleteFromPotsec$, $Zensorfestlegen$ stattgefunden hat. Dieser Anwendungsfall kann folgende Anwendungsfälle enthalten:
  - $addToLog$: Dem Benutzerwissen eine Formel hinzufügen.
  - $addToPotsec$: Den potentiellen Geheimnissen eine Formel hinzufügen.
  - $deleteFromLog$: Eine Formel aus dem Benutzerwissen entfernen.
  - $deleteFromPotsec$: Eine Formel aus den potentiellen Geheimnissen entfernen.
  - $Zensorfestlegen$: Einem Benutzer einen eindeutigen Zensor zuordnen. Dieser Zensor wird dann bei der kontrollierten Anfrageauswertung benutzt.

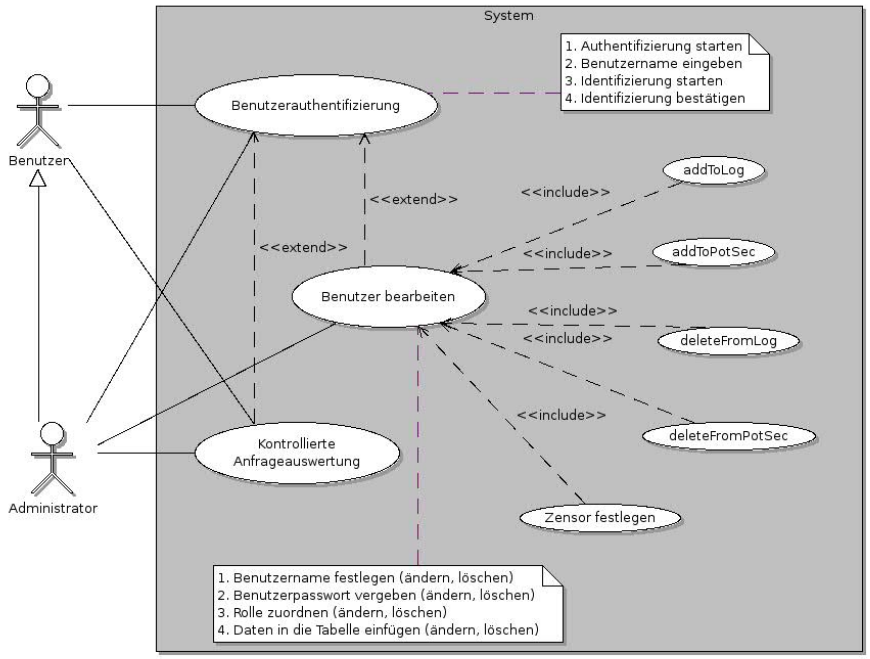

Abbildung 30: Anwendungsfälle in der zweiten Iteration

17.6.6 Hinzufügen von potentiellen Geheimnissen / Benutzerwissen

ActivityAddToPot_sec & ActivityAddToLog Wenn der Administrator ein neues potentielles Geheimnis (neues Benutzerwissen) in die pot_sec-Menge (log-Menge) einfügen möchte, sollte zunächst sichergestellt werden, dass kein anderer Administrator diese Menge bearbeitet. So werden die eventuellen Konflikte verhindert. Danach wird nur bei der pot_sec-Menge der Typ des Zensors bestimmt, dann wird die Eingabe auf syntaktische sowie semantische Korrektheit überprüft. (Bei der log-Menge wird zusätzlich überprüft, ob für die Eingabe die Vorbedingung gilt und ob sie *range restricted* ist.) In dem Fall, dass eine von den genannten Bedingungen nicht erfüllt ist, wird eine geeignete Fehlermeldung dazu ausgegeben. Sonst wird die pot_sec-Menge (log-Menge) nach Minimalisierung (siehe Abschnitt 17.3) überprüft. D. h. es wird unterschieden, ob das neue Element überhaupt in die pot_sec-Menge (log-Menge) eingefügt werden soll (wie schon bei Minimalisierung beschrieben) und wenn ja, ob eventuell einige Elemente aus der Menge gelöscht werden können oder nicht. Am Ende wird die pot_sec-Menge (log-Menge) zur eventuellen Bearbeitung von anderen Administratoren freigegeben.

Die Abbildungen 31 und 32 auf Seiten 210 und 211 stellen die Aktivitätsdiagramme von ActivityAddToPot_sec und ActivityAddToLog dar.

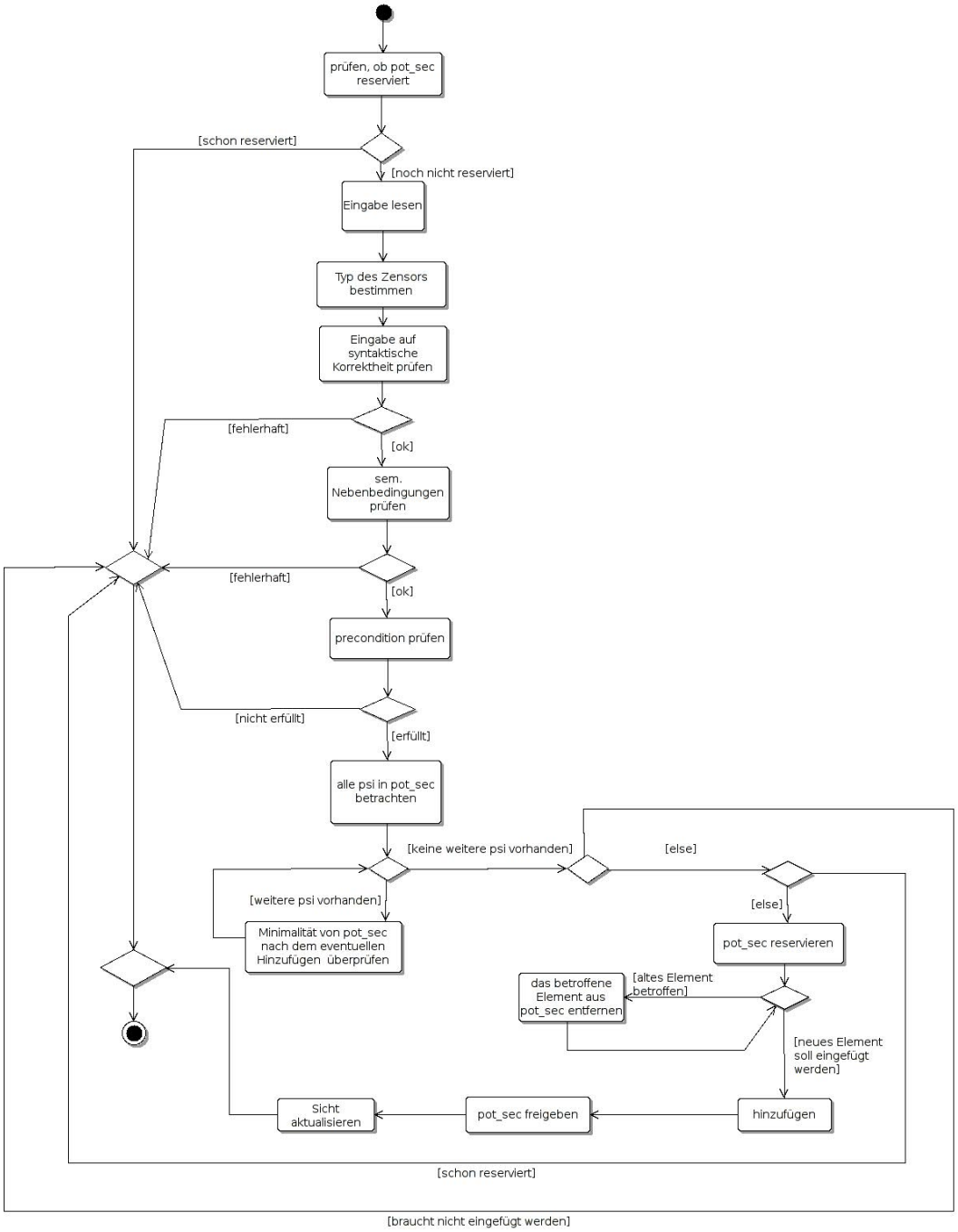

Abbildung 31: Aktivitätsdiagramm ActivityAddToPot_sec in der zweiten Iteration

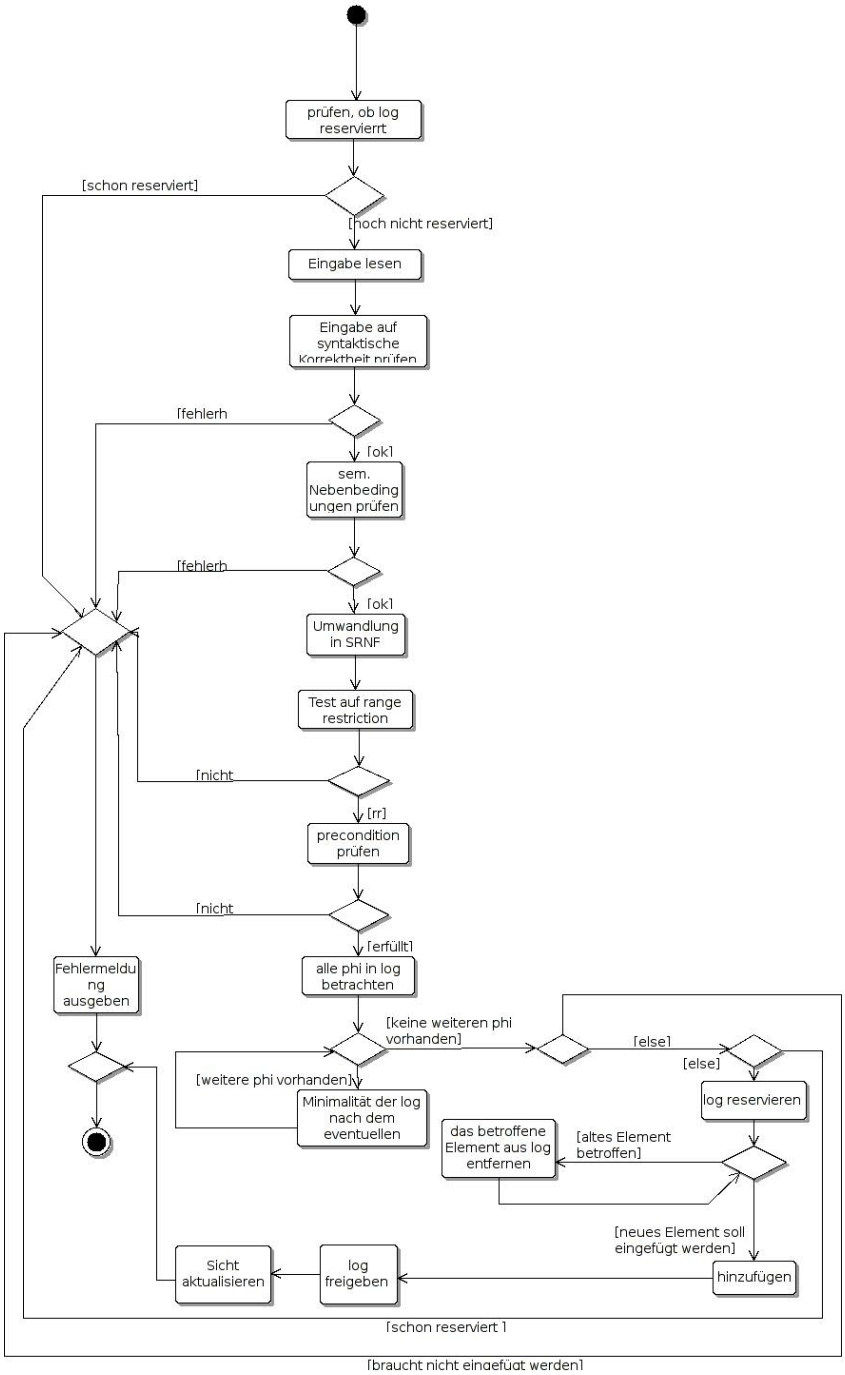

Abbildung 32: Aktivitätsdiagramm AddToLog in der zweiten Iteration

SequenceAddToPot_sec Der Administrator ruft die Methode zum Hinzufügen des potentiellen Geheimnisses über die Oberfläche auf. Die Methode wird im Application-Objekt aufgerufen.
Innerhalb dieser Methode wird ein Syntaxbaum und anschließend ein Formula-Objekt für die Eingabe erzeugt. Dann wird die Eingabe auf syntaktische und semantische Korrektheit überprüft. Danach wird der Zensortyp ermittelt, wobei es sich in diesem Beispiel um einen Lügenzensor handelt.
Anhand der Methode `isPreconditionForPot_sec()` wird überprüft, ob für die Eingabe die Vorbedingung (Precondition) gültig ist.

Da es (in diesem Beispiel) der Fall ist, wird die Eingabe normalisiert, d. h. die Disjunktionen innerhalb der Formel werden aufgelöst.
Um eventuelle Konflikte zu verhindern, wird die Datenbank vor dem Einfügen der Elemente reserviert. Danach werden die Elemente, die durch Normalisierung aus der Eingabe entstanden sind, in die pot_sec-Menge eingefügt.
Weiterhin wird überprüft, ob die Menge noch minimalisiert werden kann. Wenn es der Fall ist, werden die entsprechende Elemente aus der Menge entfernt.
Zum Schluss wird die Datenbank zu weiteren eventuellen Bearbeitungen freigegeben und die Sicht wird aktualisiert.
Die Abbildung 33 auf Seite 213 stellt das Sequenzdiagramm von AddToPot_sec dar.

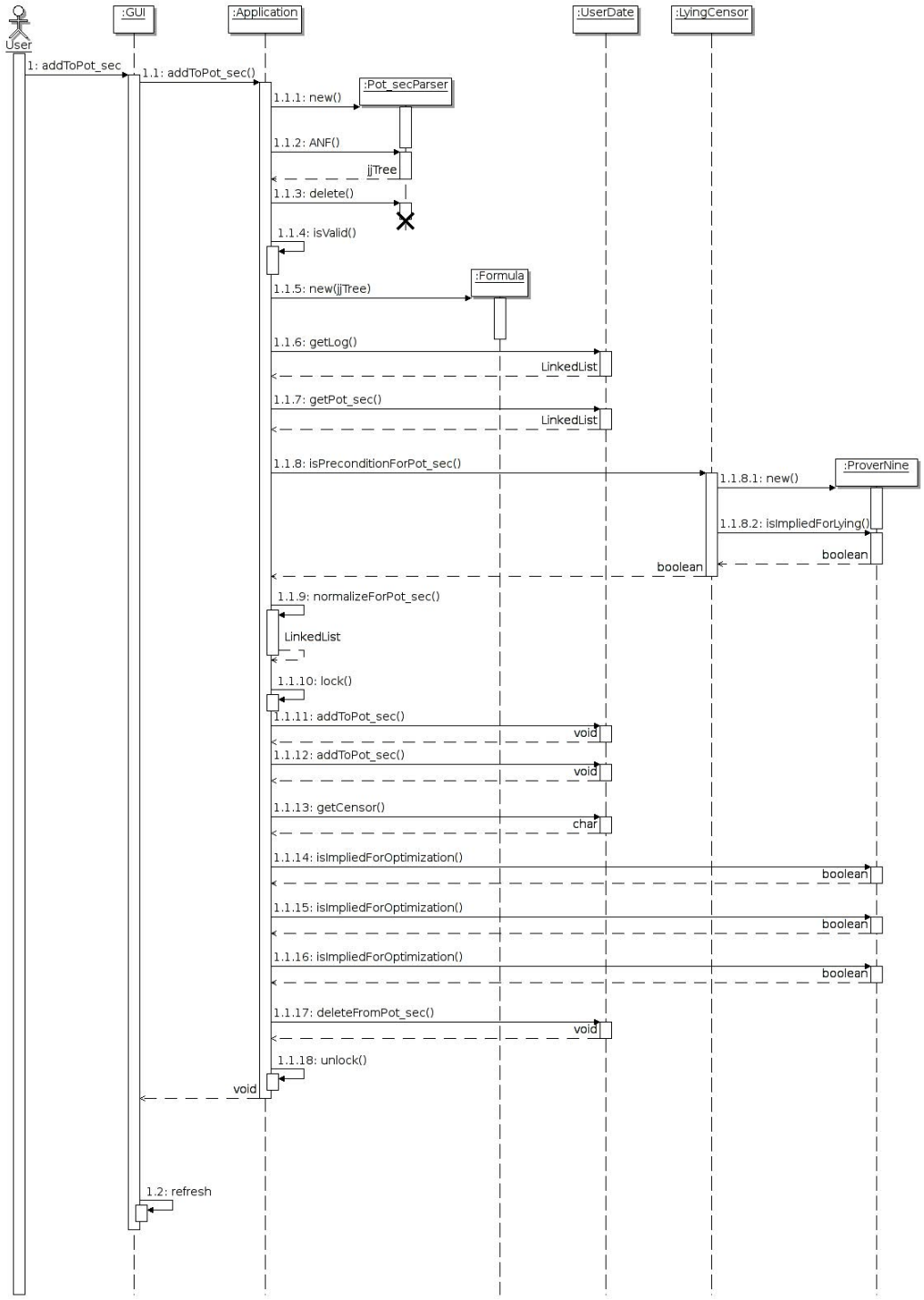

Abbildung 33: Sequenzdiagramm AddToPot_sec in der zweiten Iteration

SequenceAddToLog Der Administrator ruft die Methode zum Hinzufügen des neuen Wissens über die Oberfläche auf. Die Methode wird im Application-Objekt aufgerufen.
Der Verlauf der Methode ist zum größten Teil wie bei AddToPot_sec. Hier wird die Eingabe nur anders normalisiert. D. h. hier werden anstatt von Disjunktionen die Konjunktionen aufgelöst.
Die Abbildung 34 auf Seite 215 stellt das Sequenzdiagramm von AddToLog dar.

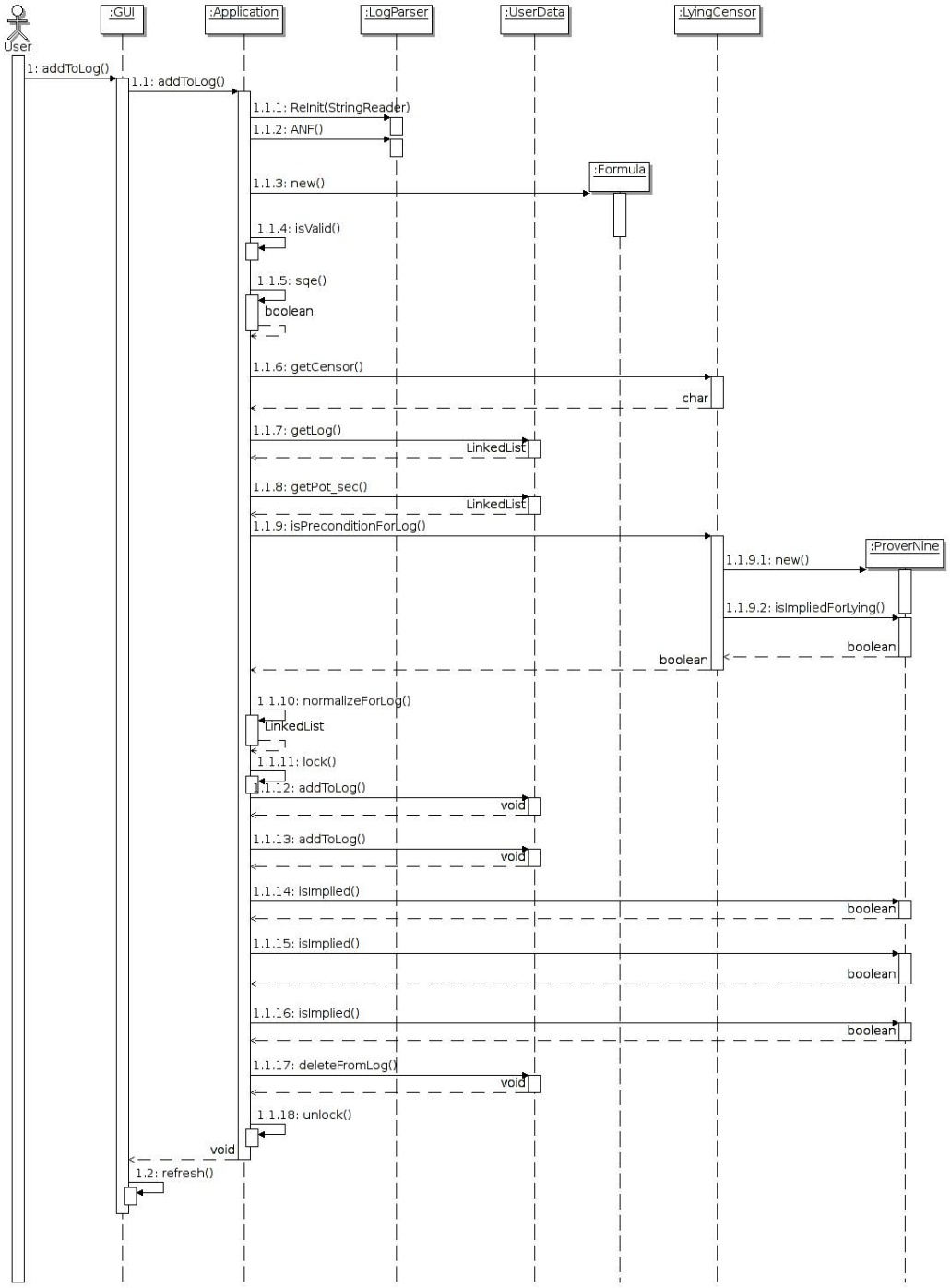

Abbildung 34: Sequenzdiagramm AddToLog in der zweiten Iteration

18 Implementierung

18.1 Implementierung der Negation in Formeln

Der folgende Abschnitt gibt eine Gegenüberstellung von Anforderung, Entwurf und tatsächlicher Implementierung bezüglich der Negation in Formeln wieder. Ziel war es, die Anfragesprache und die Sprache für das Benutzerwissen in ihrer Mächtigkeit zu erweitern. Die negationsfreien Sprachen aus der ersten Iteration schlossen unter anderem aus, dass das Wissen über funktionale Abhängigkeiten dargestellt werden konnte. Mit der nun vorliegenden Erweiterung wurde der Anforderung, dass der Benutzer „vernünftig“ mit dem System arbeiten können soll, Rechnung getragen.
Die Anforderung brachte die Problematik mit sich, mit *unsicheren* bzw. *domainabhängigen* Formeln umgehen können zu müssen. Der Ansatz war, solche Formeln zu identifizieren und zurückzuweisen. Hierzu wurden die Eigenschaften *range restricted* und *safe range* ausgenutzt (siehe Kapitel 17.1). Ob eine gegebene Formel des Relationenkalküls safe range ist, sollte ein Algorithmus *rr* entscheiden. Dieser wurde so implementiert, dass er bei Formeln, die gebundene Variablen enthalten, die nicht range restricted sind, sofort mit einer *Exception* abbricht. Ansonsten wird eine *LinkedList* zurückgegeben, die alle freien Variablen enthält, die range restricted sind. Dies entspricht exakt der Theorie im Entwurf. Dadurch leistet der Algorithmus sogar mehr, als im Moment nötig wäre, denn es werden momentan nur geschlossene Formeln betrachtet. Im Hinblick auf die Weiterentwicklung des Programms stellt dies aber einen sinnvollen Vorgriff dar.
Als eine weitergehende Einschränkung sahen die Anforderungen vor, nur „und nicht“ für Negationen zuzulassen, also negierte Literale immer unmittelbar mit nicht negierten Literalen konjunktivisch zu verknüpfen, sodass alle Variablen im negierten Literal auch im nicht negierten Literal auftauchen. Die Prüfung dieser Eigenschaft wurde in einer Methode *rrStrong* implementiert. Dabei ist es egal, ob die Negation rechts oder links von der Konjunktion steht. Auch hier wurden keinerlei Abstriche gegenüber dem Entwurf gemacht.
Zuletzt wurde der Algorithmus *toSQL* erweitert, sodass die zulässigen Formeln alle in SQL-Anfragen überführt werden können. Dies wurde auch technisch genau nach dem im Entwurf vorgestellten Schema umgesetzt. Alle Tests bezüglich der Übersetzung nach SQL (siehe 19.2.1) verliefen erfolgreich und kleinere Fehler konnten schnell ausgebessert werden, sodass davon ausgegangen werden kann, dass die Überlegungen aus dem Entwurf auch genau zur gewünschten Funktionalität führen.

18.2 Implementierung der Fehlerbehandlung

Im Laufe der Implementierungsphase stellte sich heraus, dass es möglich ist, eine Fehlerbehandlung einzig mit der generischen Klasse *Exception* zu lösen. Der Grund dafür ist, dass alle auftretenden Fehler gleich behandelt werden. Es wird lediglich die der *Exception* zugehörige Fehlermeldung in eine neue *Exception* verpackt und wiederum zur nächsthöheren Instanz weitergereicht. Es war also nicht notwendig eigene *Exceptions* zu definieren. Ein Beispiel soll verdeutlichen, wie dies in unserem Programm umgesetzt wurde.

Beispiel Folgendes Beispiel zeigt, wie wir das Exceptionkonzept in unserem Programm umgesetzt haben. Der folgende Quelltextausschnitt zeigt einen Teil der Methode *isValid*. Hier wird die Stelligkeit der Relation im Syntaxbaum (*arity*) mit der Stelligkeit der Relation im DB-Schema (*columns*) verglichen. Wenn beide Werte nicht übereinstimmen, wird eine generische *Exception* geworfen, der man bei ihrer Erzeugung gleich die zugehörigen Fehlermeldung übergibt.

```
public void isValid(TreeNode n, LinkedList vars) throws Exception {
    ...
    String rName = n.getChild(0).toString();
    int arity = n.getNumChildren()-1;
    ResultSet rs = con.createStatement().executeQuery(
        "SELECT * FROM " + rName);
    ResultSetMetaData meta = rs.getMetaData();
    int columns = meta.getColumnCount();
    if (con!=null)
        con.close();
    if (arity != columns)
        throw new Exception(
            "Stelligkeit in der Relation " + rName + " falsch");
    ...
}
```

Im nächsten Ausschnitt ist ein Teil der Methode *cqe* zu sehen.

```
public String cqe(String query) throws Exception {
    ...
    try {
       ...
       isValid(t, new LinkedList());
       ...
    }
    catch (Exception e) {
       unlock();
```

```
   throw new Exception(e.getMessage());
}
```

Der Aufruf der Methode *isValid* ist in einen *try-Block* eingebettet. Wird nun die o.g. Exception geworfen, so wird sie hier gefangen. Im entsprechenden *catch-Block* wird wiederum eine neue *Exception* geworfen, der die Fehlermeldung der gefangenen *Exception* zugewiesen wird. Die Ausnahme wird also an die nächsthöhere Instanz weitergereicht.
Der nächste Ausschnitt zeigt einen Teil der Methode *btnCqeActionPerformed* der Benutzeroberfläche. Diese Methode wird aufgerufen, wenn eine Anfrage gestartet wurde.

```
if (!txtQuery.getText().equals("")) {
    try {
        JOptionPane.showMessageDialog(this, application.cqe(
            txtQuery.getText()), "Ergebnis",
                JOptionPane.INFORMATION_MESSAGE);
    }
    catch (Exception e) {
        JOptionPane.showMessageDialog(this, e.getMessage(),
            "Fehler", JOptionPane.ERROR_MESSAGE);
    }
}
```

Wird nun die o.g. *Exception* ausgelöst, so wird sie hier abermals gefangen. Da es jetzt keinen höheren Stufen mehr gibt, wird die Fehlermeldung der *Exception* direkt mittels eines *JOptionPane* ausgegeben.

18.3 Implementierung der Benutzertrennung

18.3.1 Einleitung

Zur Erfüllung von Authentifizierungs- und Vertraulichkeitsanforderungen im Zusammenhang mit Datenbanksystemen sind spezielle Maßnahmen erforderlich, die eine Zugriffs- und Zugangskontrolle umsetzen und eine Benutzertrennung erlauben. Hierbei geht es darum, die Identität und die Eigenschaften sowie Funktionen der Nutzer individuell zu gestalten bzw. auf die Benutzerrolle abzubilden und dies mit der Erstellung der Benutzeraccounts zu unterstützen. In dieser Iteration ist das Mehrbenutzermodul mit Zugriffskontrolle und Authentifikation für das Programm „jCQE“ implementiert worden.

18.3.2 Benutzertrennung und die graphische Benutzeroberfläche des Administrators

Das Programmmodul erlaubt eine Erstellung und Verwaltung der Benutzerkonten, die durch eine geeignete Benutzeroberfläche gesteuert werden. Das schließt natürlich einen Authentifizierungsvorgang mit ein. Da alle benutzerrelevanten Daten in der Datenbank gespeichert werden sollen, wurde für diesen Zweck die Relation *benutzertabelle* angelegt:

benutzertabelle	Login	Passwort	Rolle	Zensor
	yu	xxx	einfacher Benutzer	Lügenzensor
	zhang	xxx	Administrator	Lügenzensor

Diese Relation stellt einen zentralen Baustein dar und ermöglicht uns eine datenbankinterne Benutzerverwaltung durchzuführen und diese auch in das Javamodul einzubinden.

Authentifizierung Im aktuellen Zustand des Programms können der Benutzername *zhang* und das Passwort *xxx* zum Einloggen benutzt werden. Danach kann sich jeder Projektgruppenteilnehmer ein eigenes Benutzerkonto für diese Applikation erstellen. In Abbildung 35 ist das Einloggenfenster abgebildet, welches nach dem Programmstart erzeugt und dem Nutzer angezeigt wird. Diese Funktionalität wird von der Klasse *Login* bereitgestellt. Es werden die Eingaben des Benutzernamens und -passwortes erwartet. Diese Eingaben werden nach der Betätigung des Einloggen-Buttons auf ihre Korrektheit überprüft, indem ein Aufruf der Methode *loginButton* erfolgt. Hierbei wird überprüft, ob in der Datenbank-Tabelle *benutzertabelle* der vom Benutzer eingegebene Benutzername existiert. Falls der eingegebene Benutzername tatsächlich in der DB-Tabelle vorhanden ist, wird die Richtigkeit des Passwortes überprüft.
Falls der Authentifizierungsvorgang nicht erfolgreich abgeschlossen werden konnte, wird eine Fehlermeldung ausgegeben und der Benutzer wird aufgefordert, seinen Namen und sein Passwort nochmal einzugeben. Falls die Identifikation des Nutzers erfolgreich abgelaufen ist und das Passwort korrekt eingegeben wurde, werden die Rolle,

Abbildung 35: Das Anmeldefenster

der Zensor, das Benutzerwissen und die potentiellen Geheimnisse aus der Datenbank ausgelesen. Der Benutzername, das Passwort, die Rolle des Benutzers und der Zensor werden an die Klasse *Application* zur Erzeugung des UserData-Objektes weitergeleitet. Dies ist notwendig, um programmintern die Informationen über den aktuell eingeloggten Nutzer zu verwalten, aber auch um eine kontrollierte Anfrageauswertung der Anfragen dieses Nutzers zu ermöglichen. Wie oben schon erwähnt wurde, wird nach einem erfolgreichen Authentifizierungvorgang unter anderem die Benutzerrolle aus der Datenbank-Tabelle *benutzertabelle* ausgelesen. Je nach dem Wert dieses Attributes (*Adminstrator* oder *einfacher Benutzer*) wird in der Anfrager-GUI die Funktionalität zur Steuerung der Benutzerkonten freigeschaltet. Falls der eingeloggte Benutzer die Rolle *Administrator* besitzt, wird eine Anfrager-GUI geladen, die eine Menüoption *Benutzerverwaltung* enthält. Die Abbildung 36 zeigt die Anfrager-GUI für einen *Administrator*. So ein Benutzer verfügt nun über die Lese- und Schreibrechte auf die Tabelle *benutzertabelle* in der DB und kann die Verwaltung der Benutzerdaten vornehmen. Das Anfragefenster für einen Administrator enthält nun eine Übersicht über das Benutzerwissen und die potentiellen Geheimnisse, die es auch erlaubt, direkte Modifikationen an diesen Mengen vorzunehmen. Die Funktionalitäten der Benutzerverwaltung können durch die Benutzerverwaltung-GUI realisiert werden, die im nächsten Abschnitt vorgestellt wird.
Falls der eingeloggte Benutzer die Rolle *einfacher Benutzer* besitzt, wird eine Anfrager-GUI geladen, die die oben beschriebene Menüoption nicht besitzt und dadurch keine Benutzerverwaltung ermöglicht. Außerdem können solche einfachen Benutzer nicht auf die Benutzertabelle zugreifen und sie können keine Modifikationen am eigenen Benutzerwissen vornehmen. Dies wird dadurch abgefangen, indem der Anfragestring auf das Vorkommen des Teilwortes „benutzertabelle“ untersucht wurde. Falls die Anfrage das gesuchte Teilwort enthält, wird eine Fehlermeldung ausgegeben und die Anfrage wird keinesfalls an die Datenbank geschickt. Für diese Funktionalität soll noch die Verbesserung vorgenommen werden, dass die *benutzertabelle* auf der Datenbankebene für einfache Benutzer gesperrt wird. Außerdem werden die Passwörter der Benutzer in der Tabelle *benutzertabelle* unverschlüsselt gespeichert. An dieser Stelle sollte noch dringend eine Verschlüsselung implementiert werden. Abbildung 37 zeigt das Anfragefenster für einen einfachen Benutzer ohne Administratorenrechte.

Verwaltung der Benutzeraccounts Es ist eine in das Gesamtprodukt integrierte Anwendung implementiert worden, die eine graphische Benutzeroberfläche für den

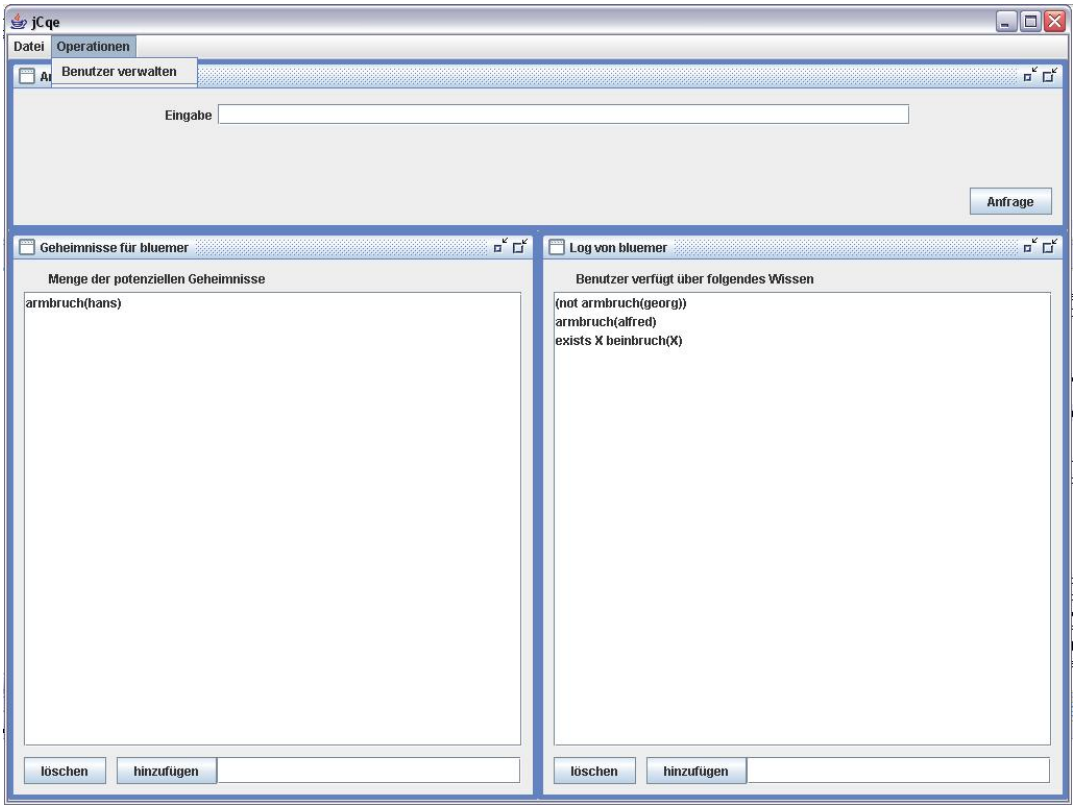

Abbildung 36: Das Anfragefenster mit den Funktionen für einen Administrator

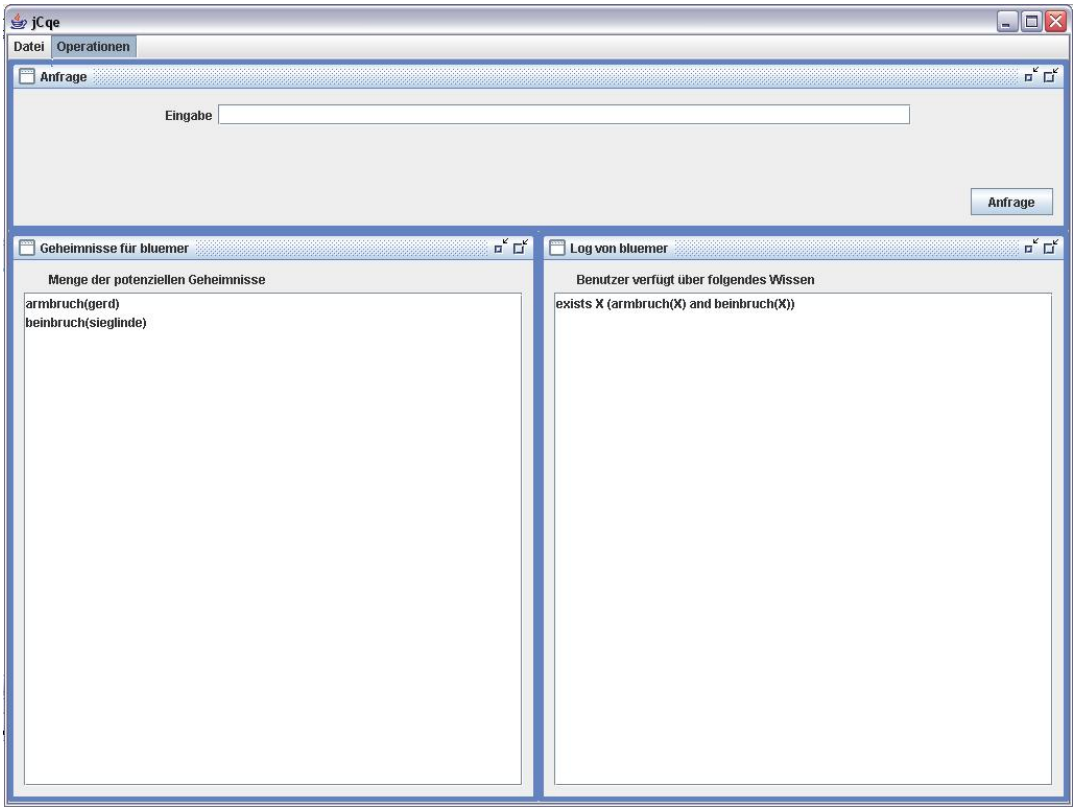

Abbildung 37: Das Anfragefenster für einen einfachen Benutzer

Administrator zur Verfügung stellt und dadurch eine GUI-basierte Verwaltung der Benutzeraccounts ermöglicht. Ein Administrator bekommt eine Übersicht über alle vorhandenen Benutzerkonten, die direkt aus der Benutzertabelle in der Datenbank ausgelesen werden. Dies ist in der Abbildung 38 veranschaulicht. Nun können neue

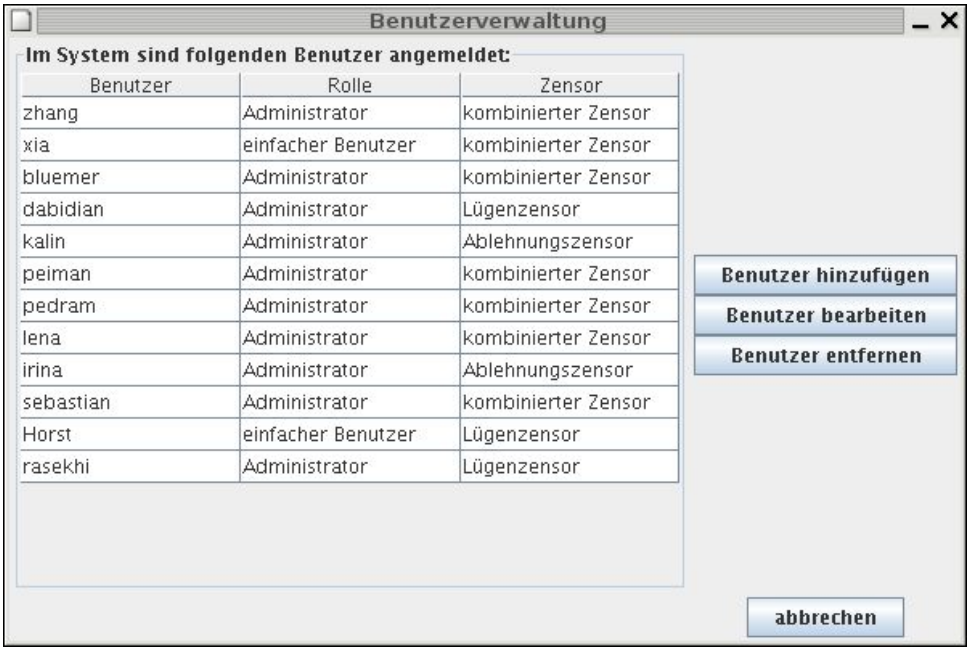

Abbildung 38: Das Benutzerverwaltungs-Frame

Benutzerkonten erstellt werden. Nach der Betätigung des *Hinzufügen*-Buttons geht das *Benutzer hinzufügen*-Fenster auf, in dem die Benutzeraccount-Daten eingegeben werden können. Nach der Bestätigung der Eingaben werden die Daten in die Datenbanktabelle *benutzertabelle* eingefügt. Die Sicht in dem Benutzerverwaltungs-Fenster wird nun aktualisiert. Die Abbildung 39 stellt diesen Vorgang zur Schau. Parallel

Abbildung 39: Erstellung eines neuen Benutzerkontos

werden für jeden neu erstellten Benutzer die Relationen *<Benutzername>Log* und *<Benutzername>Potsec* in der Datenbank erstellt. Diese Tabellen sind für den neuen Benutzer zur Speicherung des Benutzerwissens und der potentiellen Geheimnisse vorgesehen. Hier folgt ein Beispiel für solche Tabellen:

<Benutzername>Log	Formel
	armbruch(hans);
	beinbruch(peter);
	krankheit(x, armbruch);

<Benutzername>Potsec	Formel
	krankheit(x, paranoia);

Das Erstellen des neuen Benutzerkontos wird nun in der Methode *createUserEntry()* der Klasse *Application* realisiert. Die genaue Implementierung zeigt das folgende Codefragment.

```
public void createUserEntry(String newLogin,
           String newPassword, String newRole, String newCensor){
if (newLogin == null ){
throw new IllegalArgumentException("new login is null!");
}
if (newPassword == null ){
throw new IllegalArgumentException("new password is null!");
}
if (newRole == null ){
throw new IllegalArgumentException("new role is null!");
}
if (newCensor == null ){
throw new IllegalArgumentException("new censor is null!");
}
try {
Connection con = DriverManager.
getConnection("jdbc:oracle:thin:@moria:1521:db1" , "zhang", "xxx");
//Connection con = DriverManager.getConnection("jdbc:odbc:armbruch");
ResultSet rs = con.createStatement().
executeQuery("insert into benutzertabelle values" +
"('" + newLogin + "', '" + newPassword +
+  "', '" + newRole + "', '" + newCensor + "')");

ResultSet rs1 = con.createStatement().
executeQuery("create table " + newLogin +
 + "Log(formel varchar)");
ResultSet rs2 = con.createStatement().
executeQuery("create table "+ newLogin+"Potsec(formel varchar)");
```

```
rs.close();
rs1.close();
rs2.close();
con.close();
}
catch (SQLException e) {
System.out.println(e.getMessage());
}
}
```

Für ein neu erstelltes Benutzerkonto sollen nun das Benutzerwissen und die potentiellen Geheimnisse festgelegt werden. Diese Funktionalität ist nun im Benutzereigenschaften-Fenster einkodiert worden. Man wählt also einen Benutzer aus der Tabelle im Verwaltungsfenster aus und betätigt den *Editieren*-Button, nun geht das Benutzereigenschaften-Fenster auf, das alle Benutzeraccountdaten anzeigt und die Möglichkeit bietet, die benutzerrelevanten Daten, das Benutzerwissen und die potentiellen Geheimnisse des ausgewählten Benutzers hinzuzufügen oder zu löschen. Hierbei können für den ausgewählten Benutzer das Passwort und die Rolle neu festgelegt werden. Der Benutzername ist nicht änderbar, da die Tabellen *<Benutzername>Log* und *<Benutzername>Potsec* mit dem Benutzernamen referenziert sind. Der Zensor darf auch nicht mehr verändert werden, wenn er einmal festgelegt ist. Das Benutzereigenschaften-Fenster ist in der Abbildung 40 zu sehen. Wenn man die Account-Daten ändert und die Eingaben bestätigt, werden auch die entsprechen Änderungen in der Datenbanktabelle *benutzertabelle* durchgeführt. Die Sicht des Verwaltungsfensters des Administrators wird ebenfalls aktualisiert. Die Anzeige der aktuellen Daten ist somit gewährleistet. Die Methode *setUserAccount()* der Klasse *Application* realisiert alle Updates der Einträge in der Benutzertabelle.
Bei der Eingabe der Formel im Unterfenster *log* oder *potsec* wird die Formel in die Datenbanktabelle *<Benutzername>Log* oder gegebenenfalls in *<Benutzername>Potsec* eingefügt oder gelöscht. Diese Funktionalitäten werden grundsätzlich von den GUI-Klassen *ChildLog* und *ChildPotsec* unterstützt, die ihrerseits die Aufrufe der Methoden *addToLog()* oder gegebenenfalls *addToPotsec()* der Klasse *Application* vornehmen.
Der Administrator kann Benutzerkonten anlegen, ist aber auch berechtigt, diese Konten zu löschen. Es können nur existierende Benutzerkonten gelöscht werden. Bei einem Löschvorgang eines Accounts wird der entsprechende Eintrag in der Benutzertabelle und die mit dem Benutzernamen des zu löschenden Benutzers referenzierten Tabellen *<Benutzername>Log* und *<Benutzername>Potsec* gelöscht. Die dafür verantwortliche Methode *removeUser()* der Klasse *Application* ist folgendermaßen implementiert:

```
public void removeUser(String login){
try{
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@moria:1521:db1" , "zhang", "xxx");
ResultSet rs = con.createStatement().
```

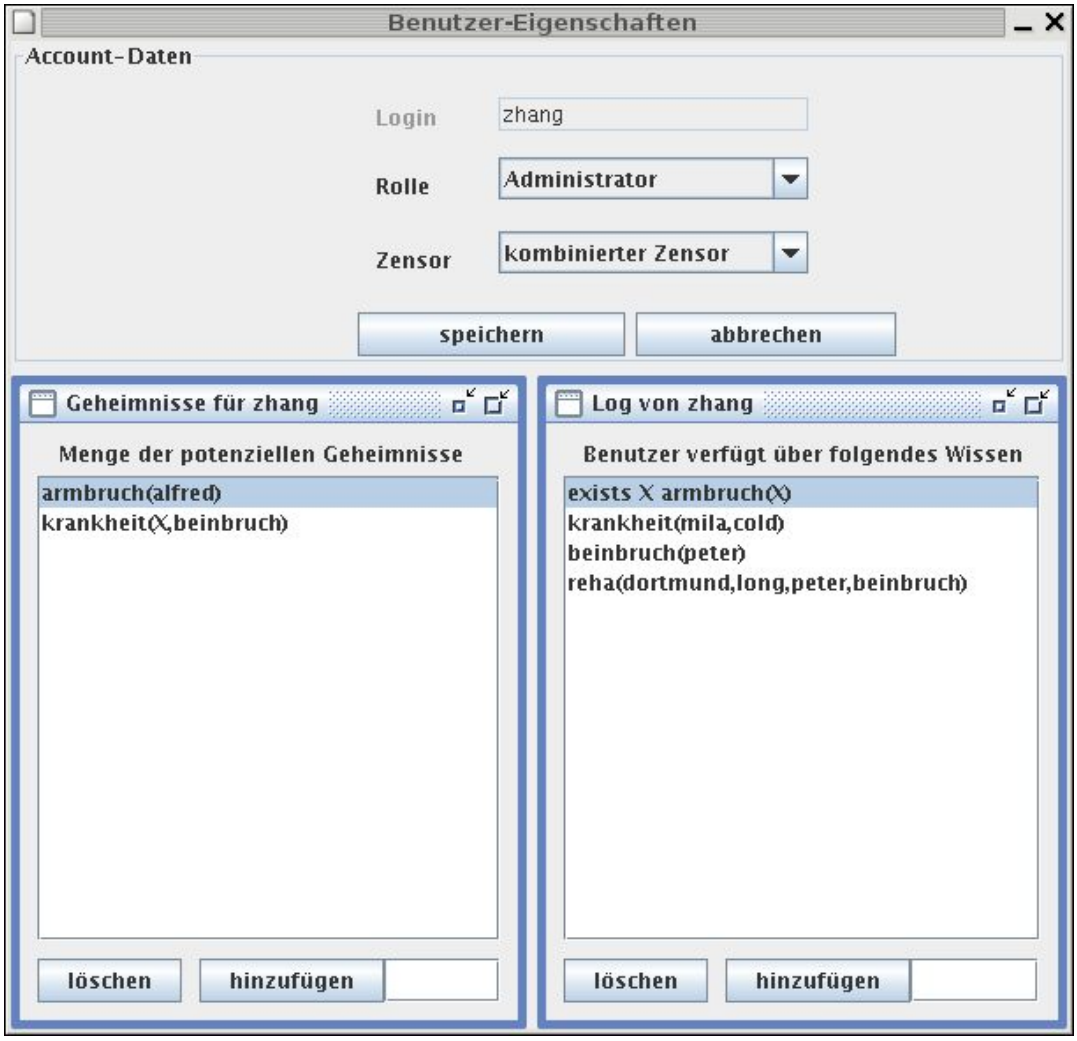

Abbildung 40: Das Benutzereigenschaften-Fenster

```
executeQuery("DELETE FROM benutzertabelle
WHERE login = " + "'" + login + "'");
ResultSet rs1 = con.createStatement().
executeQuery("drop table "+login+"Log");
ResultSet rs2 = con.createStatement().
executeQuery("drop table "+login+"Potsec");
rs1.close();
rs2.close();
rs.close();
con.close();
}
catch (SQLException se) {
System.out.println("SQL Exception: " + se.getMessage());
}
}
```

18.3.3 Verwaltung des Benutzerwissens und der potentiellen Geheimnisse

Die Formeln, die zu der Menge Benutzerwissen und zu der Menge potentieller Geheimnisse gehören, werden in der Klasse *UserData* verwaltet. Anders als in der ersten Iteration werden nun das Benutzerwissen und die potentiellen Geheimnisse in der Datenbank gehalten. Die Speicherung dieser Daten erfolgt in den oben vorgestellten Tabellen *<Benutzername>Log* und *<Benutzername>Potsec*. Beim Starten der Anwendung werden für den angemeldeten Benutzer diese Daten aus der Datenbank ausgelesen und lokal gespeichert. Bei jeder Anforderung des Benutzerwissens und der potentiellen Geheimnisse werden nun die Methoden *getLog()* und *getPotsec()* aufgerufen, die die benötigten Daten aus der Datenbank auslesen und diese in Form einer verketteten Liste, die Objekte vom Typ Formula enthält, zurückliefern. Die Änderung gegenüber dem Entwurf besteht nun darin, dass bei dem Aufruf dieser Methoden der Parameter *loginKey* übergeben werden soll. Dieser Parameter bringt die Möglichkeit zu entscheiden, welche Tabellen zu dem Benutzer gehören. Beispielsweise wird bei einem Aufruf von getLog(„zhang“) auf die Tabelle *zhangLog* in der Datenbank zugegriffen. Falls es sich um einen Ablauf der kontrollierten Anfrageauswertung handelt, werden die Methoden *getLog()* und *getPotsec()* mit dem Benutzernamen des aktuell angemeldeten Benutzers als Parameter aufgerufen, da für diesen Benutzer die kontrollierte Anfrageauswertung durchgeführt werden soll. Dementsprechend wurde die Parameterliste der Methoden, die die Aufrufe von *getLog* und *getPotsec* beinhalten um den Parameter *loginKey* erweitert. Die betroffenen Methoden der Klasse *Application* sind: *addToLog*, *addToPotsec*, *deleteFromLog*, *deleteFromPotsec*, *isInLog* und *isInPotsec*.
Die Speicherung von Formeln aus dem Benutzerwissen un den potentiellen Geheimnissen in der Datenbank ist folgendermaßen implementiert worden. Es wird die Operation *toString()* der Klasse *Formula* auf die Formula Objekte aus dem *log* und *potsec* angewendet. Das Ergebnis wird in der Tabelle gespeichert. Nach dem Auslesen dieser Zeichenkette aus der Tabelle wird das neue Objekt Formula erzeugt, das diese

Zeichenkette als Parameter erhält.

18.3.4 Datenbankverbindung und Tabellensperre

Mit den beiden implementierten Methoden *lock()* und *unlock()* der Klasse *Application* kann eine Sperre auf eine Tabelle verhängt und wieder aufgehoben werden. Somit ist ein exklusiver Zugriff auf die Datenbank-Tabellen für einen Administrator gewährleistet. Die beiden Methoden werden überall an den Stellen im Programm (dessen Aufzählung im Rahmen dieses Dokumentes zu umfangreich wäre) aufgerufen, wo eine Lese- oder Schreiboperation auf die Datenbanktabellen *<Benutzername>Log*, *<Benutzername>Potsec* oder *benutzertabelle* erfolgt.
Anschließend wird die Implementierung des Verbindungsaufbaus mit der Datenbank beschrieben. In der ersten Iteration wurde der Wunsch geäußert, die Anmeldungsdaten für die Oracle-Datenbank nicht im Javaprogramm fest zu kodieren. Die Anmeldungsdaten sollten aus einer externen Datei ausgelesen und für den Verbindungsaufbau benutzt werden. Es ist eine Klasse *DatebaseConf* implementiert worden, die das Auslesen der benötigten Daten aus der Datei *Database.conf* erlaubt. Mit den Methoden *getDBUser()* und *getDBUserPW()* werden der Benutzername und das Passwort zurückgeliefert, die für den Verbindungsaufbau notwendig sind. Die oben genannten Methoden unterstützen diese Anforderung und liefern in der *Database.conf* gespeicherten Anmeldungsdaten zurück.

19 Test

19.1 Einleitung

Die Projektgruppe hat sich in der zweiten Iteration für ein neues Testkonzept entschieden, das aus drei Phasen besteht, dem Modultest, dem Integrationstest und dem Zweigüberdeckungstest.
Es werden alle neu hinzugekommenen nichttrivialen Methoden in einem Modultest auf ihre Korrektheit hin überprüft. Die Teststrategie, die hier benutzt wird, bezeichnet man als „Bottom-Up-Strategie“, d.h. es werden zunächst die „Low-Level-Funktionalitäten“ auf Korrektheit überprüft. In jedem Modultest müssen Äquivalenzklassen definiert werden für gültige und ungültige Eingaben der zu testenden Methode. Anschließend wird anhand dieser Äquivalenzklassen die zu testende Eingabe bestimmt. Es wird jeweils das erwartete und das erhaltene Ergebnis dokumentiert. Als Werkzeug für diesen Test kommt das *JUnit-Framework* zum Einsatz.
In der zweiten Testphase wird ein Integrationstest in Form eines Schnittstellentests durchgeführt. Dabei soll speziell das Zusammenspiel der Methoden untereinander betrachtet werden. Dazu werden meist zwei Methoden betrachtet, die nacheinander aufgerufen werden, und deren Ein- und Ausgabe voneinander abhängen.
Abschließend wird in der dritten Testphase ein Strukturtest durchgeführt, der überprüfen soll, ob für bestimmte Eingaben die in den Aktivitäts- und Sequenzdiagrammen vorgegebenen Strukturen eingehalten werden. Dieser Test soll den korrekten Ablauf bzw. die richtige Reihenfolge beim Methodenaufruf sichern. Der Strukturtest wird für die drei wichtigsten Anwendungsfälle *cqe*, *addToLog* und *addToPot_ sec* durchgeführt.

19.2 Modultest

19.2.1 Testmodul Formula

Formula beinhaltet für das Gesamtprogramm Methoden und Mechanismen von zentraler Bedeutung, da durch sie Formeln dargestellt und manipuliert werden können. Die drei kritischen Methoden *negate*, *toSQL* und *toString* wurden in der letzten Iteration bereits getestet. Während sich Voraussetzungen und Verhalten der Methode *negate* praktisch nicht verändert haben, stellt vor allem die Hinzunahme der Negation zu den Formeln eine neue Herausforderung für die letzten beiden Methoden dar, weswegen diese hier nochmals zu Tests herangezogen werden. Neu hinzugekommen sind die Methoden *demorgan*, *toSRNF*, *rr* und *rrStrong*.

Methode demorgan Die Methode *demorgan* wendet auf eine Formel solange die De Morgan'schen Regeln an, bis Negationen nur noch unmittelbar vor Literalen stehen. Präfixe der Form $\exists x_1 \exists x_2 \ldots \exists x_n$ und $\neg\exists x_1 \exists x_2 \ldots \exists x_n$ bleiben hierbei unberührt, sollen also vom Algorithmus ignoriert werden. Dieser Algorithmus stellt auch den Hauptbestandteil der Umformung von Formeln in SRNF dar, weshalb die Methode *toSRNF* nicht noch zusätzlich getestet werden soll. Die folgenden Äquivalenzklassen erscheinen für den Test relevant.

Äquivalenzklassen

Gültige Eingaben

1. Alle Formeln, die keine Negationen enthalten.
2. Alle Formeln, die keine Quantoren enthalten und bei denen Negationen bereits nur noch vor Literalen stehen.
3. Alle Formeln, die keine Quantoren enthalten und bei denen Negationen an beliebigen Stellen auftreten.
4. Alle Formeln, die Existenzquantoren enthalten und bei denen Negationen bereits nur noch vor Literalen stehen.
5. Alle Formeln, die negierte Existenzquantoren enthalten und bei denen Negationen bereits nur noch vor Literalen stehen.
6. Alle Formeln, die Existenzquantoren enthalten und bei denen Negationen an beliebigen Stellen auftreten.
7. Alle Formeln, die negierte Existenzquantoren enthalten und bei denen Negationen an beliebigen Stellen auftreten.

Ungültige Eingaben Ungültig als Eingaben sind alle Formeln, die der definierten Syntax für Anfragen, dem Benutzerwissen und den potentiellen Geheimnissen nicht genügen. Die syntaktische Korrektheit der Formeln wird aber bereits bei deren Erzeugung überprüft, weshalb hier ungültige Eingaben gar nicht erst auftreten können.

Getestete Eingaben

1. $armbruch(alfred) \wedge (beinbruch(alfred) \vee krankheit(hans, cold))$
2. $\neg armbruch(alfred) \wedge (beinbruch(alfred) \vee \neg krankheit(hans, cold))$
3. $armbruch(alfred) \wedge \neg(beinbruch(alfred) \vee krankheit(hans, cold))$
4. $\exists X \exists Y \neg armbruch(X) \wedge (beinbruch(Y) \vee \neg krankheit(hans, cold))$
5. $\neg\exists X \exists Y armbruch(Y) \wedge (\neg beinbruch(X) \wedge krankheit(X, cold))$
6. $\exists X \exists Y armbruch(Y) \wedge \neg(beinbruch(X) \wedge \neg krankheit(X, cold))$
7. $\neg\exists X armbruch(X) \wedge \neg(beinbruch(X) \vee \neg krankheit(X, cold))$

Es wurde zu jedem Test der *queryParser* mit dem entsprechenden String für die Formeln initialisiert und ein Objekt f vom Typ Formula erzeugt. Daraus wurde dann mittels der Methode *demorgan* ein neues Objekt f_1 erzeugt, das die aus der zu testenden Methode resultierende Formel darstellt. Mittels der Assertion

```
Assert.assertEquals(f1.toString(), <erwarteter String>);
```

wurde dann die Korrektheit der Ausgabe geprüft.

Erwartete Ausgaben Folgende, zu den Ausgangsformeln äquivalente, Formeln muss ein korrekter Algorithmus liefern:

1. $armbruch(alfred) \wedge (beinbruch(alfred) \vee krankheit(hans, cold))$
2. $\neg armbruch(alfred) \wedge (beinbruch(alfred) \vee \neg krankheit(hans, cold))$
3. $armbruch(alfred) \wedge (\neg beinbruch(alfred) \wedge \neg krankheit(hans, cold))$
4. $\exists X \exists Y \neg armbruch(X) \wedge (beinbruch(Y) \vee \neg krankheit(hans, cold))$
5. $\neg\exists X \exists Y armbruch(Y) \wedge (\neg beinbruch(X) \wedge krankheit(X, cold))$
6. $\exists X \exists Y armbruch(Y) \wedge (\neg beinbruch(X) \vee krankheit(X, cold))$
7. $\neg\exists X armbruch(X) \wedge (\neg beinbruch(X) \wedge krankheit(X, cold))$

Beobachtete Ausgaben Alle Eingaben führten zu den erwarteten Ausgaben, es schlug also kein Test fehl.

Methode toSRNF Durch Aufruf dieser Methode soll eine zu dieser Formel äquivalente Formel in SRNF zurückgegeben werden. Die von uns erlaubten Formeln halten, bis auf das Vorhandensein von Allquantoren und dass Negationen nur vor Literalen stehen dürfen, bereits alle syntaktischen Bedingungen an die SRNF ein. Die Anwendung der De Morgan'schen Regeln wird von der Methode *demorgan* geleistet. Also muss diese Methode zuvor nur noch prüfen, ob wir es mit einer existenzquantifizierten oder allquantifizierten Formel zu tun haben. Im zweiten Fall reagiert die Methode damit, dass vor den Rest der Formel eine Negation hinzugefügt wird. Die Allquantoren bleiben stehen, werden aber programmintern wie negierte Existenzquantoren behandelt. Das korrekte Verhalten dieser Methode konnte schon während ihrer Entwicklung ausgiebig getestet werden, weshalb von einem Modultest an dieser Stelle abgesehen wird.

Methode rr Diese Methode soll für die aufrufende Formel eine Liste liefern, die alle freien Variablen enthält, die innerhalb der Formel range restricted sind. Wenn in der Formel mindetens eine gebundene Variable nicht range restricted ist, so wirft sie eine Exception. Die entsprechenden Äquivalenzklassen lassen sich hieraus konstruieren.

Äquivalenzklassen

Gültige Eingaben

- Alle Formeln, die keine Variablen enthalten.
- Alle Formeln, die ausschließlich gebundene Variablen enthalten, die range restricted sind.
- Alle Formeln, die gebundene und freie Variablen enthalten, die alle range restricted sind.
- Alle Formeln, die gebundene und freie Variablen enthalten, wobei alle gebundenen aber nicht alle ungebundenen Variablen range restricted sind.

Die letzten beiden Fälle können in der aktuellen Phase des Programms zwar nicht auftreten, sollen aber vorausgreifend bereits jetzt getestet werden, da freie Variablen für die Zukunft geplant sind. Die Funktionsfähigkeit für diese Fälle muss dann später nicht mehr getestet werden.

Ungültige Eingaben Ungültig als Eingaben für diese Methode sind alle Formeln, die nicht in SRNF vorliegen. Es wird aber stets die Umformung in SRNF vorgenommen, bevor der Algorithmus *rr* angewendet wird. Der Test dieses Moduls ist weiter oben bereits abgeschlossen worden. Bei Eingabe einer Formel, die gebundene Variablen enthält, die nicht range restricted sind, wirft die Methode *rr* eine *Exception*, weshalb solche Eingaben im Sinne des Test als ungültig angesehen werden können.

Getestete Eingaben Gültige Eingaben:

1. $armbruch(alfred) \wedge (beinbruch(alfred) \vee krankheit(hans, cold))$
2. $\exists X \exists Y armbruch(Y) \wedge (\neg beinbruch(X) \wedge krankheit(X, cold))$
3. $\exists X armbruch(X) \wedge (\neg beinbruch(Y) \wedge krankheit(Y, cold))$
4. $\exists X armbruch(X) \wedge (\neg beinbruch(Y) \wedge krankheit(hans, cold))$

Ungültige Eingaben:

1. $\exists X armbruch(alfred) \vee (\neg beinbruch(X) \wedge krankheit(hans, cold))$

Erwartete Ausgaben Die zurückgegebene *LinkedList* wurde in einer Variablen l gespeichert. In den Fällen 1, 2 und 4 bei den gültigen Eingaben muss der Algorithmus eine leere *LinkedList* als Ergebnis liefern. Deswegen ist hier folgende Assertion praktikabel:

```
Assert.assertTrue(l.size()==0);
```

Im Fall 3 muss das Ergebnis eine einelementige *LinkedList* mit dem Eintrag "Y" sein. Hierzu wurden zwei Assertions hintereinandergeschaltet:

```
Assert.assertTrue(l.size()==1);
Assert.assertEquals(l.get(0), "Y");
```

Für die ungültige Eingabe muss der Algorithmus mit einer Ausnahme abbrechen. Die Meldung der Ausnahme muss lauten: „Gebundene Variable ist nicht range restricted".

Beobachtete Ausgaben Jeder durchgeführte Test führte zur jeweils erwarteten Ausgabe.

Methode rrStrong Diese Methode prüft die aufrufende Formel auf eine syntaktisch noch etwas stärkere Einschränkung als safe rangeness. Hierbei müssen negierte Literale immer unmittelbar konjugiert sein mit positiven Literalen, die mindestens die gleichen Variablen enthalten. In Ermangelung eines besseren Begriffs wurde diese Eigenschaft von der PG „stark safe range" getauft.

Äquivalenzklassen

Gültige Eingaben

1. Alle Formeln, die safe range und stark safe range sind.
2. Alle Formeln, die safe range aber nicht stark safe range sind.

Ungültige Eingaben Ungültig sind alle Formeln, die nicht safe range sind. Der Algorithmus *rrStrong* wird aber immer nur auf Formeln angewendet, die den Test auf safe rangeness bereits bestanden haben. Die Methode *rr* wurde wiederum bereits weiter oben getestet.

Getestete Eingaben

1. $\exists X armbruch(alfred) \wedge (beinbruch(X) \wedge \neg krankheit(X, cold))$
2. $\exists X \neg armbruch(X) \wedge (beinbruch(X) \wedge krankheit(hans, cold))$

Erwartete Ausgaben Für die erste Formel muss das Ergebnis bei korrekt arbeitendem Algorithmus *true* sein. Im zweiten Fall muss $false$ herauskommen. Dazu wurden für zwei Formeln f und g die folgenden zwei Assertions getestet:

```
Assert.assertTrue(f.rrStrong(f.getSyntaxTree()));
Assert.assertTrue(!g.rrStrong(g.getSyntaxTree()));
```

Beobachtete Ausgaben Beide Tests wurden erfolgreich abgeschlossen.

Methode toSQL Die Methode *toSQL* hat innerhalb des Programms eine zentrale Bedeutung. Wie Anfragen des Relationenkalküls in SQL Anfragen übersetzt werden, war eines der am schwierigsten zu lösenden Probleme beim Entwurf und der Implementierung. Innerhalb des Algorithmus müssen viele Details beachtet werden. So müssen ständig Schemainformationen, Informationen über die Struktur der Anfragen, die vorkommenden Variablen und Konstanten etc. abgefragt werden. Entsprechend komplex ist der Algorithmus dann auch in der Implementierung geworden, was offensichtlich das Fehlerpotential stark erhöht. Im Folgenden werden die zum Test verwendeten Äquivalenzklassen definiert.

Äquivalenzklassen

Gültige Eingaben

1. Alle Anfragen, die keine Variablen und keine Negationen enthalten.
2. Alle Anfragen, die keine Variablen, aber Negationen enthalten.
3. Alle Anfragen, die keine Negationen, aber existenzquantifizierte Variablen enthalten.
4. Alle Anfragen, die keine Negationen, aber negativ existenzquantifizierte Variablen enthalten.

5. Alle Anfragen, die Negationen und existenzquantifizierte Variablen enthalten.
6. Alle Anfragen, die Negationen und negativ existenzquantifizierte Variablen enthalten.

Ungültige Eingaben Ungültig sind hier alle Formeln, die nicht stark safe range sind. Bei ihrer Erzeugung werden Formeln allerdings sofort auf diese Eigenschaft hin überprüft und im negativen Falle zurückgewiesen. Deshalb kommt es nur zur Anwendung der Methode *toSQL*, wenn dieser Test bestanden wurde. Dieser Test beinhaltet natürlich auch die Umformung der Formel in SRNF, was ebenfalls Voraussetzung für die Umformung in SQL ist.

Getestete Eingaben

1. $armbruch(alfred) \wedge beinbruch(gerd)$
2. $\neg armbruch(alfred) \vee \neg beinbruch(gerd)$
3. $\exists X armbruch(X) \wedge beinbruch(X)$
4. $\neg\exists X armbruch(X) \vee beinbruch(X)$
5. $\exists X \neg armbruch(X) \wedge beinbruch(X)$
6. $\neg\exists X armbruch(alfred) \vee (beinbruch(X) \wedge \neg krankheit(X, cold))$

Erwartete Ausgaben Die *Strings* müssen folgendermaßen aussehen:

1. ```
   SELECT 'c0' FROM DUAL WHERE EXISTS ((SELECT * FROM
   (SELECT 'c1' FROM DUAL WHERE EXISTS (SELECT * FROM armbruch
   WHERE NAME='alfred')) NATURAL JOIN (SELECT 'c2' FROM DUAL
   WHERE EXISTS (SELECT * FROM beinbruch WHERE NAME='gerd'))))
   ```
2. ```
   SELECT 'c0' FROM DUAL WHERE EXISTS ((SELECT * FROM
   (SELECT 'c1' FROM DUAL WHERE NOT EXISTS (SELECT * FROM armbruch
   WHERE NAME='alfred')) NATURAL FULL OUTER JOIN (SELECT 'c2'
   FROM DUAL WHERE NOT EXISTS
   (SELECT * FROM beinbruch WHERE NAME='gerd'))))
   ```
3. ```
   SELECT 'c0' FROM DUAL WHERE EXISTS ((SELECT * FROM
   (SELECT NAME AS X FROM armbruch) NATURAL JOIN
   (SELECT NAME AS X FROM beinbruch)))
   ```
4. ```
   SELECT 'c0' FROM DUAL WHERE NOT EXISTS ((SELECT * FROM
   (SELECT NAME AS X FROM armbruch)
   NATURAL FULL OUTER JOIN (SELECT NAME AS X FROM beinbruch)))
   ```

5. `SELECT 'c0' FROM DUAL WHERE EXISTS (SELECT NAME AS X FROM beinbruch 'c2' WHERE NAME NOT IN (SELECT NAME FROM armbruch 'c1' WHERE c1.NAME=c2.NAME))`

6. `SELECT 'c0' FROM DUAL WHERE NOT EXISTS ((SELECT * FROM (SELECT 'c1' FROM DUAL WHERE EXISTS (SELECT * FROM armbruch WHERE NAME='alfred')) NATURAL JOIN (SELECT NAME AS X FROM beinbruch 'c3' WHERE NAME NOT IN (SELECT NAME FROM krankheit 'c2' WHERE c2.NAME=c3.NAME AND c2.KRANKHEIT='cold'))))`

Beobachtete Ausgaben

1. Es fiel ein Fehler bei der fortlaufenden Nummerierung der Konstanten c_i auf. Da an einer Stelle im Quelltext vergessen wurde, den Zähler zu inkrementieren, wurde die Konstante c_1 zwei Mal verwendet. Der Fehler wurde direkt behoben. Ansonsten entsprach die Ausgabe dem erwarteten String.

2. `SELECT 'c0' FROM DUAL WHERE EXISTS ((SELECT * FROM (SELECT 'c1' FROM DUAL WHERE NOT EXISTS (SELECT * FROM armbruch W)) NATURAL FULL OUTER JOIN (SELECT 'c1' FROM DUAL WHERE NOT EXISTS (SELECT * FROM beinbruch W))))`
   An dieser Stelle fallen zwei Fehler auf: Zum einen ist der Zähler wieder nicht richtig inkrementiert worden. Zum anderen sind in den inneren Ausdrücken zu den Literalen die Strings abgeschnitten. Beide Fehler ließen sich aber schnell finden. Der erste Fehler hatte wieder eine fehlende Inkrementierung der Zählervariablen als Grund. Der zweite Fehler lag daran, dass beim Aufbau des Strings in einer Schleife, in der die Konstanten mit der *WHERE-Klausel* an die entsprechenden Werte gebunden werden, eine falsche Abbruchbedingung benutzt wurde. Anstatt der Abbruchbedingung $<$ musste $\leq$ verwendet werden.

3. Dieser Test verlief erfolgreich.

4. Dieser Test verlief ebenfalls erfolgreich.

5. `SELECT 'c0' FROM DUAL WHERE EXISTS (SELECT NAME AS X FROM beinbruch c2 WHERE NAME NOT IN (SELECT NAME FROM armbruch c1 WHERE c1.NAME=c2.null))`
   Hier fallen wieder zwei Fehler auf: Die Konstanten c_1 und c_2 stehen nicht in Hochkommas, so wie es eigentlich sein müsste. Und im inneren Ausdruck findet sich der Teilstring `c1.NAME=c2.null`, was offensichtlich falsch ist.

6. Dieser Test war wiederum erfolgreich.

Methode toString Hier soll eine String-Repräsentation der Formel zurückgegeben werden. Diese Methode funktionierte in der letzten Iteration einwandfrei. Sie soll hier aber dennoch nochmals getestet werden, da nun auch Negationen auftreten können.

Äquivalenzklassen

Gültige Eingaben Die einzigen gültigen Eingaben, die hier nochmals getestet werden sollen, sind solche, die Negationen enthalten.

Ungültige Eingaben Es sind alle nach Anfragesprache, Benutzerwissen oder potentiellen Geheimnissen syntaktisch korrekten Formeln erlaubt. Formeln, die dies nicht einhalten, werden bereits von den Parsern zurückgewiesen.

Getestete Eingaben

1. $\exists X \neg armbruch(alfred) \vee (\neg beinbruch(X) \wedge krankheit(X, cold))$
2. $\exists X \neg(((armbruch(X) \wedge beinbruch(X)) \wedge krankheit(h, c)) \wedge \neg armbruch(g))$

Erwartete Ausgaben Die korrespondierenden Strings müssen folgendermaßen aussehen:

1. `EXISTS X ((NOT armbruch(alfred)) OR ((NOT beinbruch(X)) AND krankheit(X,cold)))`
2. `EXISTS X (NOT (((armbruch(X) AND beinbruch(X)) AND krankheit(h,c)) AND (NOT armbruch(g))))`

Beobachtete Ausgaben Die Ausgabe der Strings entsprach den Erwartungen.

19.2.2 Testmodul ProverNine

Methode isImpliedForLying Die Methode *isImpliedForLying* wurde implementiert um Formeln der Form
$log \cup \{eval^*(\Phi)(db)\} \models pot\_sec\_disj$
mit dem Theorembeweiser *Prover9* auf ihren Wahrheitswert überprüfen zu können. Der Unterschied zur bereits vorhandenen Methode *isImplied* liegt darin, dass aus einer Liste von Teilformeln, die im dritten Übergabeparameter übergeben wird, eine Disjunktion dieser Teilformeln erzeugt wird.

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die gültigen Eingaben der Methode

```
isImpliedForLying(LinkedList log, Formula query, LinkedList pot_sec)
```

bestehen aus einer Liste aus Formeln *log*, einer Anfrage-Formel *query* und einer Liste aus Formeln *pot_ sec*. Dabei entsprechen die Formeln gültigen Formeln aus der Klasse *Formula*. In den Listen sollte mindestens ein Element enthalten sein und die Anfrage sollte ebenfalls nicht leer sein.
Es werden dabei folgende Äquivalenzklassen unterschieden:

1. Die Anfrage impliziert ein potentielles Geheimnis.
2. Es wird kein potentielles Geheimnis impliziert.
3. Das Benutzerwissen impliziert ein potentielles Geheimnis.

Ungültige Eingaben Die Äquivalenzklassen für die ungültigen Eingaben bestehen aus Aufrufen der Methode *isImpliedForLying* mit leeren Listen oder leerer Anfrage. Diese können nach der internen Struktur des Programms aber nicht übergeben werden, da eine Formel zuvor von einem Parser geprüft wird und leere Formeln nicht erlaubt sind. Daher entfällt hier der Test mit ungültigen Eingaben.

Getestete Eingaben Es wurden die folgenden Eingaben aus der Äquivalenzklasse der gültigen Eingaben getestet:

1. *(log={armbruch(fred)}, query={armbruch(horst)}, pot_ sec={armbruch(horst)})*
2. *(log={armbruch(fred)}, query={armbruch(horst)}, pot_ sec={armbruch(moni), armbruch(juergen), armbruch(dieter)})*
3. *(log={armbruch(fred)}, query={armbruch(horst)}, pot_ sec={armbruch(moni), armbruch(juergen), armbruch(dieter), armbruch(fred)})*

Erwartete Ausgaben Folgende Ausgaben sind dabei zu erwarten:

1. *true*
2. *false*
3. *true*

Beobachtete Ausgaben Die beobachteten Ausgaben

1. *true*
2. *false*
3. *true*

decken sich mit den erwarteten Ausgaben, daher kann der Test der Methode *isImpliedForLying* positiv abgeschlossen werden.

Methode isImpliedForOptimization Die Methode *isImpliedForOptimization* wurde implementiert, um Formeln der Form
$set1\_disj \vee \psi \models set2\_disj$
mit dem Theorembeweiser *Prover9* auf ihren Wahrheitswert überprüfen zu können. Dabei stehen die Ausdrücke $set1\_disj$ und $set2\_disj$ für eine Menge aus Teilformeln, welche disjunktiv zusammengesetzt wurden und ψ für eine Teilformel.
Der Unterschied zur bereits im Abschnitt 19.2.2 erwähnten Methode *isImpliedForLying* liegt darin, dass nun nicht nur die rechte Seite des Ausdrucks aus einer Disjunktion von Teilformeln bestehen kann, sondern zusätzlich auch noch die linke Seite.

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die gültigen Eingaben der Methode

```
isImpliedForOptimization(LinkedList set1, Formula psi, LinkedList set2)
```

bestehen aus einer Liste aus Formeln *set1*, einer Formel ψ und einer weiteren Liste aus Formeln *set2*. Dabei entsprechen die Formeln gültigen Formeln aus der Klasse *Formula*. In den Listen sollte mindestens ein Element enthalten sein. Es werden dabei folgende Äquivalenzklassen unterschieden:

1. Die Disjunktion der Formelmenge $set1\_disj$ zusammen mit ψ impliziert nicht die Disjunktion der Formelmenge $set2\_disj$.
2. Die Disjunktion der Formelmenge $set1\_disj$ zusammen mit ψ impliziert die Disjunktion der Formelmenge $set2\_disj$..

Ungültige Eingaben Die Äquivalenzklassen für die ungültigen Eingaben bestehen aus Aufrufen der Methode *isImpliedForOptimization* mit leeren Listen. Diese können nach der internen Struktur des Programms aber nicht übergeben werden, da eine Formel zuvor von einem Parser geprüft wird und leere Formeln nicht erlaubt sind. Daher entfällt hier, wie auch im vorhergehenden Abschnitt, der Test mit ungültigen Eingaben.

Getestete Eingaben Es wurden die folgenden Eingaben aus der Äquivalenzklasse der gültigen Eingaben getestet:

1. *(set1={armbruch(fred), ∃ X beinbruch(X)}, ψ={armbruch(fred)}, set2={armbruch(moni), armbruch(juergen), armbruch(dieter)})*
2. *(set1={beinbruch(moni)}, ψ={beinbruch(fred)}, set2={∃ X beinbruch(X)})*

Erwartete Ausgaben Folgende Ausgaben sind dabei zu erwarten:

1. $false$
2. $true$

Beobachtete Ausgaben Die beobachteten Ausgaben

1. $false$
2. $true$

decken sich mit den erwarteten Ausgaben, daher kann der Test der Methode *isImpliedForOptimization* positiv abgeschlossen werden.
Anmerkung: Es wurde ein kleiner Fehler beim Übersetzen des Ausdrucks in die Sprache des Theorembeweisers gefunden. Es handelte sich dabei um einen Klammerungsfehler, der direkt verbessert werden konnte.

19.3 Schnittstellentest

19.3.1 Methoden rr und toSRNF

Die Methode *toSRNF* wird zuerst eine Formel f in safe-range Normalform (*SRNF*) umwandeln. Dann wird der Wurzelknoten dieser in *SRNF* umgewandelten Formel mit Hilfe von der Methode *getSyntaxTree* der Klasse *Formula* ausgegeben. Dieser Wurzelknoten wird als Parameter in der Methode *rr* eingegeben. Darin besteht ein Schnittstellentest zwischen den beiden Methoden.

Eingaben für den Schnittstellentest

Unkritische Eingaben In diesem Fall wird eine Anfrage ohne freie Variablen eingegeben.

$$exists\, X\, armbruch(X);$$

Das erwartete Ergebnis ist, dass keine Ausnahme ausgeworfen wird, und die leere Liste ausgegeben wird.

Randwertprüfung In diesem Fall wird eine Anfrage mit freien Variablen eingegeben.

$$exists\, X\, (armbruch(X)\ or\ (not\ beinbruch(Y)));$$

Das erwartete Ergebnis ist eine von Methode *rr* ausgeworfene Ausnahme.

Testergebnis Der Schnittstellentest ist erfolgreich.

19.3.2 Methoden rr und rrStrong

Im Folgenden wird ein Schnittstellentest für die Methoden *rr* und *rrStrong* durchgeführt. Dabei werden die Methoden mit Formeln aufgerufen, die:

1. *safe-range* und stark *safe-range*,
2. *safe-range* und nicht stark *safe-range* und
3. nicht *safe-range*

sind.

Eingaben für den Schnittstellentest

Unkritische Eingaben (1.) Folgende Eingabe ist erwartungsgemäß unproblematisch im Hinblick auf das Zusammenspiel beider Methoden *rr* und *rrStrong*:
exists X (krankheit(X, armbruch) and (not beinbruch(X)))

Erwartete Ausgaben (1.) Die oben genannte Formel ist sowohl *safe-range*, als auch stark *safe-range*. Wir erwarten daher einen fehlerfreien Durchlauf.

Beobachtete Ausgaben (1.) Der Aufruf beider Methoden führt zu keinem Fehler, somit ist die Formel nicht nur *safe-range* sondern auch stark *safe-range*.

Randwertprüfung (2.) Eine problematische Eingabe wäre eine Eingabe, die *safe-range* ist, aber nicht stark *safe-range*, z.B.:
exists X (krankheit(X, armbruch) or (not beinbruch(X)))

(3.) Oder aber eine Eingabe, die nicht *safe-range* ist, z.B.:
exists X ((krankheit(X) and armbruch(X)) and (not beinbruch(X)))

Erwartete Ausgaben (2.) Da die Formel *safe-range* ist, sollte die Methode *rr* dies auch feststellen könne. Weil die Formel aber nicht stark *safe-range* ist, sollte die Methode *rrStrong* den Wert $false$ zurückgeben.

(3.) Da die Formel nicht *safe-range* ist, sollte die Methode *rr* eine Exception werfen. Hinweis: Eine Exception wird genau dann geworfen, wenn eine gebundene Variable nicht *safe-range* ist.

Beobachtete Ausgaben (2.) Die Methode *rr* gibt aus, dass die Formel *safe-range* ist. Die Methode *rrStrong* liefert wie erwartet den Wert $false$ zurück.

(3.) Die Methode *rr* wirft wie erwartet eine Exception, weil die gebundene Variable X nicht *safe-range* ist.

Weitere Besonderheiten der Schnittstelle Beim Testen fiel ein Fehler in der Methode *rrStrong* auf. Sofern das negierte Literal sich nicht an der linkesten Stelle in der Formel befand, kam es in der Methode *rrStrong* zu einem Fehler.
Es wurde zwar beispielsweise die Formel
exists X (krankheit(X, armbruch) and (not beinbruch(X)))
akzeptiert, die äquivalente Formel
exists X ((not beinbruch(X)) and krankheit(X, armbruch))
hingegen nicht. Dieser Fehler wurde während des Testens behoben, so dass nun wie gewünscht beide Möglichkeiten akzeptiert werden.

19.3.3 Methoden isPreconditionForLog und isPreconditionForPot_sec

Die Methoden *isPreconditionForLog* und *isPreconditionForPot_sec* werden getestet, um sicherzustellen, dass das gewünschte Verhalten der Schnittstelle eingehalten wird. Das gewünschte Verhalten ist hier, dass die Vorbedingung für den jeweiligen Zensor beim Einfügen von Benutzerwissen und potentiellen Geheimnissen erhalten bleibt.

Eingaben für den Schnittstellentest

Unkritische Eingaben Die Äquivalenzklassen für die unkritischen Eingaben der Methoden

```
isPreconditionForLog(LinkedList log,
   Formula lognew, LinkedList pot_sec)
```

und

```
isPreconditionForPot_sec(LinkedList log,
   Formula potnew, LinkedList pot_sec)
```

entsprechen den Äquivalenzklassen für die gültigen Eingaben der Methode

```
isImplied(LinkedList log, Formula query, LinkedList pot_sec).
```

Im Rumpf der beiden Methoden wird nur die Methode *isImplied* aufgerufen, die die Parameterliste von den beiden Methoden direkt übergeben bekommt. Deswegen bestehen die Äquivalenzklassen aus einer Liste von Formeln *log*, einer Formel (*lognew* bzw. *potnew*) und einer Liste aus Formeln *pot_sec*. Dabei gelten gleiche Regeln für die Formeln und für die Elemente aus jeder Liste wie bei *isImplied*.

Getestete Eingaben Es wurden die folgenden Eingaben aus der Äquivalenzklasse der unkritischen Eingaben für die Methode *isPreconditionForLog* getestet:

1. $log = \{armbruch(alfred), armbruch(hans), beinbruch(toni)\}$,
   $lognew = \{beinbruch(bert)\}$,
   $pot\_sec = \{armbruch(mike), armbruch(poncho), beinbruch(fred),$
   $armbruch(horst)\}$
2. $(log = \{armbruch(alfred), armbruch(hans), beinbruch(toni\}$,
   $lognew = \{beinbruch(fred)\}$,
   $pot\_sec = \{armbruch(mike), armbruch(poncho), beinbruch(fred),$
   $armbruch(horst)\})$

Es wurden die folgenden Eingaben aus der Äquivalenzklasse der unkritischen Eingaben für die Methode *isPreconditionForPot_sec* getestet:

1. $log = \{armbruch(alfred), armbruch(hans), beinbruch(toni),$
   $not\ beinbruch(alfred)\},$
   $potnew = \{beinbruch(bert)\},$
   $pot\_sec = \{armbruch(mike), armbruch(poncho), beinbruch(fred)\}$
2. $log = \{armbruch(alfred), armbruch(hans), beinbruch(toni),$
   $not\ beinbruch(alfred)\},$
   $potnew = \{beinbruch(toni)\},$
   $pot\_sec = \{armbruch(mike), armbruch(poncho), beinbruch(fred)\}$

Erwartete Ausgaben Folgende Ausgaben sind für die beiden Methoden zu erwarten:

1. $true$
2. $false$

Beobachtete Ausgaben Die beobachteten Ausgaben

1. $true$
2. $false$

decken sich mit den erwarteten Ausgaben, daher kann der Test hier positiv abgeschlossen werden.

Randwertprüfung Die kritischen Eingaben für die beiden Methoden werden durch die ungültige Eingaben für *isImplied* abgedeckt. Deswegen werden sie hier nicht weiter erläutert.

19.3.4 Methode normalizeForLog

Die Methode *normalizeForLog* ist eine Schnittstelle zwischen den Klassen *Application* und *TreeNode*. Die Methode löst die Konjunktionen innerhalb einer Formel in Teilformeln auf, sofern bestimmte Bedingungen gelten. Ohne eine korrekte Normalisierung kann die Minimalität des Benutzerwissens nicht gesichert werden. Die Schnittstelle *normalizeForLog* wird mit verschiedenen Eingaben getestet.

Eingaben für den Schnittstellentest Die Eingaben dürfen All- und Existenzquantoren enthalten. Konjunktionen und Disjunktionen sind auch bei den Eingaben erlaubt. Die Methode *normalizeForLog* soll nur die Eingaben normalisieren, die kein „Oder" bzw. keinen „Existenzquantor" enthalten.
Folgende Eingaben wurden beim Schnittstellentest verwendet:

1. $armbruch(fred) \wedge armbruch(alfred)$
2. $armbruch(fred) \vee (armbruch(alfred) \wedge beinbruch(fred))$
3. $armbruch(fred) \wedge (beinbruch(alfred) \wedge beinbruch(fred))$
4. $armbruch(alfred) \vee beinbruch(alfred)$
5. $armbruch(fred) \wedge (armbruch(alfred) \vee beinbruch(alfred))$
6. $\forall X(armbruch(X) \wedge beinbruch(X))$
7. $\forall X(armbruch(X) \wedge beinbruch(fred))$
8. $\forall X, Y(armbruch(X) \wedge beinbruch(Y))$
9. $\forall X, Y(armbruch(X) \wedge krankheit(X, Y))$
10. $\forall X(armbruch(X) \vee beinbruch(gerd))$
11. $\exists X(armbruch(X) \wedge beinbruch(X))$

Unkritische Eingaben Unkritische Eingaben sind die Formeln, die aus nur einem Literal bestehen.

Erwartete Ausgaben Folge Ausgaben werden der Reihe nach erwartet:

1. $armbruch(fred),\ armbruch(alfred)$
2. $armbruch(fred) \vee (armbruch(alfred) \wedge beinbruch(fred))$ (unverändert)
3. $armbruch(fred),\ beinbruch(alfred), beinbruch(fred)$

4. $armbruch(alfred) \vee beinbruch(alfred)$ (unverändert)
5. $armbruch(fred),\ (armbruch(alfred) \vee beinbruch(alfred))$
6. $\forall X armbruch(X),\ \forall X beinbruch(X)$
7. $\forall X armbruch(X),\ beinbruch(fred)$
8. $\forall X armbruch(X),\ \forall Y beinbruch(Y)$
9. $\forall X armbruch(X),\ \forall X, Y krankheit(X, Y)$
10. $\forall X(armbruch(X) \vee beinbruch(gerd))$ (unverändert)
11. $\exists X(armbruch(X) \wedge beinbruch(X))$ (unverändert)

Beobachtete Ausgaben Die erwarteten Ausgaben werden alle korrekt ausgegeben. Es sind keine falschen oder unerwarteten Ausgaben aufgetreten.

Randwertprüfung Die Eingaben, die disjunktiv verbunden werden, könnten falsch normalisiert werden. Z.B.:
$armbruch(fred) \vee (armbruch(alfred) \wedge beinbruch(fred))$
Es wurde aber richtig normalisiert und folgende Ausgabe ist raus gekommen:
$armbruch(fred) \vee (armbruch(alfred) \wedge beinbruch(fred))$
Die Eingabe wurde nicht normalisiert und unverändert ausgegeben.

19.3.5 Methode normalizeForPot_sec

Die Methode *normalizeForPot_sec* ist eine Schnittstelle zwischen den Klassen *Application* und *TreeNode*. Die Methode löst die Disjunktionen innerhalb einer Formel in Teilformeln auf, sofern bestimmte Bedingungen gelten. Ohne eine korrekte Normalisierung kann die Minimalität der Menge der potentiellen Geheimnissen nicht gesichert werden. Die Schnittstelle *normalizeForPot_sec* wird mit verschiedenen Eingaben getestet:

Eingaben für den Schnittstellentest Die Eingaben dürfen Existenzquantoren enthalten. Konjunktionen und Disjunktionen sind auch bei der Eingaben erlaubt.
Folgende Eingaben wurden beim Schnittstellentest verwendet:

1. $armbruch(fred) \vee armbruch(alfred)$
2. $armbruch(fred) \wedge (armbruch(alfred) \vee beinbruch(fred))$
3. $armbruch(fred) \vee (beinbruch(alfred) \vee beinbruch(fred))$
4. $armbruch(fred) \vee (armbruch(alfred) \wedge beinbruch(alfred))$
5. $armbruch(alfred) \wedge beinbruch(alfred)$
6. $\exists X(armbruch(X) \vee beinbruch(fred))$
7. $\exists X(armbruch(X) \vee beinbruch(X))$
8. $\exists X \exists Y(armbruch(X) \vee beinbruch(Y))$
9. $\exists X, Y(armbruch(X) \vee krankheit(X, Y))$
10. $\exists X(armbruch(X) \wedge beinbruch(X))$

Unkritische Eingaben Unkritische Eingaben sind die Formeln, die aus nur einem Literal bestehen.

Erwartete Ausgaben Folgende Ausgaben werden der Reihe nach erwartet:

1. $armbruch(fred),\ armbruch(alfred)$
2. $armbruch(fred) \wedge (armbruch(alfred) \vee beinbruch(fred))$ (unverändert)
3. $armbruch(fred),\ beinbruch(alfred), beinbruch(fred)$
4. $armbruch(fred),\ armbruch(alfred) \wedge beinbruch(alfred)$
5. $armbruch(alfred) \wedge beinbruch(alfred)$ (unverändert)

6. $\exists X armbruch(X),\ beinbruch(fred)$
7. $\exists X armbruch(X),\ \exists X beinbruch(X)$
8. $\exists X armbruch(X),\ \exists Y beinbruch(Y)$
9. $\exists X armbruch(X),\ \exists X \exists Y krankheit(X, Y)$
10. $\exists X(armbruch(X) \wedge beinbruch(X))$ (unverändert)

Beobachtete Ausgaben Die erwarteten Ausgaben werden alle korrekt ausgegeben. Es sind keine falschen oder unerwarteten Ausgaben aufgetreten.

Randwertprüfung Die Eingaben, die konjunktiv verbunden werden, könnten falsch normalisiert werden. Z.B.:
$armbruch(fred) \wedge (armbruch(alfred) \vee beinbruch(fred))$
Es wurde aber richtig normalisiert und folgende Ausgabe ist raus gekommen:
$armbruch(fred) \wedge (armbruch(alfred) \vee beinbruch(fred))$
Die Eingabe wurde nicht normalisiert und unverändert ausgegeben.

19.4 Strukturtest

Für die zentralen Methoden *cqe*, *addToLog* und *addToPot_sec* innerhalb von *Application* soll überprüft werden, ob der Programmfluss tatsächlich der im Entwurf vorgesehenen Struktur folgt. Die durch die Aktivitäts- bzw. Sequenzdiagramme gegebenen Reihenfolgen der Ausführung bestimmter Methoden sollen vom Programm auch eingehalten werden.
Da es leider keine anerkannten Testmethoden hierfür gibt, soll hier manuell vorgegangen werden. Es soll ein Debugginglauf für die drei Anwendungsfälle erstellt werden, anhand dessen mittels Eingaben aus verschiedenen Äquivalenzklassen das Programmverhalten im Einzelschrittmodus zurückverfolgt wird. Der Fokus liegt hierbei auf der Klasse *Application*. Es soll nicht tiefer in die Methoden hineingesehen werden, sondern nur der Durchlauf der Methodenaufrufe innerhalb der Hauptklasse überprüft werden. Bei einer vollständig korrekten Eingabe wird auch ein vollständiger Durchlauf, wie er im Aktivitätsdiagramm bzw. Sequenzdiagramm vorgesehen ist, erwartet. Bei den zahlreich möglichen fehlerhaften Eingaben muss die Hauptroutine an den entsprechenden Stellen, an denen der Fehler erkannt wird, mit einer *Exception* aussteigen.

19.4.1 Debugginglauf für cqe

Die Hauptroutine sieht so aus:

```
01 queryParser.ReInit(new StringReader(query));
02 Formula f;
03 String answer;
04 try {
05     TreeNode t = new TreeNode(queryParser.ANF());
06     isValid(t, new LinkedList());
07     f = new Formula(t);
08     f = f.toSRNF();
09     f.rr(f.getSyntaxTree());
10     if (!f.rrStrong(f.getSyntaxTree()))
11         throw new Exception("Anfrage ist nicht stark safe range");
12     lock(userData.getUserName()+"Log");
13     lock(userData.getUserName()+"Potsec");
14     answer = (modificator.modify(sqe(f.toSQL()), censor.classify(
           userData.getLog(), f, userData.getPot_sec())));
15 }
16 catch (QueryParser.ParseException e) {
17     throw new Exception("Syntaxfehler");
18 }
19 catch (Exception e) {
20     unlock();
21     throw new Exception(e.getMessage());
22 }
```

```
23 if (!answer.equals("‘MUM"’)) {
24    if (answer.equals("false")) {
25        f.negate();
26        f = new Formula(f.demorgan(f.getSyntaxTree()));
27    }
28    LinkedList subTrees = normalizeForLog(f.getSyntaxTree());
29    for (int i=0; i<subTrees.size(); i++) {
30       TreeNode t = (TreeNode)subTrees.get(i);
31       userData.addToLog(new Formula(t));
32    }
33 }
34 minimizeLog();
35 unlock();
36 return answer;
```

Äquivalenzklassen

1. Eingaben, die die Routine auf natürlichem Wege durchlaufen, also keine *Exception* werfen. Dabei ist es egal, wie die Ausgabe des Modifikators lautet.
2. Eingaben, die syntaktische Fehler enthalten. Hier muss in Zeile 5 eine *ParseException* geworfen werden, also entsprechend in den dafür vorgesehenen *catch-Block* gesprungen werden.
3. Formeln, die die semantischen Nebenbedingungen verletzen. In Zeile 6 muss die Methode *isValid* eine *Exception* werfen, die dann im entsprechenden *catch-Block* gefangen wird.
4. Formeln, die nicht *safe range* sind. Diese scheitern an der Methode *rr* in Zeile 9.
5. Formeln, die *safe range* aber nicht *stark safe range* sind. Diese kommen nicht über Zeile 11 hinaus, wo eine entsprechende *Exception* geworfen wird.

Getestete Eingaben

1. a) $\exists X armbruch(X) \wedge beinbruch(X)$
   mit Verweigerungszensor und
   $(db \models (\exists X armbruch(X) \wedge beinbruch(X)))$ und $pot\_sec = \emptyset$.
   b) $armbruch(alfred)$
   mit Lügenzensor und
   $(db \models (armbruch(alfred)))$ und $pot\_sec = \{armbruch(alfred)\}$.
2. $armbruch(alfred) \wedge$!!! $beinbruch(alfred)$
3. a) $armbruch(alfred, hans)$
   Relation hat falsche Stelligkeit.

   b) $urlaub(alfred)$
   Relation existiert nicht.

4. $\exists X \neg armbruch(X)$

5. $\exists X \neg armbruch(X) \wedge (beinbruch(X) \wedge beinbruch(alfred))$

Erwartete Pfade

1. a) Die Eingabe durchläuft den Programmabschnitt bis Zeile 14 problemlos und überspringt somit die *catch-Blöcke*. Der *if-Block* in Zeile 23 wird betreten und die *for-Schleife* in Zeile 29 wird zwei Mal durchlaufen, da sich die Formel in zwei Teilformeln zerlegen lässt. Dann geht es weiter bis zur *return-Anweisung* in Zeile 36.

   b) Diese Eingabe durchläuft genau dieselben Stationen wie die erste Anfrage. Der einzige Unterschied tritt in Zeile 24 auf. Hier wird der *if-Zweig* betreten und die Formel negiert, weil als Antwort $false$ ausgegeben wird.

2. Die Eingabe löst in Zeile 5 eine *ParseException* aus, die vom entsprechenden *catch-Block* verarbeitet wird.

3. a) Es wird das Programm nur bis Zeile 6 normal ausgeführt. An dieser Stelle tritt dann eine *Exception* auf, die anzeigt, dass die angegebene Relation eine falsche Stelligkeit aufweist.

   b) Auch hier soll das Programm bis in Zeile 6 eine Ausnahme auslösen. Die Fehlermeldung hier muss lauten: „Relation existiert nicht“.

4. Das Programm läuft bis Zeile 9, wo die Methode *rr* eine Ausnahme auslöst. Diese wird im *catch-Block* in Zeile 19 gefangen und die Meldung lautet: „Gebundene Variable ist nicht range restricted“.

5. In Zeile 10 muss der *if-Zweig* betreten werden und somit eine *Exception* mit der entsprechenden Fehlermeldung geworfen werden.

Beobachtete Pfade

1. a) Es wurde genau der vorhergesagte Pfad durchlaufen.

   b) Auch hier keine Abweichungen.

2. Es trat bei dieser Eingabe nicht die erwartete *ParseException* auf. Stattdessen wurde eine *Exception* im *TokenManager* des Parsers ausgelöst, die aber nicht ohne Weiteres von unserem Programm gefangen werden kann. Der Fehler liegt vermutlich darin, dass ein *Token* verwendet wurde, das im Parser nicht vorgesehen ist, nämlich das „!“. An dieser Stelle tritt dann ein lexikalischer Fehler auf. Um diesen Fehler im Programm behandeln zu können, sind wahrscheinlich

Eingriffe in den Parserklassen notwendig, was allerdings zum einen sehr aufwändig und in der Kürze der Zeit nicht zu realisieren ist. Zum anderen raten die Entwickler von *JavaCC* dringend davon ab, an den vom Parser-Generator erzeugten Klassen Veränderungen vorzunehmen. Dieses Problem wird in die Überlegungen für die Weiterentwicklung einbezogen.

3. a) Dieser Test verlief wie erwartet.
   b) Dieser Test verlief erwartunsgsgemäß.
4. Keine Abweichungen beobachtet.
5. Ebenfalls erfolgreich.

19.4.2 Debugginglauf für addToLog

Die Hauptroutine für *addToLog*:

```
01 logParser.ReInit(new StringReader(expr));
02 try {
03    TreeNode tree = new TreeNode(logParser.ANF());
04    isValid(tree, new LinkedList());
05    Formula f = new Formula(tree);
06    f = f.toSRNF();
07    f.rr(f.getSyntaxTree());
08    if (!f.rrStrong(f.getSyntaxTree()))
09       throw new Exception("Anfrage ist nicht stark safe range");
10    if (userData.getCensor()=='r') {
11       if (!this.sqe(f.toSQL()))
12          throw new Exception("Formel nicht wahr in DB");
13    }
14    if (censor.isPreconditionForLog(
         userData.getLog(), f, userData.getPot_sec())) {
15       LinkedList subTrees = normalizeForLog(f.getSyntaxTree());
16       lock(userData.getUserName()+"Log");
17       lock(userData.getUserName()+"Potsec");
18       for (int i=0; i<subTrees.size(); i++) {
19          TreeNode t = (TreeNode)subTrees.get(i);
20          userData.addToLog(new Formula(t));
21       }
22       minimizeLog();
23       unlock();
24    }
25    else
26       throw new Exception("Wissen gefaehrlich");
27 }
28 catch (LogParser.ParseException e) {
```

```
29    throw new Exception("Syntaxfehler");
30 }
31 catch (Exception e) {
32    unlock();
33    throw new Exception(e.getMessage());
34 }
```

Äquivalenzklassen

1. Eingaben, die die Routine auf natürlichem Wege durchlaufen, also keine *Exception* werfen.
2. Eingaben, die gefährlich sind, also zusammen mit dem alten Wissen mindestens ein potentielles Geheimnis aufdecken würden.
3. Eingaben, die syntaktische Fehler enthalten. Hier muss in Zeile 3 eine *ParseException* geworfen werden, also entsprechend in den dafür vorgesehenen *catch-Block* gesprungen werden.
4. Formeln, die die semantischen Nebenbedingungen verletzen. In Zeile 4 muss die Methode *isValid* eine *Exception* werfen, die dann im entsprechenden *catch-Block* gefangen wird.
5. Formeln, die nicht *safe range* sind. Diese scheitern an der Methode *rr* in Zeile 7.
6. Formeln, die *safe range* aber nicht *stark safe range* sind. Diese kommen nicht über Zeile 9 hinaus, wo eine entsprechende *Exception* geworfen wird.

Getestete Eingaben

1. $\exists X armbruch(X) \wedge beinbruch(X)$
   mit Verweigerungszensor und
   $(db \models (\exists X armbruch(X) \wedge beinbruch(X)))$ und $pot\_sec = \emptyset$.
2. $armbruch(alfred)$
   mit Lügenzensor und
   $(db \models (armbruch(alfred)))$ und $pot\_sec = \{armbruch(alfred)\}$.
3. $armbruch(alfred) \wedge$!!! $beinbruch(alfred)$
4. a) $armbruch(alfred, hans)$
      Relation hat falsche Stelligkeit.
   b) $urlaub(alfred)$
      Relation existiert nicht.
5. $\exists X \neg armbruch(X)$
6. $\exists X \neg armbruch(X) \wedge (beinbruch(X) \wedge beinbruch(alfred))$

Erwartete Pfade

1. Die Eingabe durchläuft den Programmabschnitt problemlos und überspringt somit die *catch-Blöcke*. Die *for-Schleife* in Zeile 18 wird einmal durchlaufen, da sich die Formel nicht zerlegen lässt. Ansonsten wird die Methode auf natürlichem Wege verlassen.
2. Für diese Eingabe liefert *isPrecondition* in Zeile 14 $false$, weshalb in den *else-Zweig* in Zeile 26 gesprungen wird. Hier wird eine *Exception* geworfen.

Für die restlichen Fälle wird genau dasselbe Verhalten erwartet, das auch schon beim Test von *cqe* erwartet wurde.

Beobachtete Pfade Die erste Eingabe wurde wie erwartet behandelt. Bei der zweiten Eingabe fiel allerdings ein Fehler auf. Das Wissen wurde trotz seiner Gefährlichkeit aufgenommen. Der Fehler war in der Methode *isPreconditionForLog* der Klasse *LyingCensor* begründet. Das Wissen wurde hier fälschlicherweise als unbedenklich eingestuft. Das lag wiederum daran, dass im Falle einer leeren log-Menge die Überprüfung der Implikation übersprungen und automatisch *true* zurückgegeben wird. Die Bedingung muss aber sein, dass die Menge *pot_sec* leer sein muss, damit ohne den Theorembeweiser zu bemühen, *true* zurückgegeben werden kann.
Im dritten Fall trat wiederum der aus dem vorherigen Test bereits bekannte Fehler mit dem *TokenManager* auf.
Alle anderen Tests verliefen zufriedenstellend.

19.4.3 Debugginglauf für addToPot_sec

Dieser Test sollte praktisch analog zu dem von *addToLog* sein. Hier können sogar noch weniger Fehler auftreten, da eine Prüfung auf *rr* und *rrStrong* entfällt. Einige Schnelltests haben im Prinzip das gleiche Verhalten wie für *addToLog* gezeigt.

19.4.4 Fazit

Insgesamt war diese Art von Test sehr ergiebig. Es wurden nicht nur die dokumentierten Fehler gefunden, sondern viele andere kleinere Inkonsistenzen und falsche Aufrufe, die aber unmittelbar beim Testen behoben und deshalb nicht dokumentiert wurden. Der Fall der falschen Bedingung, die in der Klasse *LyingCensor* auftrat, hat beispielhaft die Strahlwirkung von Fehlern gezeigt. Fehler, die in diesem Test zu Tage traten, waren fast immer an anderen Stellen zu finden, als direkt am Ort ihres Auftretens.

Teil V
Drittes Release

20 Entwurf

20.1 Offene Anfragen

20.1.1 Einleitung

Bislang wurden im Programm die geschlossenen Anfragen unter Verwendung der Lügenmethode, der Ablehnungsmethode und der kombinierten Methode durchgeführt. Ab jetzt wollen wir offene Anfragen unter Verwendung der alternativen kombinierten Methode [BB06] ausführen.
Dazu sollen wir die Klassen *Application*, *CombinedCensor*, *ProverNine* und *Formula* so erweitern, dass eine offene Anfrage, als eine Formel mit freien Variablen aufgebaut, ausgewertet und zensiert werden kann.
Im Rahmen der alternativen kombinierten Methode wird eine offene Anfrage zuerst in eine vollständige äquivalente Sequenz von geschlossenen Formeln umgewandelt, d.h. die freien Variablen aus der Anfrage werden durch die Konstanten aus einer unendlichen aufzählbaren Domäne ersetzt. Damit diese Sequenz nicht unendlich wird, brauchen wir eine obere Schranke für die Konstantenaufzählung.
Dafür wird ein Vollständigkeitstest verwendet, der Folgendes besagt:

- Ab einem Wert *j+1* (j ist eine ganze Zahl, die einer Kombination von Konstanten in einer *einheitlichen* Aufzählung, d.h. einer Aufzählung, die unabhängig von der Anfrage ist, einen Wert zuordnet) wird die durch Ersetzung von freien Variablen durch die Konstanten erhaltene prädikatenlogische geschlossene Formel stets $false$ liefern.

Um diesen Vollständigkeitstest durchzuführen, wird zuerst die Formel
$Complete(\Phi(x_1, ..., x_n), j)$ [BB06] gebildet. Diese Formel liefert genau dann *true* zurück, wenn alle geschlossenen Formeln, die durch Ersetzung von freien Variablen durch *die Konstanten ab* $j+1$ erhalten werden, $false$ zurückliefern. Die Methode *complete* der Klasse *CombinedCensor* wird aufgerufen, um $Complete(\Phi(x_1, ..., x_n), j)$ zu bilden. Beim Aufruf bekommt diese Methode eine ganze Zahl j und die Anfrage $\Phi(x_1, ..., x_n)$ als Übergabeparameter und gibt eine geschlossene prädikatenlogische Formel der Art zurück:

$$(\forall x_1)...(\forall x_n)[(x_1 = c_1 \wedge ... \wedge x_n = c_1) \vee ... \vee (x_1 = c_i \wedge ... \wedge x_n = c_l) \vee \neg\Phi(x_1, ..., x_n)],$$

wobei

- $\Phi(x_1, ..., x_n)$ die vom Benutzer gestellte Anfrage ist,
- $x_1, ..., x_n$ die freien Variablen in der Anfrage $\Phi(x_1, ..., x_n)$ sind,
- $c_i, ..., c_l$ die Kombination von Konstanten ist, die der Zahl j in der Aufzählung entspricht.

Da diese Zahl j die obere Schranke repräsentiert, werden im Rumpf dieser Methode die Methode *getFreeVariables* der Klasse *Formula* und die Methode *IndexToCombination* der Klasse *CombinedCensor* wie folgt aufgerufen:

- Die Methode *getFreeVariables* wird einmalig aufgerufen, um die Liste von allen freien Variablen zu liefern.
- Die Methode *IndexToCombination* der Klasse *CombinedCensor*, die in der Abhängigkeit von einer Zahl (laut der einheitlichen Aufzählung, siehe die Tabellen 8 und 9) eine Liste der Kombinationen von Konstanten zurückliefert, wird so oft aufgerufen, bis die Schranke j erreicht ist.

Die Formel $Complete(\Phi(x_1, ..., x_n), j)$ ist ein Bestandteil der Antwortmenge und der Logmenge. Deswegen soll unser Benutzer nicht imstande sein, durch die Kenntnis von $Complete(\Phi(x_1, ..., x_n), j)$ und von der Aufzählung der Domäne ein potentielles Geheimnis abzuleiten. Dafür wird die Zahl j so angepasst, dass es für den Benutzer unmöglich wird, irgendwelche potentiellen Geheimnisse abzuleiten. Für die Ermittlung dieser Zahl j werden die Methoden *getK, getM, getKOpt* und *getMOpt* der Klasse *CombinedCensor* verwendet.

Zuerst wird die Methode *getK* aufgerufen. Die Methode bekommt als Übergabeparameter die Anfrage $\Phi(x_1, ..., x_n)$ und gibt eine Zahl k zurück. Diese Zahl k ist der minimale Wert, bei welchem $Complete(\Phi(x_1, ..., x_n), k)$ in Bezug auf unsere Datenbank *true* liefert.

Als Nächstes wird die Methode *getM* aufgerufen. Die Methode bekommt als Übergabeparameter den Wert k (von der Methode *getK* übergeben) und die Anfrage $\Phi(x_1, ..., x_n)$ und muss eine Zahl m zurückliefern. Diese Zahl m ist der minimale Wert, bei welchem $Complete(\Phi(x_1, ..., x_n), m)$ zusammen mit dem Benutzerwissen *log* keine potentiellen Geheimnisse impliziert. Den Wert m erhält man dadurch, dass man intern für alle potentiellen Geheimnisse eine Schleife durchläuft. In dieser Schleife wird die Methode *isImplied* der Klasse *ProverNine* aufgerufen, die als Übergabeparameter das Benutzerwissen *log*, die Formel $Complete(\Phi(x_1, ..., x_n), k)$ und ein potentielles Geheimnis aus *pot_sec* bekommt. Die Methode *isImplied* wird solange aufgerufen, bis sie $false$ liefert.

Nachher wird die Methode *getKOpt* aufgerufen. Die Methode bekommt als Übergabeparameter den Wert m von der Methode *getM* übergeben und die Anfrage $\Phi(x_1, ..., x_n)$ und muss eine Zahl k^* zurückliefern. Diese Zahl k^* ist der maximale Wert, bei welchem die Formeln $\Phi(c_{k^*}), Complete(\Phi(x_1, ..., x_n), m)$ und das Benutzerwissen *log* keine potentiellen Geheimnisse implizieren. Den Wert k^* erhält man dadurch, dass man für $k^* \leq m$ für alle potentiellen Geheimnisse eine Schleife durchläuft. In dieser Schleife wird die Methode *isImplied* der Klasse *ProverNine* aufgerufen, die als Übergabeparameter das Benutzerwissen *log*, die Formel $Complete(\Phi(x_1, ..., x_n), m)$, $\Phi(c_j)$ und ein potentielles Geheimnis aus *pot_sec* bekommt. Die Methode *isImplied* wird solange aufgerufen, bis sie $false$ liefert. Sobald die Methode *isImplied* $false$ liefert, wird das erhaltene k^* mit dem vorherigen k^* verglichen. Falls das aktuelle k^* größer

als das vorherige ist, wird dem k^* der neue Wert zugewiesen. Am Ende bekommt man als Ergebnis ein maximales k^*.

Anschließend wird die Methode *getMOpt* aufgerufen. Die Methode bekommt als Übergabeparameter den Wert k^* (von der Methode *getKOpt* übergeben) und die Anfrage $\Phi(x_1, ..., x_n)$ und muss eine Zahl m^* zurückliefern. Diese Zahl m^* ist der minimale Wert, bei welchem $\Phi(c_{k^*}), Complete(\Phi(x_1, ..., x_n), m^*)$ und das Benutzerwissen *log* keine potentiellen Geheimnisse implizieren. Den Wert m^* erhält man dadurch, dass man für $m^* \geq k^*$ für alle potentiellen Geheimnisse eine Schleife durchläuft. In dieser Schleife wird die Methode *isImplied* der Klasse *ProverNine* aufgerufen, die als Übergabeparameter das Benutzerwissen *log*, die Formel $Complete(\Phi(x_1, ..., x_n), m^*)$, $\Phi(c_{k^*})$ und ein potentielles Geheimnis aus *pot_sec* bekommt. Die Methode *isImplied* wird solange aufgerufen, bis sie $false$ liefert. Sobald die Methode *isImplied* $false$ liefert, wird das erhaltene m^* mit dem vorherigen m^* verglichen. Falls das aktuelle m^* kleiner als das vorherige ist, wird dem m^* der neue Wert zugewiesen. Am Ende bekommt man als Ergebnis ein minimales m^*.

Da die ermittelten $Complete(\Phi(x_1, ..., x_n), m^*)$ und $\Phi(c_{k^*})$ keine potentiellen Geheimnisse ableiten lassen, werden diese Zwischenergebnisse unmittelbar dem Benutzer mitgeteilt und dem Benutzerwissen hinzugefügt. Die Methode *getStorage* aus der Klasse *CombinedCensor* wird diese zwei Ergebnisse ausgeben. Die Methode *getStorage* bekommt als Übergabeparameter m^*, k^* (von den Methoden *getKOpt* und *getMOpt* übergeben) und die Anfrage $\Phi(x_1, ..., x_n)$ übergeben und gibt die Formel $\Phi(c_{k^*})$ und die Formel $Complete(\Phi(x_1, ..., x_n), m^*)$ zurück.

Nachdem wir m^* ermittelt haben, können wir unsere offene Anfrage in eine Sequenz von geschlossenen Formeln umwandeln. Wir ersetzen die freien Variablen in der Anfrage durch Kombinationen von Konstanten (aus der Aufzählung) von 1 bis m^*, ausgenommen k^*. Jede geschlossene Formel aus dieser Sequenz wird dann durch den kombinierten Zensor (Iteration 2) geprüft und entsprechend klassifiziert.

Der Benutzer wird nun als Antwort auf die von ihm gestellte offene Anfrage eine Liste von geschlossenen Formeln bekommen. In dieser Liste gibt es zwei Arten von geschlossenen Formeln, hier am Beispiel einer Anfrage mit zwei freien Variablen:

- $\Phi(c_s, c_t)$: Die geschlossene Formel $\Phi(c_s, c_t)$ wird ausgegeben, falls:
  - die durch eine Belegung der freien Variablen (x_1, x_2) durch die Konstanten (c_s, c_t) erhaltene geschlossene Formel $\Phi(c_s, c_t)$ $true$ liefert und keine potentiellen Geheimnisse impliziert.
  - die durch eine Belegung der freien Variablen (x_1, x_2) durch Konstanten (c_s, c_t) erhaltene geschlossene Formel $\neg\Phi(c_s, c_t)$ $true$ liefert und ein potentielles Geheimnis aufdeckt. Die Negation der geschlossenen Formel $\neg\neg\Phi(c_s, c_t)$ dagegen deckt kein potentielles Geheimnis auf. (Folglich wird die korrekte Antwort $(\neg\Phi(c_s, c_t))$ gelogen und die Negation der korrekten Antwort $(\Phi(c_s, c_t))$ wird zu der Liste hinzugefügt.)
- $Complete(\Phi(x_1, x_2), m^*)$:

Die geschlossene Formel $Complete(\Phi(x_1, x_2), m^*)$ repräsentiert den Vollständigkeitstest im Rahmen der alternativen kombinierten Methode nach [BB06], der keine potentiellen Geheimnisse aufdeckt.

Für unser System sollen die folgenden getroffenen Annahmen erfüllt werden:

- Der Benutzer kennt die Aufzählung von Konstanten in unserer Domäne. Der Benutzer soll nicht im Stande sein , aus der Kenntnis der Aufzählung potentielle Geheimnisse abzuleiten. Dazu wird im Programm eine *einheitliche* Aufzählung (unabhängig von der Anfrage) benutzt.
- Die Anzahl der freien Variablen in der Anfrage soll kleiner oder gleich als *2* sein. Wenn die Anzahl der freien Variablen größer als *2* ist, soll der Benutzer eine Fehlermeldung erhalten.
- Statt einer unendlichen Domäne wird ein Wörterbuch (Oracle-Datentabelle) verwendet. Damit das Verhalten einer unendlichen Domäne simuliert werden kann, wird bei nicht ausreichender Anzahl von Konstanten eine Fehlermeldung ausgegeben. Dabei sind zwei Fälle zu unterscheiden:
  - Eine in der Anfrage auftretende Konstante ist nicht in unserem Wörterbuch vorhanden. Dann wird eine Fehlermeldung an den Benutzer ausgegeben, dass seine Anfrage nicht durchgeführt werden kann.
  - Bei der Durchführung des Vollständigkeitstests wurde die letzte Konstante in der Aufzählung (Wörterbuch) erreicht, d.h. der Vollständigkeitstest kann nicht weiter durchgeführt werden. Dann wird eine Fehlermeldung an den Benutzer ausgegeben, dass seine Anfrage nicht durchgeführt werden kann.

Nr.	x
1	c_1
2	c_2
3	c_3
4	c_4
5	c_5
6	c_6
7	c_7
8	c_8
9	c_9
10	c_{10}
11	c_{11}
12	c_{12}
.	.

Tabelle 8: Aufzählung für eine freie Variable x

Nr.	x	y
1	c_1	c_1
2	c_1	c_2
3	c_2	c_1
4	c_2	c_2
5	c_1	c_3
6	c_2	c_3
7	c_3	c_1
8	c_3	c_2
9	c_3	c_3
10	c_1	c_4
11	c_2	c_4
12	c_3	c_4
.	.	.

Tabelle 9: Aufzählung für zwei freie Variablen x und y

20.1.2 Aktivitätsdiagramm für offene Anfragen

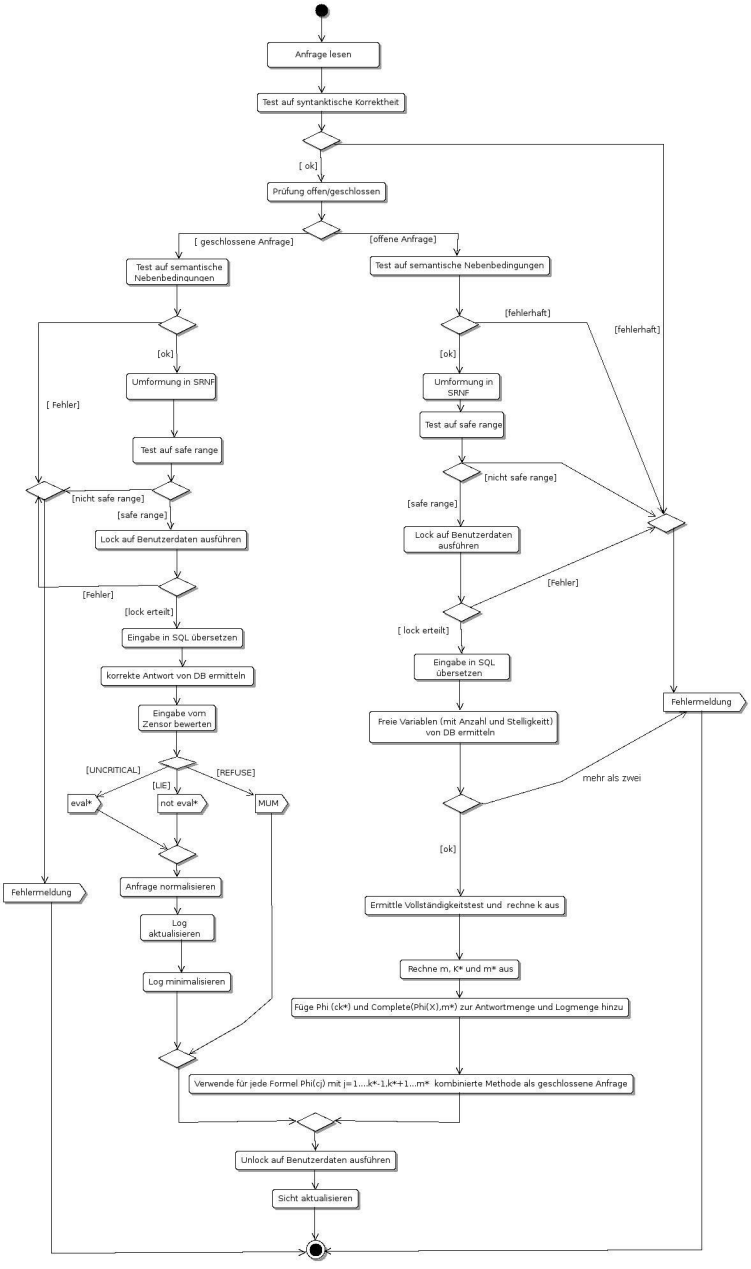

Abbildung 41: Ablauf der offenen Anfrageauswertung Gesamtansicht

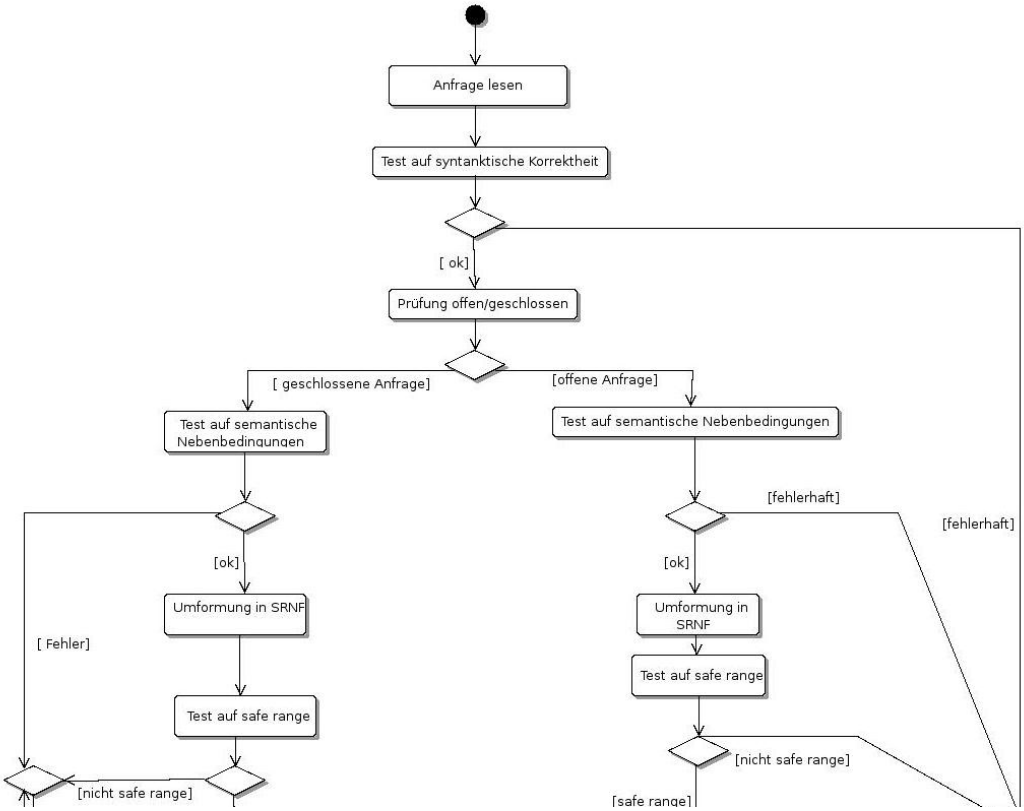

Abbildung 42: Ablauf der offenen Anfrageauswertung Teil 1

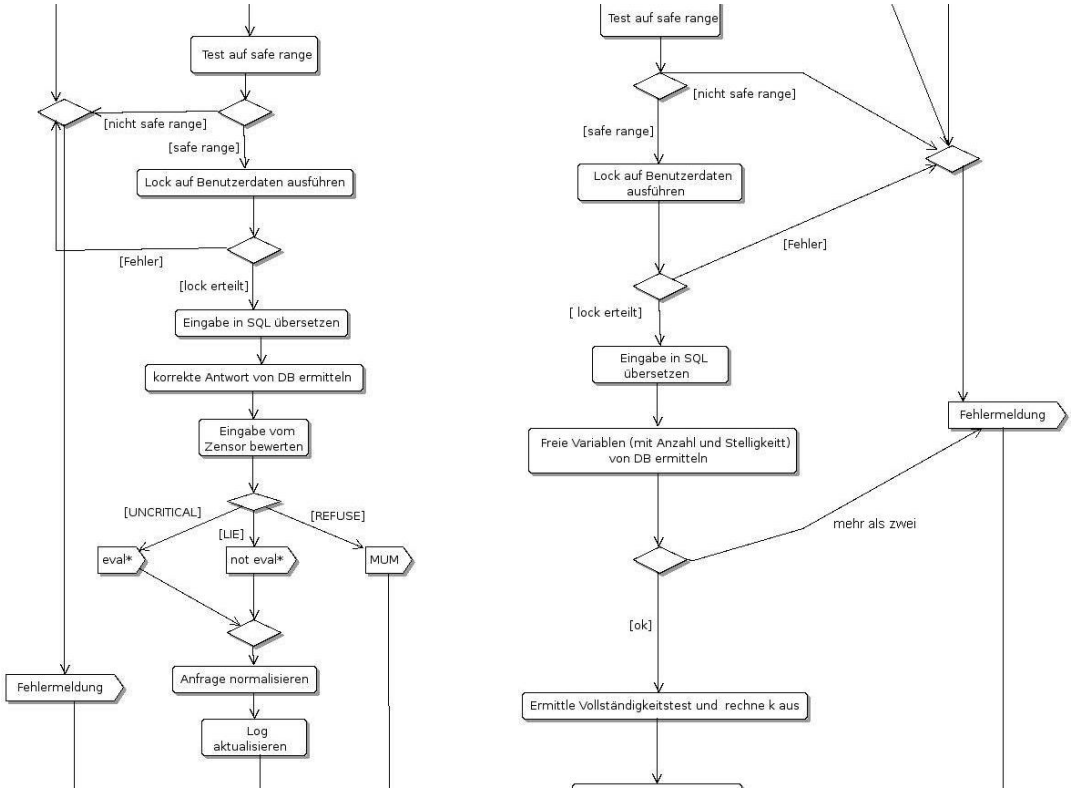

Abbildung 43: Ablauf der offenen Anfrageauswertung Teil 2

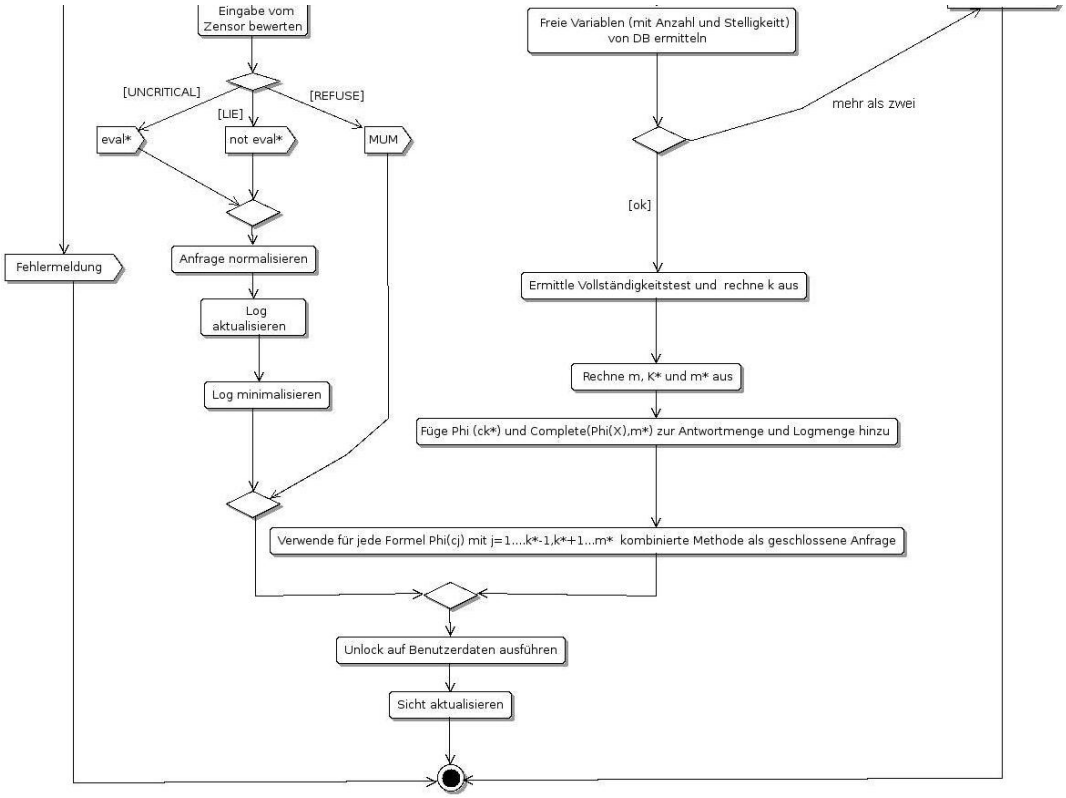

Abbildung 44: Ablauf der offenen Anfrageauswertung Teil 3

Die Abbildung 41 auf der Seite 262 zeigt den Ablauf der kontrollierten Anfrageauswertung für eine Anfrage. Nach der syntaktischen Prüfung wird das Programm in zwei Teile getrennt. Die linke Seite beschreibt den Ablauf für eine geschlossene Anfrage (siehe „zweites Release“ ActivityCQE). Die rechte Seite zeigt den Ablauf der kontrollierten Anfrageauswertung für eine offene Anfrage. Der Ablauf unterscheidet sich von dem im alten Release durch die zusätzliche Umformung, die im Rahmen der alternativen kombinierten Methode für die offene Anfrage durchgeführt wird.

Die alternative kombinierte Methode wird wie folgt beschrieben:

- Zuerst wird die Anzahl der freien Variablen in der vom Benutzer gestellten Anfrage ermittelt.
- Danach wird eine *SQL*-Tabelle als Antwort auf die vom Benutzer gestellte Anfrage zurückgeliefert.
- In der Abhängigkeit von der Anzahl der freien Variablen wird der Wert k ausgerechnet, bei welchem $Complete(\Phi(x_1, ..., x_n), k)$ in Bezug auf unsere Datenbank *true* liefert. (Es wurde so festgelegt, dass nur maximal 2 freie Variablen in unserem Programm erlaubt sind. Wenn es mehr als zwei freie Variablen gibt, wird direkt eine Fehlermeldung ausgeworfen.)
- Dann wird das minimale m ausgehend von k wie folgt bestimmt:
  Es wird überprüft, ob die Vereinigung von $Complete(\Phi(x), m)$ und log kein Element aus pot_sec impliziert.
- Danach wird ein k^* wie folgt bestimmt:
  Als das größte j, wobei $j \leq m$ ist, so dass log vereinigt mit $\Phi(c_j)$ und $Complete(\Phi(x), m)$ kein Element aus pot_sec impliziert.
- Und ein m^* wird wie folgt bestimmt:
  Als das kleinste j, wobei $k^* \leq j$ ist, so dass log vereinigt mit $\Phi(c_{k^*})$ und $Complete(\Phi(x), j)$ kein Element aus pot_sec impliziert.
- Die ausgerechnete Antwort $\Phi(c_{k^*})$ und $Complete(\Phi(x), m^*)$ werden schließlich zur Antwortmenge und Logmenge hinzugefügt.
- Zuletzt werden die $\Phi(c_j)$ von $j = 1...k^*-1, k^*+1...m$ als geschlossene Anfragen mit dem kombinierten Zensor betrachtet.

Danach werden entsprechend die Ergebnisse an die *GUI* weitergegeben und bei Bedarf wird das Benutzerwissen aktualisiert.

20.1.3 Sequenzdiagramm für offene Anfragen

Das Sequenzdiagramm aus Abbildung 45 zeigt einen möglichen Ablauf der kontrollierten Anfrageauswertung mit der alternativen kombinierten Methode im Detail. Hierbei wird sichtbar, welche Methoden von welchen Klassen aus aufgerufen werden und welche Informationen dabei fließen. Das Sequenzdiagramm zeigt genau einen möglichen Ablauf des Aktivitätsdiagramms. Hierbei gelten folgende Voraussetzungen:

- Die Anfrage ist syntaktisch korrekt und erfüllt alle semantischen Nebenbedingungen.
- Die Anfrage kann in *SRNF* überführt werden und besteht den Test auf safe rangeness.
- Der Exklusivzugriff auf die Tabellen wird erteilt.
- Es gibt zwei potentielle Geheimnisse.
- Die Anfrage ist gefährlich.

Wenn diese Voraussetzungen gelten, ergibt sich genau der dargestellte Ablauf.

Die Benutzeroberfläche ruft die Methode *cqe* im Objekt *Application* auf und übergibt ihr den vom Benutzer eingegeben String. Innerhalb der Methode *cqe* wird als nächstes unter Übergabe des eingegebenen Strings der *QueryParser*, ein Parser für die Eingabe, initialisiert. Durch den Aufruf der Methode *ANF* (die das Startsymbol ist, siehe Kapitel 12.1.3) wird der Parser gestartet. Da die Eingabe eine gültige Anfrage war, wird ein Objekt vom Typ *SimpleNode* zurückgegeben, welches die Wurzel des entsprechenden Syntaxbaums als Java-Datenstruktur darstellt.

Nun wird ein Objekt vom Typ *TreeNode* erzeugt. Der Konstruktor erhält als Parameter die vom Parser erzeugte SimpleNode Datenstruktur, aus welcher ein neuer Syntaxbaum, deren Knoten vom Typ *TreeNode* sind, erzeugt wird. Der Syntaxbaum wird nun mittels der Methode *isValid* auf seine Gültigkeit bezüglich der semantischen Nebenbedingungen geprüft. Nach Voraussetzung wird hier *true* zurückgegeben.

Anschließend wird die Anfrage in *SRNF* überführt und mittels der Methode *rr* auf safe rangeness überprüft. Danach wird mittels der Methode *lock* der benötigte Exklusivzugriff auf die entsprechenden Tabellen angefordert.

Dann wird die Methode *getFreeVariables* von der Klasse *Formula* aufgerufen. Rückgabe ist eine LinkedList, in der die freien Variablen der offenen Anfrage sind. Hier wird angenommen, dass die Anfrage eine offene Anfrage ist. Anhand der Rückgabe von *getFreeVariables* kann man wissen, ob die Anfrage eine offene Anfrage ist. Die Methode *cqeOpen* wird benutzt.

Die folgenden Schritte werden innerhalb der Methode *cqeOpen* ausgeführt. Die Methode *getResultSet* der Klasse *Formula* wird aufgerufen. Die Methode liefert das Ergebnis der offenen Anfrage zurück. Danach werden die Methoden *getK, getM, getKOpt,*

getMOpt hintereinander aufgerufen, um eine obere Schranke für die Konstantenaufzählung festzulegen.

Dann wird die Methode *getStorage* aufgerufen. Drei Parameter (zwei Integer-Werte *kOpt* und *mOpt* und eine Formel $\Phi(x)$) muss man übergeben. Die freie Variable in der Formel soll durch die Konstante *C_kOpt* ersetzt werden und in eine LinkedList, die die Rückgabe der Methode ist, gespeichert werden. Außerdem muss die Formel $Complete(\Phi(x), mOpt)$ auch in der Rückgabeliste gespeichert werden. *cqeOpen* übernimmt die Rückgabe von *getStorage* und speichert die beiden Elemente in der LinkedList im Benutzer-Log und der Antwortmenge der Anfrage.

Danach wird die Methode *getClosedQuery* aufgerufen. Die Parameterliste der Methode enthält zwei Integer-Werte *kOpt* (k^*) und *mOpt* (m^*) und eine Formel f (Anfrage $\Phi(x_1, ..., x_n)$). Die Methode übernimmt eine offene Anfrage und eine Menge von Konstanten, die von $c_1, ..., c_{k^*}, ..., c_{m^*}$ durchnummeriert sind, und die Methode ersetzt die freien Variablen in der Anfrage durch die entsprechenden Konstanten, also zum Beispiel, es gibt eine offene Anfrage *armbruch(X)*. Die Methode *getClosedQuery* ersetzt die Variable X durch $c_1, ..., c_{k^*-1}, c_{k^*+1}, ..., c_{m^*}$. Danach wird eine Menge von geschlossenen Anfragen erzeugt, also

$armbruch(c_1), ..., armbruch(c_{k^*-1}), armbruch(c_{k^*+1}), ..., armbruch(c_{m^*})$.

Dann speichert sie die geschlossenen Anfragen in einer LinkedList. Die Rückgabe der Methode ist die LinkedList.

Dann wird jedes Element der LinkedList, die von der Methode *getClosedQuery* zurückgeliefert wird, durch die Methode *cqe_Closed* behandelt.

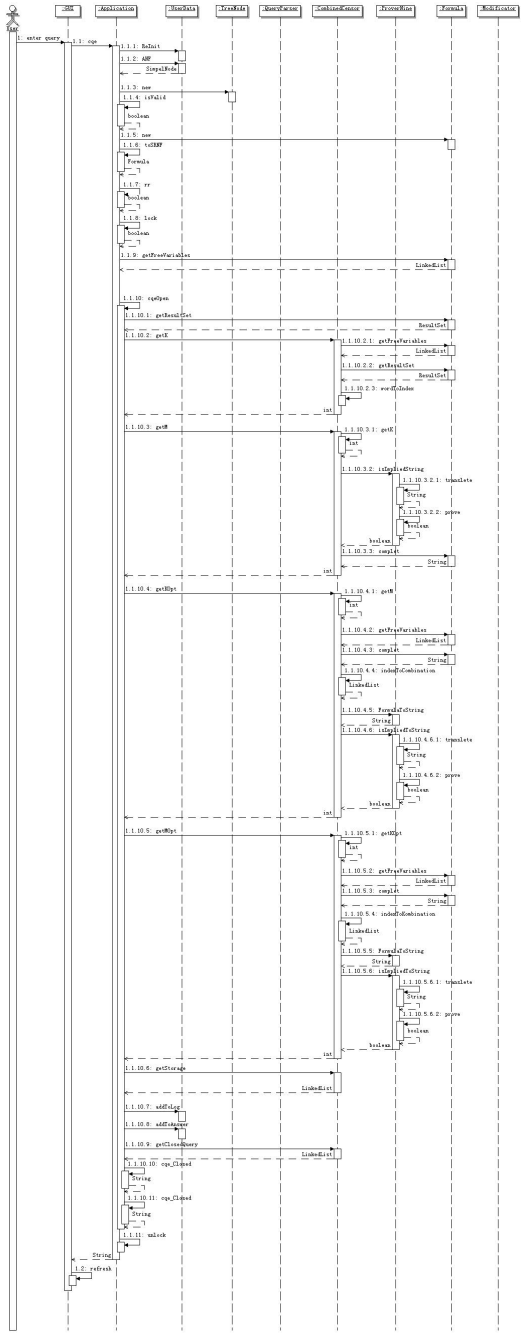

Abbildung 45: Sequenzdiagramm der kontrollierten Anfrageauswertung in der dritten Iteration Gesamtansicht

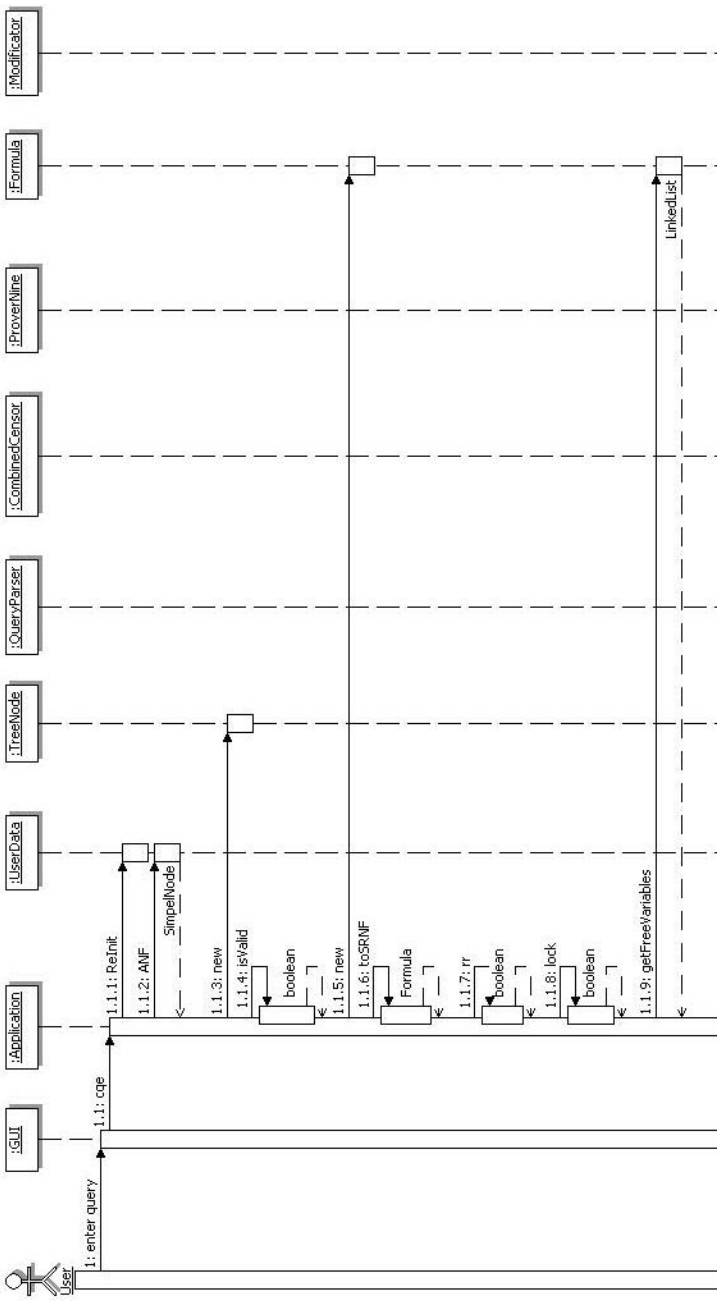

Abbildung 46: Sequenzdiagramm der kontrollierten Anfrageauswertung in der dritten Iteration Teil 1

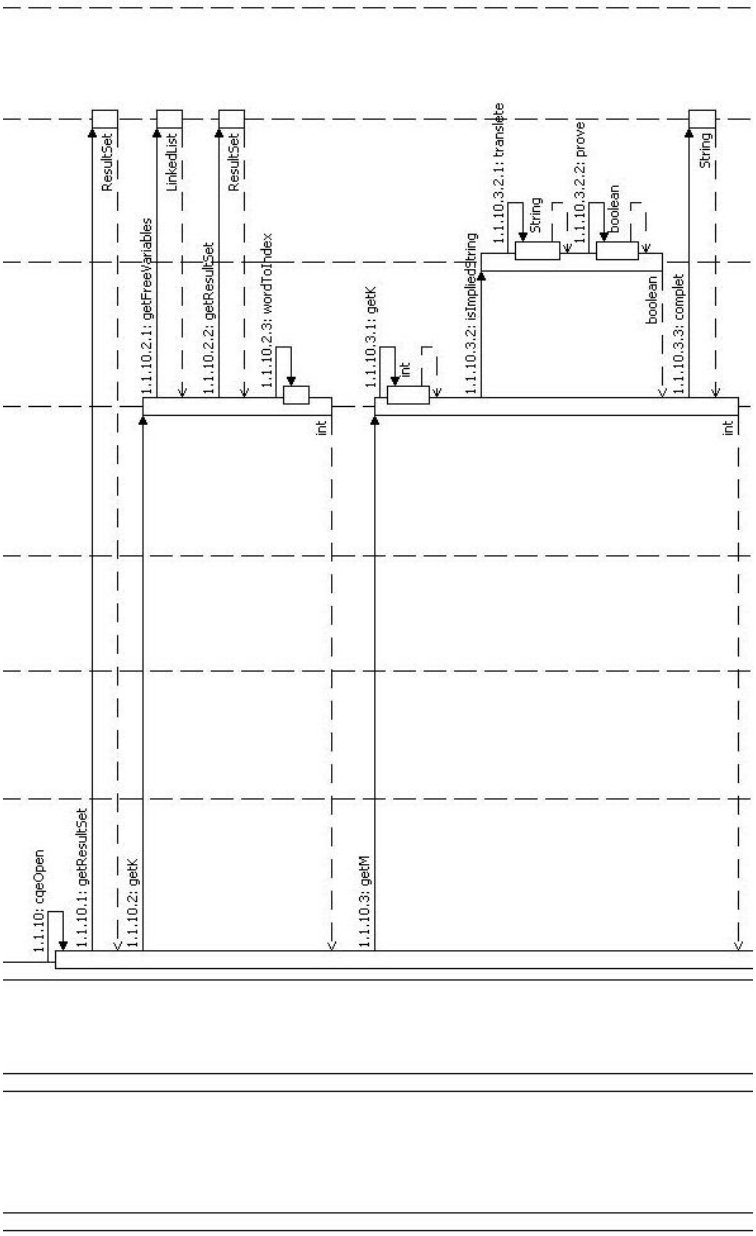

Abbildung 47: Sequenzdiagramm der kontrollierten Anfrageauswertung in der dritten Iteration Teil 2

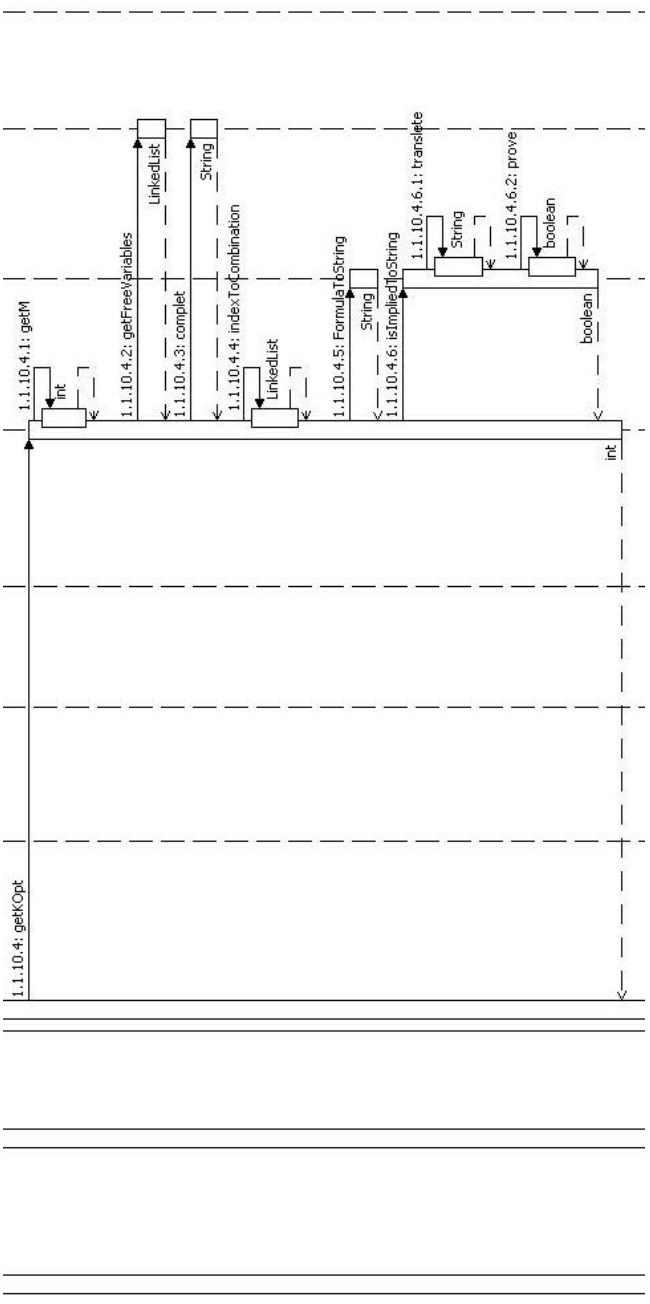

Abbildung 48: Sequenzdiagramm der kontrollierten Anfrageauswertung in der dritten Iteration Teil 3

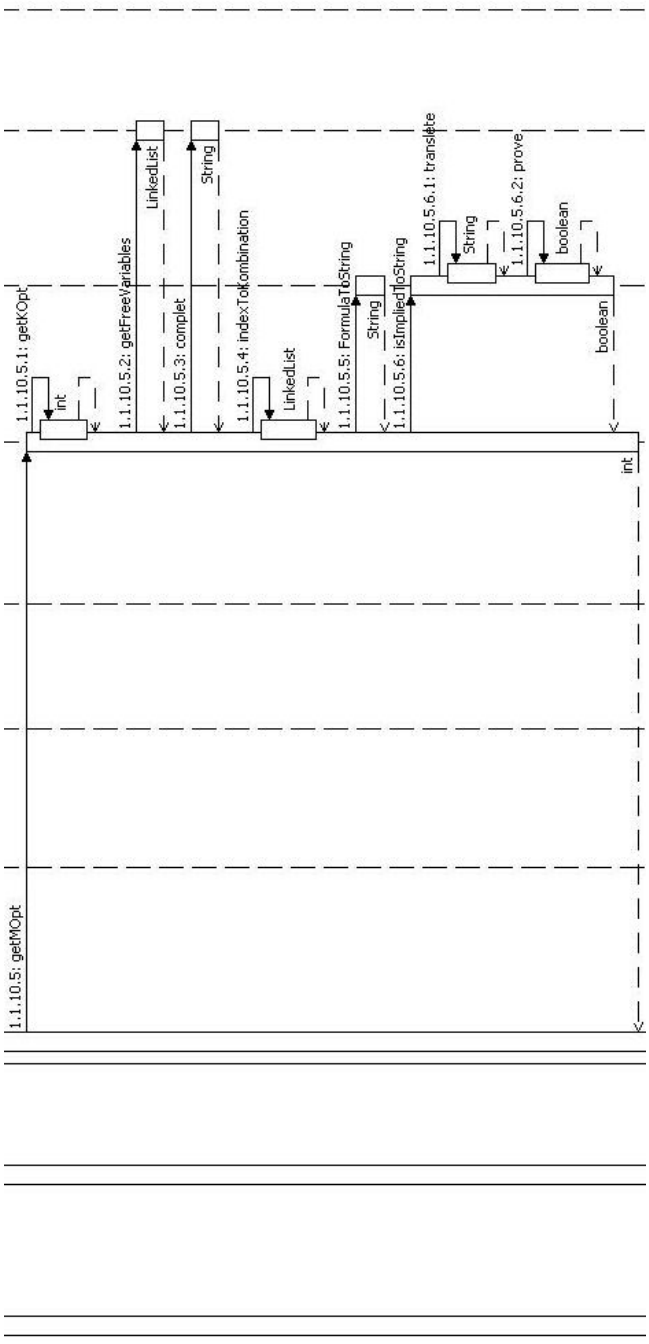

Abbildung 49: Sequenzdiagramm der kontrollierten Anfrageauswertung in der dritten Iteration Teil 4

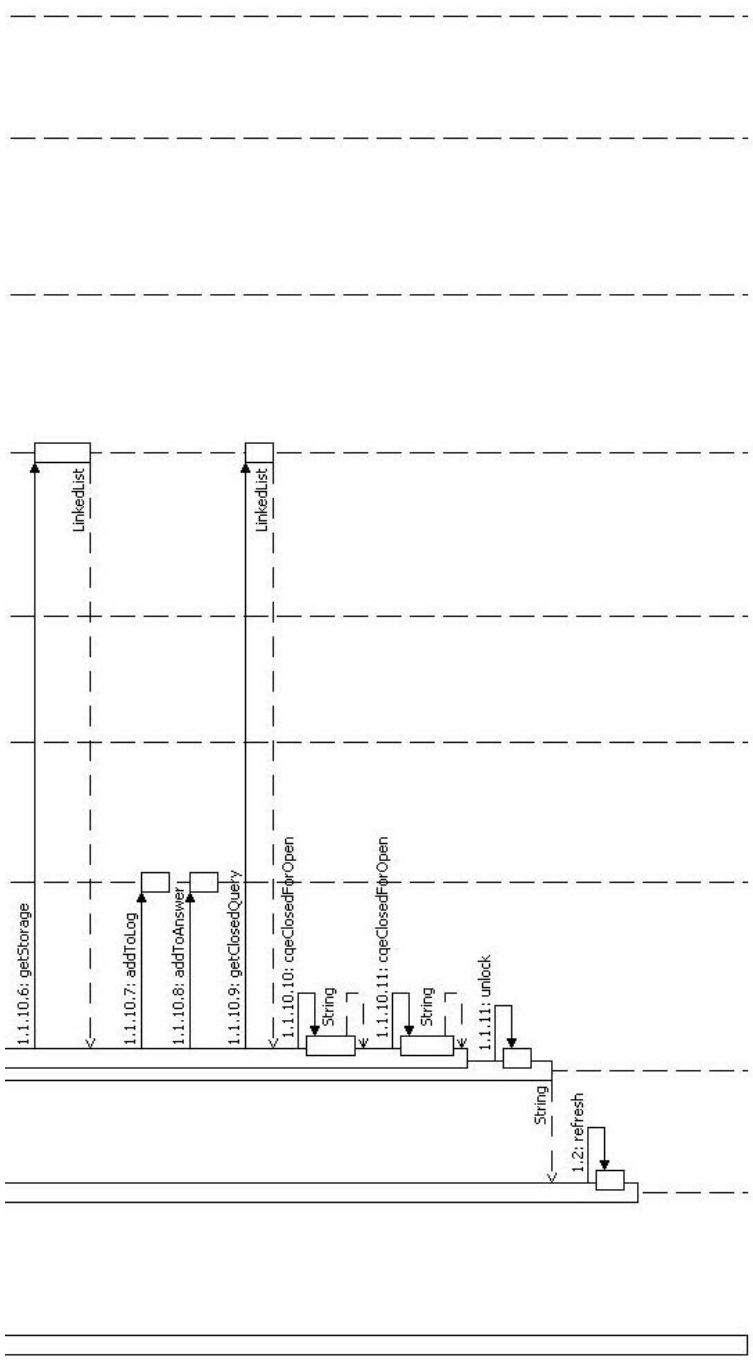

Abbildung 50: Sequenzdiagramm der kontrollierten Anfrageauswertung in der dritten Iteration Teil 5

20.2 Optimierung

20.2.1 Problemstellung

Der Einsatz des Theorembeweisers zur Inferenzkontrolle erfordert viel Rechenzeit innerhalb des Programms. Daher ist es sinnvoll sich Konstellationen des aktuellen Benutzerwissens, der Anfragen und der potentiellen Geheimnisse zu überlegen, bei denen eine Inferenzkontrolle auch ohne Theorembeweiser erfolgen kann bzw. gar nicht notwendig ist. Dazu sind geeignete Strukturen zu definieren, mit deren Hilfe solche Fälle erkannt und alternative Vorgehensweisen eingeschlagen werden können.
Eine generische Datenstruktur eines *Optimierers* muss zumindest die folgenden Informationen zur Verfügung haben:

- Das aktuelle Benutzerwissen *log*
- Die Sicherheitspolitik *pot_sec*
- Die Anfrage *query*

Ein Optimierer kann vor jeder Anfrageauswertung, anhand der Eigenschaften der drei o.g. Elemente, u.U. einen passenden Ausführungsplan wählen, der effizienter als eine Inferenzkontrolle mit Hilfe des Theorembeweisers ist.

20.2.2 Ein Spezialfall

Eine solche spezielle Eigenschaft kann sein, dass das Benutzerwissen, die potentiellen Geheimnisse und auch alle Anfragen lediglich aus Grundliteralen bestehen. In diesem Fall muss nur noch entschieden werden, ob *query* mit einem Element aus *pot_sec* äquivalent ist. Dies kann über einen einfachen syntaktischen Vergleich erfolgen, da äquivalente Grundliterale auch immer syntaktisch gleich sind. Hierfür kann die in *Formula* bereits existierende Methode *equals* verwendet werden, die genau dann *true* liefert, wenn die übergebene Formel die gleiche String-Repräsentation besitzt wie die aufrufende Formel. Mit dieser Vorgehensweise kann sowohl die für den Verweigerungs- und den kombinierten Zensor relevante Frage, ob

$$\exists \Psi \in pot\_sec : query \models \Psi \quad (1)$$

und die für den Lügenzensor relevante Frage, ob

$$query \models pot\_sec\_disj \quad (2)$$

gilt, beantwortet werden, indem man den Wahrheitswert von

$$\exists \Psi \in pot\_sec : query \equiv \Psi \quad (3)$$

überprüft. In diesem Falle reicht also eine Implementierung für alle drei Zensoren. Da die Äquivalenz der Aussagen (2) und (3) nicht offensichtlich ist, soll diese in folgendem Satz festgehalten werden.

Satz Seien $\chi_1, \chi_2, \ldots, \chi_n$ verschiedene Grundliterale und $\Psi_1, \Psi_2, \ldots, \Psi_m$ verschiedene Grundliterale. Dann gilt:

$$\chi_1 \wedge \chi_2 \wedge \ldots \wedge \chi_n \models_{DB} \Psi_1 \vee \Psi_2 \vee \ldots \vee \Psi_m \quad \text{gdw. } \exists i, j \quad \chi_i \equiv \Psi_j$$

Beweis:
"$\Leftarrow$": trivial
"$\Rightarrow$": durch Kontraposition
Annahme:

$$\forall i, j \quad \chi_i \not\equiv \Psi_j \quad (1)$$

Betrachte folgende *db* (Interpretation):

$$db \models \chi_i \quad i = 1, \ldots n \quad (2)$$
$$db \not\models \Psi_j \quad j = 1, \ldots m \quad (3)$$

Diese ist durch die Annahmen wohldefiniert, da man jedes Grundliteral unabhängig von allen anderen definieren kann. Bei gleichen Atomen nehmen die Grundliterale komplementäre Wahrheitswerte in *db* an.

- Aus (2) folgt: $db \models \chi_1 \wedge \ldots \wedge \chi_n$
- Aus (3) folgt: $db \not\models \Psi_1 \vee \ldots \vee \Psi_m$

Dann gilt aber $\chi_1 \wedge \ldots \wedge \chi_n \not\models_{DB} \Psi_1 \vee \ldots \vee \Psi_m$, also Gegenbeispiel gefunden.□

20.2.3 Umsetzung im Programm

Der o.g. Spezialfall ist nur einer von vielen denkbaren. Im Programm soll jedoch zunächst nur dieser eine Fall abgedeckt werden, wobei die Strukturen so gewählt werden sollen, dass die Erkennung beliebiger weiterer Spezialfälle problemlos integriert werden kann.
Es wird daher eine neue Klasse *Optimizer* konstruiert, welche mit o.g. Eingabe durch die Methode *optimize* ggf. ein geeignetes Vorgehen zur Optimierung der Inferenzkontrolle vorschlägt.

Änderung in der Klasse *Formula* Für den uns im Moment interessierenden Fall ist es wichtig, über jede Formel der Klasse *Formula* erfahren zu können, ob sie ein Grundliteral ist. Es scheint sinnvoll zu sein, dies mittels eines Attributes *groundatom* festzuhalten. Der aktuelle Wert soll über die Methode *isGroundatom* abgefragt werden können. Bei der Erzeugung einer Formel (d.h. im Konstruktor) kann der Wert direkt

gesetzt werden, indem überprüft wird, ob der Wurzelknoten genau ein Kind und dieses Kind wiederum ein oder zwei Kinder hat. Bei einem Kind handelt es sich um ein positives, bei zwei Kindern um ein negatives Grundfakt. Bei Anwendung der Methoden *toSRNF, demorgan* oder *negate* bleiben Grundliterale erhalten. Es kann also nicht passieren, dass ein Grundliteral nach Anwendung einer dieser drei Methoden kein Grundliteral mehr ist.

Änderung in der Klasse *UserData* Die bereits oben erwähnten Informationen über das Benutzerwissen *log* und die Sicherheitspolitik *pot_sec* werden im Programm durch die Klasse *UserData* verwaltet. Hier bietet es sich daher an, jeweils für das Benutzerwissen und die Menge der potentiellen Geheimnisse zwei Attribute zu definieren (*groundatomLog* und *groundatomPot_sec*), die festhalten, ob die Mengen nur aus Grundliteralen bestehen. Dazu müssen die Methoden *addToLog*, *addToPot_sec*, *getLog*, *getPot_sec*, *deleteFromLog* und *deleteFromPot_sec* erweitert werden.
Wird eine Formel mittels *addToLog* oder *addToPot_sec* zur jeweiligen Menge hinzugefügt und besteht die Menge aus Grundliteralen, so muss die neue Formel entsprechend überprüft werden, ob sie ein Grundliteral darstellt, damit ggf. das entsprechende Attribut *groundatomLog* bzw. *groundatomPot_sec* angepasst werden kann.
Analog gilt dies auch für die Methoden *getLog* und *getPot_sec*, die das Benutzerwissen bzw. die potentiellen Geheimnisse aus der Datenbank in das Programm einlesen. Wenn eine Formel aus einer der Mengen durch Aufruf der Methoden *deleteFromLog* oder *deleteFromPot_sec* entfernt wird und die Menge vorher nicht ausschließlich aus Grundliteralen bestand, so muss überprüft werden ob die zu löschende Formel kein Grundliteral ist und die Menge nun ggf. nur noch aus Grundliteralen besteht, damit das zugehörige Attribut *groundatomLog* bzw. *groundatomPot_sec* aktualisiert werden kann.

Änderung in der Klasse *Censor* In der Klasse *Censor* kann durch die Methode

```
isImpliedForGroundAtoms(Formula query, LinkedList pot_sec)
```

die syntaktische Überprüfung gemacht werden, ob eine Anfrage *query* des Benutzers ein potentielles Geheimnis aus der Menge *pot_sec* impliziert. Dazu muss lediglich die String-Repräsentation der Anfrage mit den String-Repräsentationen der potentiellen Geheimnisse verglichen werden. Die angepasste Klassifizierungsmethode *classifyForGroundAtoms* benutzt die Methode *isImpliedForGroundAtoms* zur Inferenzkontrolle.

Änderung in der Klasse *Application* Bevor die eigentliche kontrollierte Anfrageauswertung gemacht wird, soll zunächst der Optimierer befragt werden, ob dieser anhand des übergebenen Benutzerwissens, der potentiellen Geheimnisse und der Benutzeranfrage eine effizienzsteigernde Optimierung vorschlägt. Dieser liefert in unserem Spezialfall entweder den Rückgabewert *GROUNDATOMS*, d.h. die Anfrage und die Menge der potentiellen Geheimnisse bestehen aus Grundliteralen und somit sollte die effiziente Methode *classifyForGroundAtoms* des Zensors benutzt werden. Im anderen Fall

liefert der Optimierer den Rückgabewert *NOOPT*, es wird somit keine Optimierung vorgeschlagen und die kontrollierte Anfrageauswertung muss wie gewohnt ausgeführt werden.

20.3 DB-Implikation

Die Informationen dieses Abschnitts stammen von [BB06].

20.3.1 Einleitung

Unser bislang benutzter Theorembeweiser (Prover9) kann nur die Gültigkeit der allgemeinen Implikation überprüfen. Unser Ziel ist jetzt, dieses Überprüfen innerhalb des Programms auf die sogenannte DB-Implikation zu erweitern. Dazu werden wir erst die Definitionen und Theoreme über DB-Implikation betrachten und dann daraus ableiten, wie man es in das Programm einbauen kann.

20.3.2 Ansatz

Definition 1 Eine Interpretation I ist eine DB-Interpretation g.d.w. folgende Bedingungen gelten:

1. dom(I)= dom, (dom(I) ist gleich mit dem Universum)
2. p^I ist endlich in $\mathfrak{L}$ für alle Prädikatensymbole p,
3. $c_i^I = c_i$, für alle Konstanten c_i in $\mathfrak{L}$

Wobei

- $dom = \{c_1, c_2, ..., c_i, ...\}$ eine abzählbare unendliche Menge,
- p Prädikatensymbol in $\mathfrak{L}$,
- $\mathfrak{L}$ eine funktionsfreie Sprache, in der die Menge der Konstanten eine endliche Untermenge von dom ist,
- dom(I) die Domain von einer Interpretation I,
- p^I die Interpretation von p in I,
- c^I die Interpretation von c in I

sind.

Definition 2 Für jede Menge von Aussagen Γ und jede Aussage φ gilt:

- $\Gamma \models_{DB} \varphi$ g.d.w. für alle DB-Interpretationen I: $I \models \Gamma$ impliziert $I \models \varphi$;
- $\Gamma \models_{gen} \varphi$ g.d.w. für alle allgemeinen (generellen) Interpretationen I : $I \models \Gamma$ impliziert $I \models \varphi$.

Definition 3 Die aktive Domain von einer Interpretation I bezüglich einer Formel φ ist definiert durch

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

Definition 4 $Out_\varphi(x)$ ist wie folgt definiert:

$(\forall y_1)...(\forall y_m)$
$\bigwedge_p \ \bigwedge_{0\leq k\leq arity(p)} \ \neg p(y_1, ..., y_k, x, y_{k+2}, ..., y_{arity(p)}) \wedge \ \bigwedge_{c\in const(\varphi)} \ \neg(x = c)$

Dabei ist m die maximale Stelligkeit eines Prädikatensymbols p.
p ist ein Prädikatensymbol in unserer Datenbank.
Es ist zu beachten, dass $(\exists x)Out_\varphi(x)$ genau dann als *true* interpretiert wird, wenn ein x existiert, das nicht aus $active_\varphi(I)$ ist.
Weiterhin bezeichnen wir die Aussage $(\exists x)Out_\varphi(x)$ mit *non-totality assumption.*

Beispiel Wir versuchen jetzt anhand eines einfachen Beispiels die Aussage $(\exists x)Out_\varphi(x)$ zu veranschaulichen.
Nehmen wir an, dass in unserer Datenbank nur zwei Prädikate *armbruch* und *krankheit* existieren, wobei *armbruch* ein einstelliges Prädikat und *krankheit* ein zweistelliges ist. Die *non-totality assumption* für eine Anfrage $\varphi = armbruch(alfred)$ lautet:

$(\exists x)$
$(\forall y_1)(\forall y_2)$
$[\neg armbruch(x) \wedge$
$\neg krankheit(y_1, x) \wedge \neg krankheit(x, y_2) \wedge \neg(x = alfred)]$

Definition 5 Für alle Formeln φ lässt sich UNA_φ (*unique name assumption for* φ) wie folgt definieren:

$$\bigwedge \{c_i \neq c_j |\ \{c_i, c_j\} \in const(\varphi)\ und\ i < j\}.$$

Theorem Für alle $\Psi \in \neg\mathcal{L}$ und alle $\varphi \in \mathcal{L}$ sind die folgenden Eigenschaften äquivalent:

1. $\Psi \models_{DB} \varphi$;
2. $\Psi\ \wedge (\exists x)Out_{\Psi,\varphi}(x) \wedge UNA_{\Psi,\varphi} \models_{gen} \varphi$;

Dabei gilt:

- $\mathcal{L}$ ist ein Fragment der Prädikatenlogik erster Stufe, das abgeschlossen unter $\vee$ ist.

- $\Psi \models_{gen} \Phi$ zwischen zwei *geschlossenen Formeln (Sentences)* ist äquivalent mit der Eigenschaft „$\neg\Psi \vee \Phi$ ist allgemeingültig“.
- $\neg\mathcal{L} = \{\neg\varphi | \varphi \in \mathcal{L}\}$

Anhand dieses Theorems werden wir versuchen, innerhalb des Programms die DB-Implikation einzubauen.
Wir werden die beiden Methoden *generateUNA*() und *generateOut*() in der Klasse *ProverNine* (zum Realisieren von $UNA_{\Psi,\varphi}$ und $Out_{\Psi,\varphi}(x)$) und die Methode *getConstants*() in der Klasse *Formula* (um eine Liste der Konstanten in der Formel auszugeben) verwenden.

21 Implementierung

21.1 DB-Implikation

Um die im Entwurfsdokument beschriebene DB-Implikation zu simulieren, haben wir zunächst zwei Methoden namens

$$LinkedList < String > \ getConstants(TreeNode)$$

und

$$LinkedList < String > \ getPredicates(TreeNode)$$

in der Klasse *Formula* implementiert, die dazu dienen, eine Liste der benötigten Konstanten bzw. Prädikate zur Erstellung von *non-totality assumption* und *unique name assumption* zu bekommen.

Die beiden oben genannten Annahmen werden anhand der Methoden

$$String\ generateOut(LinkedList,\ Formula,\ LinkedList)$$

(zur Erzeugung der *non-totality assumption*) und

$$String\ generateUNA(LinkedList,\ Formula,\ LinkedList)$$

(zur Erzeugung der *unique name assumption*) innerhalb der Klasse *ProverNine* erstellt, wobei den beiden Methoden als Eingabeparameter Benutzer-log und Benutzer-pot_sec als LinkedList und Benutzer-Anfrage vom Typ Formula übergegeben werden. Die Methoden erzeugen die entsprechenden Annahmen und liefern diese als String zurück.
Eine andere Methode, die dabei implementiert wurde, ist

$$String\ getVariablesPermutations(String\ name,\ String\ arity).$$

Diese Methode gehört auch zur Klasse *ProverNine* und erzeugt die verschiedenen Einsetzungen einer Variablen x an alle Positionen in einem Atom, wie sie innerhalb der *unique name assumption* gebraucht werden. Zum Beispiel für eine Relation namens R mit Stelligkeit 2 wird ausgegeben:

$$\neg R(x, y_2) \wedge \neg R(y_1, x),$$

und für eine dreistellige Relation namens P lautet die Ausgabe:

$$\neg P(x, y_2, y_3) \wedge \neg P(y_1, x, y_3) \wedge \neg P(y_1, y_2, x).$$

Die beiden Methoden *generateOut* und *generateUNA* werden innerhalb der Methode

$$boolean\ isImplied(LinkedList,\ LinkedList,\ Formula,\ LinkedList)$$

(die bereits aus vorigen Iterationen existiert) aufgerufen und die zurückgelieferten Annahmen werden dann im linken Teil der Eingabe vom Theorembeweiser eingefügt. D.h. der Theorembeweiser überprüft statt $\Psi \models_{gen} \varphi$ die erweiterte Form

$$\Psi \ \wedge non-totality\ assumption \wedge unique\ name\ assumption \models_{gen} \varphi$$

und so ist die DB-Implikation simuliert.

22 Test

22.1 Methode getClosedQueries

Diese Methode soll für die Eingaben k^*, m^* und eine offene Anfrage (mit höchstens 2 frei vorkommenden Variablen) eine Liste aller zugehörigen geschlossenen Anfragen liefern. Dies sind jeweils $m^* - 1$ viele, da $\Phi(c_{k^*})$ jeweils nicht mit in die Liste aufgenommen wird. Für diesen Test wird ein kleines Wörterbuch vorausgesetzt, das die Konstanten $c_1, \ldots, c_4$ enthält.
Die Konstanten werden auch in genau dieser Reihenfolge aufgezählt. Für zwei freie Variablen wird in folgender Reihenfolge ersetzt:

c_1	c_1
c_1	c_2
c_2	c_1
c_2	c_2
c_1	c_3
c_2	c_3
c_3	c_1
c_3	c_2
c_3	c_3
c_1	c_4
c_2	c_4
c_3	c_4
c_4	c_1
c_4	c_2
c_4	c_3
c_4	c_4

Äquivalenzklassen

Gültige Eingaben Folgende Unterscheidungen zwischen gültigen Eingaben erscheinen sinnvoll:

1. Eingabe einer offenen Anfrage mit einer frei vorkommenden Variablen, wobei gilt: $1 \leq k^* \leq 4$ und $k^* = m^*$.
2. Eingabe einer offenen Anfrage mit einer frei vorkommenden Variablen, wobei gilt: $1 \leq k^* \leq 4$ und $1 \leq m^* \leq 4$ und $k^* < m^*$.
3. Eingabe einer offenen Anfrage mit zwei frei vorkommenden Variablen, wobei gilt: $1 \leq k^* \leq 16$ und $k^* = m^*$.
4. Eingabe einer offenen Anfrage mit zwei frei vorkommenden Variablen, wobei gilt: $1 \leq k^* \leq 16$ und $1 \leq m^* \leq 16$ und $k^* < m^*$.

Ungültige Eingaben Alle Eingaben, bei denen $k^* > m^*$, $k^* < 1$, $m^* < 1$, $k^* > AnzahlKombinationen$ oder $m^* > AnzahlKombinationen$ gilt, sind ungültig. Es werden innerhalb der Methode auch keine Absicherungsmaßnahmen getroffen, um einen solchen Fehlerfall zu erkennen und abzufangen. Die Methode verläßt sich auf die korrekte Arbeitsweise der Methoden *getKOpt* und *getMOpt*, welche einen solchen Fall ausschließt.

Getestete Eingaben Aus den jeweiligen Äquivalenzklassen wurden folgende Repräsentanten ausgewählt:

1. Anfrage: $armbruch(X)$, $k^* = m^* = 3$
2. Anfrage: $armbruch(X)$, $k^* = 2, m^* = 4$
3. Anfrage: $krankheit(X, Y)$, $k^* = m^* = 6$
4. Anfrage: $krankheit(X, Y)$, $k^* = 5, m^* = 8$

Setup Folgendes Setup wurde zur Durchführung der Tests aufgesetzt:

```
logParser.ReInit(new StringReader("armbruch(X);"));
Formula f = new Formula(new TreeNode(logParser.ANF()));
logParser.ReInit(new StringReader("krankheit(X,Y);"));
Formula g = new Formula(new TreeNode(logParser.ANF()));
```

Assertions Die Assertions wurden alle nach einem festen Schema aufgebaut, so z.B. die Assertion für die erste Eingabe:

```
Assert.assertEquals(ref, censor.getClosedQueries(3,3,f));
```

wobei *ref* eine *LinkedList* ist, die jeweils die erwarteten geschlossenen Anfragen enthielten. Alle anderen Tests wurden analog durchgeführt.

Erwartete Ausgaben Die folgenden Listen wurden für die jeweiligen Tests als Ausgaben erwartet:

1. $armbruch(c_1), armbruch(c_2)$
2. $armbruch(c_1), armbruch(c_3), armbruch(c_4)$
3. $krankheit(c_1, c_1), krankheit(c_1, c_2), krankheit(c_2, c_1),$
   $krankheit(c_2, c_2), krankheit(c_1, c_3)$
4. $krankheit(c_1, c_1), krankheit(c_1, c_2), krankheit(c_2, c_1),$
   $krankheit(c_2, c_2), krankheit(c_2, c_3), krankheit(c_3, c_1),$
   $krankheit(c_3, c_2)$

Beobachtete Ausgaben Für alle Tests deckten sich die beobachteten Ausgaben mit den erwarteten. So war z.B bei keinem Test $\Phi(c_{k^*})$ in der Ausgabe enthalten.

22.2 Methoden getK, getM, getKOpt und getMOpt

Im Folgenden werden die Methoden *getK*, *getM*, *getKOpt* und *getMOpt*, die aufeinander aufbauen, getestet. Das verwendetete Wörterbuch hat dabei folgendes Aussehen:

KONSTANTE	ID
frank	1
beinbruch	2
yu	3
armbruch	4
alfred	5
dortmund	6
muenchen	7
kurz	8
hamm	9
lang	10

Die Tabelle „Armbruch“:

NAME
yu
alfred

Die Tabelle „Krankheit“:

NAME	KRANKHEIT
frank	beinbruch
alfred	armbruch

22.2.1 Testmodul getK

Die Methode *getK* erhält eine Anfrage $\Phi(x_1, ..., x_n)$ (mit $n \in \{1, 2\}$) als Übergabeparameter und liefert eine Zahl k zurück. Dabei ist k der minimale Wert, bei welchem die Vollständigketitsformel $Complete(\Phi(x_1, ..., x_n), k)$ erfüllt ist.

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die gültigen Eingaben der Methode

```
int getK(Formula f)
```

bestehen aus Anfragen der Form $\Phi(x_1, ..., x_n)$ (mit $n \in \{1, 2\}$).
Hier sollten daher die Fälle $n = 1$ und $n = 2$ betrachtet werden.

Ungültige Eingaben Die Äquivalenzklassen für die ungültigen Eingaben bestehen aus Aufrufen der Methode *getK* mit Anfragen, die keine freien Variablen oder mehr als zwei freie Variablen besitzen. Eine Anfrage, die keine freien Variablen enthält, kann nach der internen Struktur des Programms aber nicht übergeben werden, da *getK* nur für Formeln aufgerufen wird, die freie Variablen besitzen. Für den Fall, dass mehr als 2 freie Variablen in der Anfrage vorkommen, wird bei Eingabe der Anfrage ein Hinweis an den Benutzer ausgegeben. Somit wird die Methode *getK* nur für gültige Eingaben aufgerufen.

Getestete Eingaben Es wurden die folgenden Eingaben aus den Äquivalenzklassen der gültigen Eingaben getestet:

1. Für $n = 1$: *armbruch(X)*
2. Für $n = 2$: *krankheit(X, Y)*

Erwartete Ausgaben Folgende Ausgaben sind dabei zu erwarten:

1. Für $n = 1$: $k = 5$
2. Für $n = 2$: $k = 24$

Beobachtete Ausgaben Die beobachteten Ausgaben

1. Für $n = 1$: $k = 5$
2. Für $n = 2$: $k = 24$

decken sich mit den erwarteten Ausgaben, daher kann der Test der Methode *getK* positiv abgeschlossen werden.

22.2.2 Testmodul getM

Die Methode *getM* erhält als Übergabeparameter eine Anfrage $\Phi(x_1, ..., x_n)$ (mit $n \in \{1, 2\}$), das Benutzerwissen *log*, die potentiellen Geheimnisse *pot_ sec* und den Wert k (mit Methode *getK* berechnet). Sie liefert eine Zahl m zurück. Dabei ist m der minimale Wert, bei welchem $Complete(\Phi(x_1, ..., x_n), m)$ mit dem Benutzerwissen *log* keine potentiellen Geheimnisse impliziert.

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die gültigen Eingaben der Methode

```
int getM (Formula f, LinkedList<Formula> log,
    LinkedList<Formula> pot_sec, int k)
```

bestehen aus Anfragen der Form $\Phi(x_1, ..., x_n)$ (mit $n \in \{1, 2\}$).

Ungültige Eingaben Die Äquivalenzklassen für die ungültigen Eingaben bestehen aus Aufrufen der Methode *getM* mit Anfragen die keine freien Variablen oder mehr als zwei freie Variablen besitzen. Dies ist, wie schon bei dem Testmodul von der Methode *getK* beschrieben, durch die interne Struktur des Programms nicht möglich. Die restlichen Übergabeparameter können zu keinem fehlerhaften Verhalten der Methode führen, da selbst falls $k = 0$ ist oder das Benutzerwissen bzw. die potentiellen Geheimnisse leer sind, dennoch korrekt weitergearbeitet werden kann. Somit wird die Methode *getM* nur für gültige Eingaben aufgerufen.

Getestete Eingaben Es wurden die folgenden Eingaben aus den Äquivalenzklassen der gültigen Eingaben getestet:
Für $n = 1$:

1. Anfrage: $armbruch(X)$, log=$\emptyset$, pot_sec=$\{armbruch(muenchen)\}$
2. Anfrage: $armbruch(X)$, log=$\emptyset$, pot_sec=$\{not\ armbruch(muenchen)\}$
3. Anfrage: $armbruch(X)$, log=$\emptyset$, pot_sec=$\{not\ armbruch(alfred)\}$

Für $n = 2$:

1. Anfrage: $krankheit(X, Y)$, log=$\emptyset$, pot_sec=$\{krankheit(frank, beinbruch)\}$
2. Anfrage: $krankheit(X, Y)$, log=$\emptyset$, pot_sec=$\{not\ krankheit(alfred, armbruch)\}$
3. Anfrage: $krankheit(X, Y)$, log=$\emptyset$, pot_sec=$\{not\ krankheit(dortmund, muenchen)\}$

Erwartete Ausgaben Folgende Ausgaben sind dabei zu erwarten:

Für $n = 1$:

1. $m = 5$
2. $m = 7$

3. $m = 5$

Für $n = 2$:

1. $m = 24$
2. $m = 24$
3. $m = 42$

Beobachtete Ausgaben Die beobachteten Ausgaben für $n = 1$:

1. $m = 5$
2. $m = 7$
3. $m = 5$

und für $n = 2$:

1. $m = 24$
2. $m = 24$
3. $m = 42$

decken sich mit den erwarteten Ausgaben, daher kann der Test der Methode *getK* positiv abgeschlossen werden.

22.2.3 Testmodul getKOpt

Die Methode *getKOpt* liefert eine Zahl k^* zurück. Dabei ist k^* der maximale Wert, bei welchem die Formeln $\Phi(c_{k^*}), Complete(\Phi(x_1, ..., x_n), m)$ und das Benutzerwissen *log* keine potentiellen Geheimnisse implizieren.

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die gültigen Eingaben der Methode

```
getKOpt(Formula f, LinkedList<Formula> log,
    LinkedList<Formula> pot_sec, int m)
```

bestehen aus Anfragen der Form $\Phi(x_1, ..., x_n)$ (mit $n \in \{1, 2\}$).

Ungültige Eingaben Wie bereits bei *getK* beschrieben, sind ungültige Eingaben durch die interne Struktur des Programms nicht möglich.

Getestete Eingaben Es wurden die folgenden Eingaben aus der Äquivalenzklassen der gültigen Eingaben getestet:
Für $n = 1$:

1. Anfrage: $armbruch(X)$, log=$\{armbruch(yu)\}$, pot_sec=$\{armbruch(alfred)\}$
2. Anfrage: $armbruch(X)$, log=$\{armbruch(yu)\}$, pot_sec=$\{armbruch(dortmund)\}$
3. Anfrage: $armbruch(X)$, log=$\{armbruch(yu)\}$, pot_sec=$\{not\ armbruch(dortmund)\}$

Für $n = 2$:

1. Anfrage: $krankheit(X, Y)$, log=$\{krankheit(frank, beinbruch)\}$, pot_sec=$\{krankheit(alfred, beinbruch)\}$
2. Anfrage: $krankheit(X, Y)$, log=$\{krankheit(frank, beinbruch)\}$, pot_sec=$\{not\ krankheit(alfred, beinbruch)\}$
3. Anfrage: $krankheit(X, Y)$, log=$\{krankheit(frank, beinbruch)\}$, pot_sec=$\{not\ krankheit(alfred, dortmund)\}$

Erwartete Ausgaben Folgende Ausgaben sind dabei zu erwarten:

Für $n = 1$:

1. $kOpt = 4$
2. $kOpt = 5$
3. $kOpt = 6$

Für $n = 2$:

1. $kOpt = 24$
2. $kOpt = 24$
3. $kOpt = 30$

Beobachtete Ausgaben Die beobachteten Ausgaben für $n = 1$:

1. $kOpt = 4$
2. $kOpt = 5$
3. $kOpt = 6$

und für $n = 2$:

1. $kOpt = 24$
2. $kOpt = 24$
3. $kOpt = 30$

decken sich mit den erwarteten Ausgaben, daher kann der Test der Methode *getK* positiv abgeschlossen werden.

22.2.4 Testmodul getMOpt

Die Methode *getMOpt* liefert eine Zahl m^* zurück. Dabei ist m^* ist der minimale Wert, bei welchem $\Phi(c_{k^*}), Complete(\Phi(x_1, ..., x_n), m^*)$ und das Benutzerwissen *log* keine potentiellen Geheimnisse implizieren.

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die gültigen Eingaben der Methode

```
getMOpt(Formula f, LinkedList<Formula> log,
    LinkedList<Formula> pot_sec, int j)
```

bestehen aus Anfragen der Form $\Phi(x_1, ..., x_n)$ (mit $n \in \{1, 2\}$).

Ungültige Eingaben Wie bereits bei *getK* beschrieben, sind ungültige Eingaben durch die interne Struktur des Programms nicht möglich.

Getestete Eingaben Es wurden die folgenden Eingaben aus den Äquivalenzklassen der gültigen Eingaben getestet:
Für $n = 1$:

1. Anfrage: $armbruch(X)$, log=$\{armbruch(yu)\}$, pot_sec=$\{armbruch(alfred)\}$
2. Anfrage: $armbruch(X)$, log=$\{armbruch(yu)\}$, pot_sec=$\{armbruch(dortmund)\}$
3. Anfrage: $armbruch(X)$, log=$\{armbruch(yu)\}$,
   pot_sec=$\{not\ armbruch(dortmund)\}$

Für $n = 2$:

1. Anfrage: $krankheit(X, Y)$, log=$\{krankheit(frank, beinbruch)\}$,
   pot_sec=$\{krankheit(alfred, beinbruch)\}$
2. Anfrage: $krankheit(X, Y)$, log=$\{krankheit(frank, beinbruch)\}$,
   pot_sec=$\{not\ krankheit(alfred, beinbruch)\}$
3. Anfrage: $krankheit(X, Y)$, log=$\{krankheit(frank, beinbruch)\}$,
   pot_sec=$\{not\ krankheit(alfred, dortmund)\}$

Erwartete Ausgaben Folgende Ausgaben sind dabei zu erwarten:

Für $n = 1$:

1. $mOpt = 4$
2. $mOpt = 5$
3. $mOpt = 6$

Für $n = 2$:

1. $mOpt = 24$
2. $mOpt = 24$
3. $mOpt = 30$

Beobachtete Ausgaben Die beobachteten Ausgaben für $n = 1$:

1. $mOpt = 4$
2. $mOpt = 5$
3. $mOpt = 6$

und für $n = 2$:

1. $mOpt = 24$
2. $mOpt = 24$
3. $mOpt = 30$

decken sich mit den erwarteten Ausgaben, daher kann der Test der Methode *getK* positiv abgeschlossen werden.

22.3 Methode combinationToIndex

Die Methode *combinationToIndex* liefert zu zwei übergebenen Konstantennamen aus dem Wörterbuch den entsprechenden Index der Aufzählung bzgl. der Kombination der beiden freien Variablen. Ist die Anzahl der Variablen gleich *eins*, wird die Methode *wordToIndex* aufgerufen, hier verlassen wir uns auf die Korrektheit dieser Methode. Unser Wörterbuch sieht wie folgt aus:

1	alfred
2	gerd
3	hans
4	frank
5	yu
6	fred
7	cold
8	paranoia
9	cancer
10	lykanthropie

Eingaben für den Test

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die unkritischen Eingaben der Methode

```
combinationToIndex(String x, String y, int freeVarCount)
```

bestehen aus zwei bzw. einem Konstantennamen aus unserem Wörterbuch.

Ungültige Eingaben Die Äquivalenzklassen für die ungültigen Eingaben bestehen aus Aufrufen der Methode *combinationToIndex* mit mehr als zwei oder weniger als einem Konstantennamen, oder der Name der Konstanten existiert nicht in unserem Wörterbuch. In diesen Fällen wird von der Methode der Rückgabewert *-1* als Fehlerwert zurückgeliefert.

Getestete Eingaben Für gültige Eingaben:

1. *(„fred“,“ “,1)*
2. *(„hans“,“cold“,2)*

Für ungültige Eingaben:

1. *(„fed“,“ “,1)*
2. *(„hans“,“cold“,4)*

Assertions Die Assertions wurden alle nach einem festen Schema aufgebaut, so z.B. die Assertion für die erste Eingabe:

```
assertEquals(censor.combinationToIndex(
    "hans","cold",f.getFreeVariables().size()),true);
```

Erwartete Ausgaben Für gültige Eingaben:

1. 6
2. 37

Für ungültige Eingaben:

1. Programm gibt Fehler aus.
2. -1

Beobachtete Ausgaben Für alle Tests deckten sich die beobachteten Ausgaben mit den erwarteten.

22.4 Methode complete

Die Methode *complete* wird getestet, um sicherzustellen, dass das gewünschte Verhalten der Schnittstelle angesprochen wird. Das gewünschte Verhalten ist hier, dass in Abhängigkeit von der Anzahl der freien Variablen in der Anfrage eine geschlossene Formel der Art aufgebaut wird:

- für eine freie Variable:
  $(\forall x)[x = c_1 \vee ... \vee x = c_j \vee \neg\Phi(x)]$;

- für zwei freie Variablen:
  $(\forall x)(\forall y)[(x = c_1 \wedge y = c_1) \vee ... \vee (x = c_i \wedge y = c_l) \vee \neg\Phi(x, y)]$.

Eingaben für den Test

Unkritische Eingaben Die Äquivalenzklassen für die unkritischen Eingaben der Methode

```
complete(Formula f, int k)
```

entsprechen der Äquivalenzklassen für die gültigen Eingaben der Methoden

```
indexToCombination(int j, Formula f)
```

und

```
getFreeVariables(Formula f).
```

Im Rumpf der Methode *complete* werden diese beiden Methoden *indexToCombination* und *getFreeVariables* aufgerufen, die die Parameterliste von der Methode *complete* übergeben bekommen. Deswegen werden die Äquivalenzklassen in Abhängigkeit von der Anzahl der freien Variablen in der Formel f und in der Abhängigkeit von der ganzen Zahl j gebildet. Dabei gelten gleiche Regeln für die Formel und für die Zahl j wie bei *indexToCombination* und *getFreeVariables*.

Es werden die folgenden Eingaben aus der Äquivalenzklasse der unkritischen Eingaben für die Methode *complete* getestet:

- für eine freie Variable:

  1. $f = \{armbruch(X))\}$, $j = 1$
  2. $f = \{armbruch(X))\}$, $j = 3$

- für zwei freie Variablen:

  1. $f = \{krankheit(X, Y))\}$, $j = 1$
  2. $f = \{krankheit(X, Y))\}$, $j = 3$

Erwartete Ausgaben Folgende Ausgaben sind für die beiden Methoden zu erwarten:

- für eine freie Variable:
  1. *forall X (equals(X,frank) or (not armbruch(X)))*
  2. *forall X (equals(X,frank) or (equals(X,edith) or (equals(X,yu) or (not armbruch(X)))))*
- für zwei freien Variablen:
  1. *forall X forall Y ((equals(X,frank) and equals(Y,frank)) or (not krankheit(X, Y)))*
  2. *forall X forall Y ((equals(X,frank) and equals(Y,frank)) or ((equals(X,frank) and equals(Y,edith)) or ((equals(X,edith) and equals(Y,frank)) or (not krankheit(X, Y)))))*

Beobachtete Ausgaben Die beobachteten Ausgaben

- für eine freie Variable:
  1. *forall X (equals(X,frank) or (not armbruch(X)))*
  2. *forall X (equals(X,frank) or (equals(X,edith) or (equals(X,yu) or (not armbruch(X)))))*
- für zwei freien Variablen:
  1. *forall X forall Y ((equals(X,frank) and equals(Y,frank)) or (not krankheit(X, Y)))*
  2. *forall X forall Y ((equals(X,frank) and equals(Y,frank)) or ((equals(X,frank) and equals(Y,edith)) or ((equals(X,edith) and equals(Y,frank)) or (not krankheit(X, Y)))))*

decken sich mit den erwarteten Ausgaben, daher kann der Test hier positiv abgeschlossen werden.

Randwertprüfung Die kritischen Eingaben für die beiden Methoden werden durch die ungültige Eingaben für *indexToCombination* und *getFreeVariables* abgedeckt. Deswegen werden sie hier nicht weiter erläutert.

22.5 Methode generateOut

Die Methode *generateOut* erzeugt die non-totality assumption für die gegebene Anfrage.

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die gültigen Eingaben der Methode

```
generateOut(LinkedList log, Formula query, LinkedList pot_sec)
```

bestehen aus einer Liste aus Formeln *log*, einer Anfrage-Formel *query* und einer Liste aus Formeln *pot_ sec*. Dabei entsprechen die Formeln gültigen Formeln aus der Klasse *Formula*. In den Listen sollte mindestens ein Element enthalten sein und die Anfrage sollte ebenfalls nicht leer sein.

Ungültige Eingaben Die Äquivalenzklassen für die ungültigen Eingaben bestehen aus Aufrufen der Methode *generateOut* mit leeren Listen oder leerer Anfrage. Diese können nach der internen Struktur des Programms aber nicht übergeben werden, da eine Formel zuvor von einem Parser geprüft wird und leere Formeln nicht erlaubt sind. Daher entfällt hier der Test mit ungültigen Eingaben.

Getestete Eingaben

1. Anfrage: $armbruch(X)$, log $=\{armbruch(fred)\}$,
   pot_sec $=\{armbruch(hans)\ and\ beinbruch(alfred)\}$
2. Anfrage: $armbruch(X)\ and\ krankheit(X, cold)$, log $=\{armbruch(fred)\}$,
   pot_sec $=\{armbruch(hans)\ and\ beinbruch(alfred)\}$
3. Anfrage: $armbruch(X)\ and\ (not\ krankheit(X, cold))$, log $=\{armbruch(fred)\}$,
   pot_sec $=\{armbruch(hans)\ and\ beinbruch(alfred)\}$

Erwartete Ausgaben Folgende Ausgaben (in Syntax von Prover9) sind dabei zu erwarten:

1. $exists\ X\ (-armbruch(X)\ \&\ -beinbruch(X)\ \&\ -(X = FRED)\ \&\ -(X = ALFRED)\ \&\ -(X = HANS))$
2. $exists\ X\ all\ Y1\ all\ Y2\ (-armbruch(X)\ \&\ -beinbruch(X)\ \&\ -krankheit(X, Y2)\ \&\ -krankheit(Y1, X)\ \&\ -(X = FRED)\ \&\ -(X = ALFRED)\ \&\ -(X = HANS)\ \&\ -(X = COLD))$
3. $exists\ X\ all\ Y1\ all\ Y2\ (-armbruch(X)\ \&\ -beinbruch(X)\ \&\ -krankheit(X, Y2)\ \&\ -krankheit(Y1, X)\ \&\ -(X = FRED)\ \&\ -(X = ALFRED)\ \&\ -(X = HANS)\ \&\ -(X = COLD))$

Hierbei bedeutet „-“ die Negation.

Beobachtete Ausgaben Die beobachteten Ausgaben

1. $exists\ X\ (-armbruch(X)\ \&\ -beinbruch(X)\ \&\ -(X = FRED)\ \&\ -(X = ALFRED)\ \&\ -(X = HANS))$
2. $exists\ X\ all\ Y1\ all\ Y2\ (-armbruch(X)\ \&\ -beinbruch(X)\ \&\ -krankheit(X, Y2)\ \&\ -krankheit(Y1, X)\ \&\ -(X = FRED)\ \&\ -(X = ALFRED)\ \&\ -(X = HANS)\ \&\ -(X = COLD))$
3. $exists\ X\ all\ Y1\ all\ Y2\ (-armbruch(X)\ \&\ -beinbruch(X)\ \&\ -krankheit(X, Y2)\ \&\ -krankheit(Y1, X)\ \&\ -(X = FRED)\ \&\ -(X = ALFRED)\ \&\ -(X = HANS)\ \&\ -(X = COLD))$

decken sich mit den erwarteten Ausgaben, daher kann der Test der Methode *generateOut* positiv abgeschlossen werden.

22.6 Methode generateUNA

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die gültigen Eingaben der Methode

```
generateUNA(LinkedList log, Formula query, LinkedList pot_sec)
```

bestehen aus einer Liste von Formeln *log*, einer Anfrage-Formel *query* und einer Liste von Formeln *pot_ sec*. Dabei entsprechen die Formeln gültigen Formeln aus der Klasse *Formula*. In den Listen sollte mindestens ein Element enthalten sein und die Anfrage sollte ebenfalls nicht leer sein.

Ungültige Eingaben Die Äquivalenzklassen für die ungültigen Eingaben bestehen aus Aufrufen der Methode *generateUNA* mit leeren Listen oder leerer Anfrage. Diese können nach der internen Struktur des Programms aber nicht übergeben werden, da eine Formel zuvor von einem Parser geprüft wird und leere Formeln nicht erlaubt sind. Daher entfällt hier der Test mit ungültigen Eingaben.

Getestete Eingaben

1. Anfrage: $(armbruch(X)\ and\ krankheit(alfred,\ cold))$, log $=\{armbruch(fred)\}$, pot_sec $=\{exists\ X\ armbruch(X)\}$
2. Anfrage: $(armbruch(hans)\ and\ krankheit(X,\ hans))$, log $=\{armbruch(fred)\}$, pot_sec $=\{exists\ X\ armbruch(X)\}$
3. Anfrage: $(armbruch(hans)\ and\ krankheit(X,\ gerd))$, log $=\{armbruch(fred)\}$, pot_sec $=\{armbruch(hans)\ and\ beinbruch(alfred)\}$

Erwartete Ausgaben

1. $(FRED! = ALFRED)$ & $(FRED! = COLD)$ & $(ALFRED! = COLD)$
2. $(FRED! = HANS)$
3. $(FRED! = ALFRED)$ & $(FRED! = HANS)$ & $(FRED! = GERD)$ & $(ALFRED! = HANS)$ & $(ALFRED! = GERD)$ & $(HANS! = GERD)$

Beobachtete Ausgaben

1. $(FRED! = ALFRED)$ & $(FRED! = COLD)$ & $(ALFRED! = COLD)$
2. $(FRED! = HANS)$
3. $(FRED! = ALFRED)$ & $(FRED! = HANS)$ & $(FRED! = GERD)$ & $(ALFRED! = HANS)$ & $(ALFRED! = GERD)$ & $(HANS! = GERD)$

22.7 Methode isImplied

Äquivalenzklassen

Gültige Eingaben Die Äquivalenzklassen für die gültigen Eingaben der Methode

```
isImplied(LinkedList log, Formula query, LinkedList pot_sec)
```

bestehen aus einer Liste aus Formeln *log*, einer Anfrage-Formel *query* und einer Liste aus Formeln *pot_ sec*. Dabei entsprechen die Formeln gültigen Formeln aus der Klasse *Formula*. In den Listen sollte mindestens ein Element enthalten sein und die Anfrage sollte ebenfalls nicht leer sein.

Ungültige Eingaben Die Äquivalenzklassen für die ungültigen Eingaben bestehen aus Aufrufen der Methode *isImplied* mit leeren Listen oder leerer Anfrage. Diese können nach der internen Struktur des Programms aber nicht übergeben werden, da eine Formel zuvor von einem Parser geprüft wird und leere Formeln nicht erlaubt sind. Daher entfällt hier der Test mit ungültigen Eingaben.

Getestete Eingaben

1. `log = exists X beinbruch(X), pot_sec = exists X armbruch(X), query = exists X armbruch(X)`

2. `log = exists X beinbruch(X), pot_sec = exists X armbruch(X), query = armbruch(gerd)`

3. `log = exists X beinbruch(X), pot_sec = exists X armbruch(X), query = exists X krankheit(X, gerd)`

Erwartete Ausgaben Folgende Ausgaben muss ein korrekt arbeitender Algorithmus ausgeben:

1. `true`
2. `true`
3. `false`

Beobachtete Ausgaben Folgende Ausgaben wurden tatsächlich beobachtet:

1. `true`
2. `true`
3. `false`

22.8 Testmodul Dictionary

Die drei Methoden *getWord()*, *wordToIndex()* und *indexToWord()* von der Klasse *Censor* werden innerhalb der Klasse *xxDictionary* getestet. Die Testklasse enthält folgende Testmethoden:

- *testGetWord()*
- *testWordToIndex()*
- *testIndexToWord()*

Es wurde ein Wörterbuch (Dictionary) in der SQL-Datenbank angelegt. Das Wörterbuch ist endlich und hat (zur Zeit) *30* Elemente. Jedes Element hat einen Index.
Jede diese Methoden testet die entsprechende Methode von der Klasse *Censor* anhand der verschiedenen Übergabewerte.

Methode getWord() Hierbei wird die Methode *getWord()* von der Klasse *Censor* getestet.

Gültige Eingaben Alle Zahlen vom Typ Integer.

Getestete Eingaben Wir haben versucht, die Randwerte testen und noch die Indizes, die außerhalb des Wörterbuchs liegen.

- 1
- 30
- 0
- 31

Die Übereinstimmung von den Elementen mit den entsprechenden Indizes wird hier getestet.
Mit der Methode *assertEquals()* werden die jeweiligen Werte überprüft.

Erwartete Ausgaben Folgende Ausgaben sind dabei zu erwarten:

- frank
- peiman
- *Leerer String*
- *Leerer String*

Beobachtete Ausgaben Es liegen keine Abweichungen von den erwarteten Ausgaben vor.

Methode wordToIndex() Hierbei wird die Methode *wordToIndex()* von der Klasse *Censor* getestet.
Ein Wort aus dem Wörterbuch wird als Eingabe genommen und sein Index wird erwartet.
Mit der Methode *Assert.assertEquals()* werden die jeweiligen Werte überprüft.

Gültige Eingaben Alle Werte vom Typ String.

Getestete Eingaben Hierbei wurden Wörter aus dem Wörterbuch sowie Wörter, die nicht im Wörterbuch existieren, getestet.

- frank
- peiman
- parsa
- *Leerer String*

Erwartete Ausgaben

- 1
- 30
- 0
- 0

Beobachtete Ausgaben Es liegen keine Abweichungen von den erwarteten Ausgaben vor.

Methode indexToWord() Hierbei wird die Methode *indexToWord()* von der Klasse *Censor* getestet.
Es wird ein Index vom Wörterbuch als Eingabe in die Testmethode gegeben und es wird erwartet, dass die Wörter von 1 bis zum eingegebenen Index ausgegeben werden.

Gültige Eingaben Alle Zahlen vom Typ Integer.

Getestete Eingaben Hierbei wurden ganze Zahlen innerhalb vom Wertebereich des Wörterbuchindexes und auch außerhalb von diesem Wertebereich getestet.

- 5
- 1
- 30
- 32
- 0

Erwartete Ausgaben

- Die Wörter mit den Indizes 1 bis 5
- Das erste Wort im Wörterbuch
- Die Wörter mit den Indizes 1 bis 30 (Alle Wörter in unserem bereits existierenden Wörterbuch)
- Die Wörter mit den Indizes 1 bis 30 (Alle Wörter in unserem bereits existierenden Wörterbuch)
- Leere Linkedlist wird ausgegeben

Beobachtete Ausgaben Es liegen keine Abweichungen von den erwarteten Ausgaben vor.

Teil VI
Fazit

23 Reflexion

23.1 Ziele der Projektgruppe

Ziel der Projektgruppe war es, auf die Frage, ob man tatsächlich mit effizienten informatorischen Mitteln inhaltliche Informationen vertraulich halten kann, eine positive Antwort zu geben. Hierbei sollte mindestens der Entwurf einer modular aufgebauten „praktischen Architektur“ für die kontrollierte Anfrageauswertung für relationale Datenbanken gelingen.
Hinter einer solch allgemein gehaltenen Beschreibung für ein Programm verbargen sich natürlich zahlreiche und vielfältige Herausforderungen, die mit der Implementierung eines solchen Programms einher gehen. Welche das im einzelnen waren, soll vor allem der vorliegende Endbericht wiedergeben. Als Beispiele seien hier die Datenstruktur für Formeln und die Übersetzung einer Formel in eine passende SQL-Anfrage genannt. Abbildung 1 auf Seite 13 zeigt die theoretische Architektur für das Programm, an der sich die Projektgruppe bei der Umsetzung auch stark orientierte.

23.2 Vorliegendes Produkt

Das nun vorliegende Produkt erfüllt die Anforderungen mit folgenden Features:

- Kontrollierte Anfrageauswertung für geschlossene Anfragen vermöge der Verweigerungs-, Lügen- und kombinierten Methode.
- Kontrollierte Anfrageauswertung für offene Anfragen vermöge der alternativen kombinierten Methode.
- Sicherstellung der Konsistenz und Minimalität des Benutzerwissens.
- Prüfung jeder Anfrage durch einen Optimierer und u.U. Festlegung einer alternativen Ausführungsmethode.
- Trennung der Benutzer in die Rollen *Administrator* und *Benutzer*.
- Erkennung von Fehlersituationen und Benutzerunterstützung durch informative Fehlerausgaben.

Der entwickelte Prototyp entspricht im Prinzip genau der theoretischen Architektur, wobei im Zensor der o.g. Optimierer steckt. Dieser manipuliert aber nur die Abläufe innerhalb des Zensors, hat also keine Auswirkungen auf die Architektur. Abbildung 51 zeigt das endgültige Klassendiagramm. Bei dessen Betrachtung erkennt man klar den modularen Aufbau. Zensor, Modifikator und Schnittstelle zum Theorembeweiser sind jeweils in eigenen Klassen implementiert. Insbesondere ist der Theorembeweiser austauschbar. Hierzu muss eine andere Klasse lediglich die Methoden aus der Theorembeweiserschnittstelle passend zu einem anderen externen Theorembeweiser implementieren.

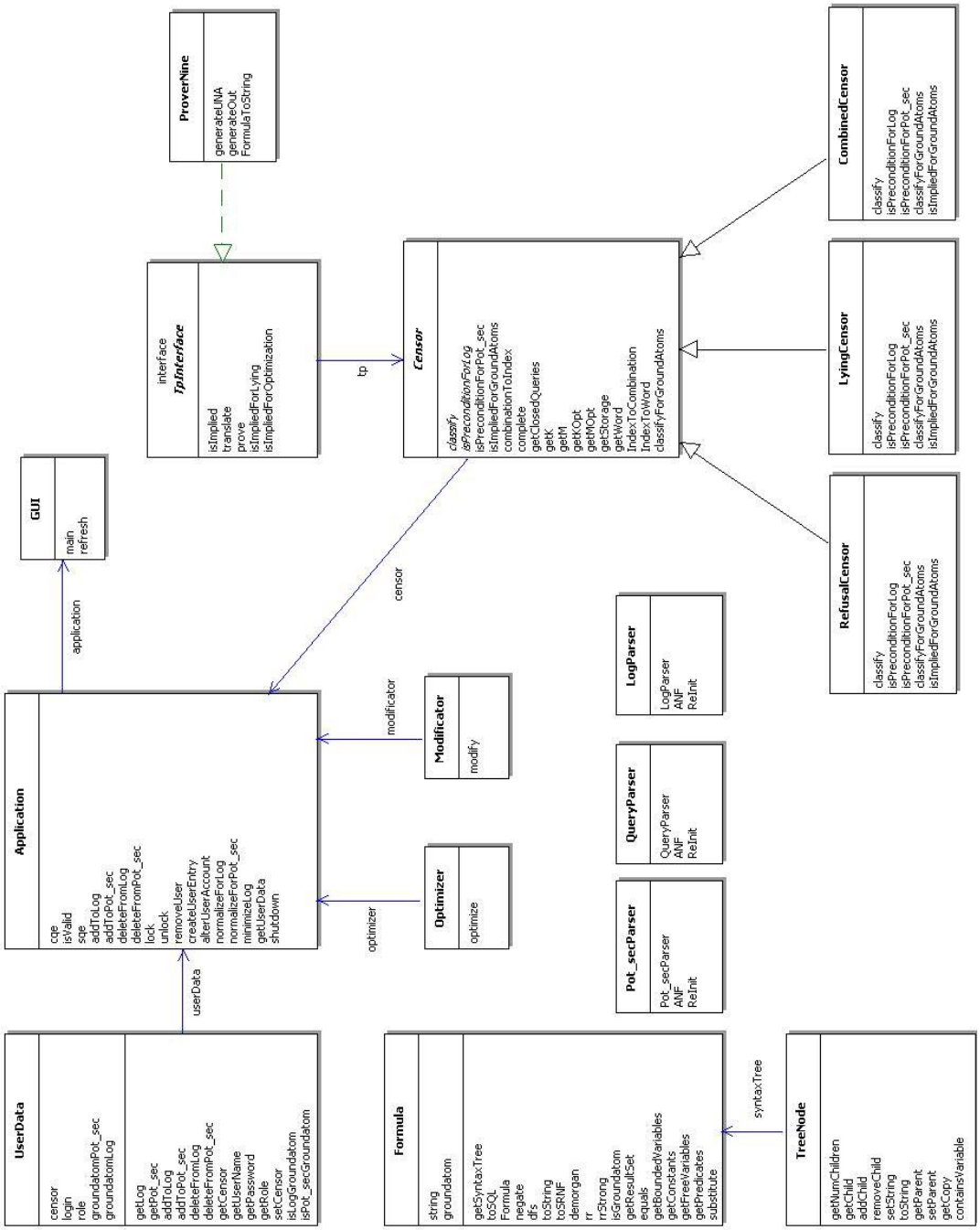

Abbildung 51: Klassendiagramm des Prototyps

23.3 Kritik

Es gibt einige Punkte, die auch nach Abschluss der Projektgruppe weiterhin unbearbeitet geblieben sind. So wurde die kontrollierte Anfrageauswertung für offene Anfragen nur mit der alternativen kombinierten Methode implementiert. Alle anderen Methoden sind noch nicht verfügbar.
Ein anderes Problem ist die Darstellung des Vollständigkeitstests. Bereits bei einer Wörterbuchgröße von 100 Wörtern und zwei frei vorkommenden Variablen in einer Anfrage kann der Vollständigkeitstest 10000 Disjunkte enthalten. Hier ergibt sich sowohl ein Darstellungs- als auch ein Speicherungsproblem. Zum einen kann der Oracle Datentyp *VARCHAR* lediglich 4000 Zeichen aufnehmen, zum anderen nimmt das Parsen solch großer Formeln enorm viel Speicher- und Rechenkapazität in Anspruch.
Um dieses Problem in den Griff zu bekommen, gab es innerhalb der Projektgruppe zwei Vorschläge: Entweder man verwendet den Oracle Datentyp *CLOB*, der bis zu 4GB Daten aufnehmen kann, für die Speicherung der Formeln. Diese Lösung bringt den Nachteil mit sich, dass nicht der gesamte Inhalt eines Datenfeldes dieses Typs auf einmal gelesen werden kann, sondern erst umständlich durch einen gepufferten Eingabestrom gelesen werden muss. Darüber hinaus wird hierdurch nicht das Darstellungsproblem auf dem Bildschirm und das Problem des Parsens gelöst. Eine andere Lösung sieht vor, den Vollständigkeitstest nicht voll ausgeschrieben zu speichern, sondern eine komprimierte Darstellungsweise zu finden. Eine solche Darstellung könnte z.B. sein, dass man eine Formel $Complete(\Phi(X,Y),9999)$ eben genau so abspeichert und erst bei Bedarf, also wenn sie dem Theorembeweiser zugeführt wird, expandiert. Für beide Lösungen ist aber weiterhin das Problem zu lösen, dass der Theorembeweiser an einem gewissen Punkt aussteigt.

23.4 Vorgehensmodell

Zuletzt soll noch einmal das Vorgehensmodell angesprochen werden. Die Projektgruppe hat sich entschlossen, das Produkt in mehreren Durchläufen gemäß Spiralmodell sukzessive zu entwickeln. Dabei sind nach und nach neue Features in das Programm eingeflossen. Diese Vorgehensweise spiegelt sich auch im Endbericht wider, der in drei Hauptteile unterteilt ist, die jeweils den Entwicklungen in den drei Durchläufen entsprechen. Diese Vorgehensweise bot sich insbesondere aufgrund der Tatsache, dass sich offene Anfragen durch eine Folge von geschlossenen Anfragen auswerten lassen, an. So konnten zunächst in der ersten Iteration Grundfunktionen für syntaktisch einfache, geschlossene Anfragen implementiert werden. In der zweiten Iteration wurde der Umfang der Eingabesprache erweitert, wobei aber weiterhin nur geschlossene Anfragen gültig waren, und die Mechanismen zur kontrollierten Anfrageauswertung für diese Anfragen verfeinert. So wurde schließlich im dritten Durchlauf die Menge der erlaubten Anfragen um offene Anfragen mit maximal zwei frei vorkommenden Variablen erweitert.

Mit einer anderen Vorgehensweise, z.B. Extreme Programming, wäre die Realisierung des Projekts wahrscheinlich nicht so gut gelungen. Grund dafür ist, wie bereits oben beschrieben, der modulare Aufbau und die Möglichkeit, das Programm sukzessive um Funktionalitäten zu erweitern.

Literatur

[ACA02] M. Abbey, M. Corey, and I. Abramson. *Oracle9i für Einsteiger.* Hanser, München, Wien, 2002.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, München, 1995.

[Aul03] M. Ault. *Oracle-Datenbanken.* mitp-Verlag, Bonn, 2003.

[BB01] Joachim Biskup and Piero A. Bonatti. Lying versus refusal for known potential secrets. *Data Knowl. Eng.*, 38(2):199–222, 2001.

[BB04a] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation for enforcing confidentiality in complete information systems. *Int. J. Inf. Sec.*, 3(1):14–27, 2004.

[BB04b] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation for known policies by combining lying and refusal. *Ann. Math. Artif. Intell.*, 40(1-2):37–62, 2004.

[BB06] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation with open queries for a decidable relational submodel. In Jürgen Dix and Stephen J. Hegner, editors, *FoIKS*, volume 3861 of *Lecture Notes in Computer Science*, pages 43–62, Berlin, 2006. Springer.

[BH01] Becker and Hähnle. Vorlesungsskript Automatisches Beweisen. Karlsruhe, 2001.

[Bis95] Joachim Biskup. *Grundlagen von Informationssystemen.* Vieweg, Wiesbaden, 1995.

[Buh04] Axel Buhl. *Grundkurs Software-Projektmanagement.* Hanser, München, 2004.

[CB03] G. Kern-Isberner C. Beierle. *Methoden wissensbasierter Systeme.* Vieweg, Wiesbaden, 2003.

[FC05] J. Farley and W. Crawford. *Java Enterprise in a Nutshell.* O'Reilly, Sebastopol, 2005.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, Amsterdam, 2001.

[Sch06] S. Schulz. *E 0.9 user manual*, 2006.

[TN01] M. Therault and A. Newman. *Oracle Security Handbook.* McGraw-Hill, New-York, 2001.

[Wei06] G. Weikum. Vorlesungsskript Informationssysteme. Saarbrücken, 2006.

Index