Bachelorarbeit

Benutzerautorisierung in Informationssystemen mit parametrisierten Rollen

Sebastian Schweizer

vorgelegt am: 17.04.2014

am Fachbereich Informatik der
Technischen Universität Kaiserslautern

Studiengang:	Bachelor Informatik
Erstgutachter:	Prof. Dr. Arnd Poetzsch-Heffter
Zweitgutachter:	M. Sc. Mathias Weber

Benutzerautorisierung in Informationssystemen mit parametrisierten Rollen

Zusammenfassung

Moderne Informationssysteme speichern eine Vielzahl von vertraulichen Daten, welche nicht in falsche Hände geraten dürfen. Als Benutzerautorisierung bezeichnet man dabei die Überprüfung, ob ein Benutzer eine bestimmte Berechtigung hat. Rollenbasierte Ansätze vereinfachen die Rechteverwaltung, haben aber zumeist Einschränkungen in der Flexibilität.

Diese Arbeit stellt ein Verfahren vor, bei dem Rollen mit einem Parameter versehen werden können und so durch eine endliche Beschreibung zur Designzeit unbeschränkt viele Rollen zur Laufzeit definiert werden können. Eine einmalige Beschreibung der Rechte von *Projektleiter für Projekt X* reicht aus, damit für jedes im System verwaltete Projekt eine Rolle für den Projektleiter mitsamt Berechtigungen definiert ist. Dabei werden auch Vererbungen der Rechte zwischen den Rollen mittels einer Hierarchie der Rollen ermöglicht.

In einer exemplarischen Implementierung wird eine Sicherheitsschicht geschaffen, die die Prüfung von Berechtigungen übernimmt und zwischen die GUI-Schicht und die Datenzugriffsschicht geschaltet werden kann. Das Prüfen der Berechtigungen in einer eigenen, tiefer angesiedelten Schicht ermöglicht eine robuste Zugriffskontrolle.

Abstract

Modern information systems manage confidential data which must not fall into the wrong hands. User authorization is the process of checking whether a user has been granted specific permissions. A role based approach simplifies access control at the cost of flexibility.

We are introducing a method to define roles with parameters. This way, a finite description at design time can define an arbitrary number of roles at runtime. For example, a single description of the role *project manager for project X* suffices to define the role of project manager in every project managed by the system. It is also possible to structure roles in a hierarchical way and to inherit permissions from other roles.

An exemplary implementation of a security layer undertakes the task of checking permissions. It may be placed between the GUI and the persistence layer. Checking permissions in a dedicated deeper layer enables robust access control.

Eidesstattliche Erklärung

Ich erkläre an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel *Benutzerautorisierung in Informationssystemen mit parametrisierten Rollen* selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe und dass ich alle Stellen, die ich wörtlich oder sinngemäß aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe. Die Arbeit hat bisher in gleicher oder ähnlicher Form oder auszugsweise noch keiner Prüfungsbehörde vorgelegen.

Ich versichere, dass die eingereichte schriftliche Fassung der auf dem beigefügten Medium gespeicherten Fassung entspricht.

Kaiserslautern, den 17.04.2014

(Sebastian Schweizer)

Inhaltsverzeichnis

1. Rollenbasierte Zugriffssysteme in Informationssystemen

Moderne Informationssysteme sind von der Interaktion vieler Benutzer geprägt, je nach Definition sind die Benutzer auch Teil des Systems. In dieser Arbeit bezieht sich der Begriff „Informationssystem“ auf den technischen Software-Anteil, der für die Verarbeitung und Speicherung der zu verwaltenden Daten verantwortlich ist. Bei immer größeren digital verarbeiteten Datenmengen ist es wichtig, die Vertraulichkeit zu wahren, das heißt nur wer unter rechtlichen und moralischen Aspekten ein berechtigtes Interesse an den Informationen hat, soll sie abrufen können. Zudem müssen Daten vor unbefugter Manipulation geschützt werden. Dazu ist es von großer Bedeutung, dass alle Benutzer des Systems nur die Berechtigungen haben, die für ihre Position erforderlich sind. Dies sorgt zum Einen dafür, dass ein Benutzer keine unberechtigten Zugriffe auf für ihn nicht gedachte Daten tätigen kann, aber begrenzt auch im Fall, dass in einen Benutzeraccount eingedrungen wurde, den potentiellen Schaden auf die diesem Account zur Verfügung stehenden Berechtigungen.

Man unterscheidet hier zwischen zwei Aspekten:

> „**Authentifizierung:** Während der Authentifizierung wird die Identität eines Benutzers verifiziert. Dies kann unter anderem durch Eingabe eines Paßworts, Übertragung von Zertifikaten oder Verwendung von Chip-/Magnetkarten geschehen.
>
> **Autorisierung:** Bei der Autorisierung von Zugriffen wird geprüft, ob der Benutzer das Recht hat, die angegebene Operation auf dem angeforderten Objekt ausführen zu dürfen. Ist das Ergebnis der Autorisierung negativ, wird dem Benutzer der Zugriff verweigert.“[Fec98, S. 3]

Neben der Verifikation eines Benutzers („Ist er das wirklich?“) kann auch die Feststellung seiner Identität („Wer ist das?“) zur Authentifizierung gezählt werden - oft hängen diese beiden Fragestellungen stark zusammen. Für die Authentifizierung in Systemen, bei denen sich der Benutzer lediglich mit einem ihm zugeordneten Anmeldenamen und einem Passwort anmelden muss, reicht eine einfache Liste mit diesen Zugangsdaten im System aus. Ggf. müssen mit dem Anmeldenamen noch weitere Informationen wie Vor- und Zuname der Person verknüpft werden.

Die Autorisierung hingegen ist ein komplexerer Prozess. Moderne Informationssysteme speichern viele Daten und bieten unterschiedlichste Funktionen an. Jedem Benutzer einzeln für jedes Objekt, auf das er zugreifen können soll, die Berechtigung zu gewähren, ist sehr aufwändig. Beim Übertragen der Berechtigungen, die sich

aus Überlegungen im Kontext des Informationssystems ergeben, in das System selbst, können Fehler gemacht werden. Nur ein Mensch kann entscheiden, welche Berechtigungen gewährt werden sollen, und muss diese Entscheidung fehlerfrei im System abbilden.

Untersucht man die Benutzer unter dem Kontext, für den das Informationssystem arbeitet, so stellt man häufig fest, dass es verschiedene Personengruppen mit ähnlichen Aufgaben gibt. Dies kann man auch auf die Rechteverwaltung übertragen:

> „Bei der rollenbasierten Zugriffskontrolle werden die Rechte nicht direkt an die zugreifenden Subjekte [Benutzer des Systems, Anm. d. Verf.] vergeben, sondern zu Rollen zusammengefasst. Die zugreifenden Subjekte werden erst in einem zweiten Schritt den Rollen zugeordnet. Rollen stellen somit die Zusammenfassung von Rechten dar, die zur Erfüllung von mit ihnen verbundenen Aufgaben notwendig sind. Die Hauptmotivation hinter der rollenbasierten Zugriffskontrolle liegt in der Tatsache, dass bei der Ausführung von Geschäftstätigkeiten nicht die Identität der einzelnen Person, sondern die organisatorischen Verantwortlichkeiten der Person im Vordergrund stehen.“[WW07, S. 440]

Neben der besseren Abbildung der Geschäftstätigkeiten in das Informationssystem vereinfacht dieses Vorgehen auch die Rechteverwaltung. Ändern sich die Aufgaben einer Rolle so müssen die Rechte nur einmalig angepasst werden, statt für jeden Benutzer einzeln. Zudem ist es so einfacher, neuen Benutzern ihre Rechte zuzuweisen, da man nur noch aus einer kleineren Menge an Rollen wählen muss statt aus der Liste aller Objekte. Außerdem sichern die Rollen eine Konsistenz der Rechte: Alle Mitglieder derselben Rolle haben die gleichen Rechte.

Um das System flexibel zu gestalten ist es denkbar, den Benutzern mehrere Rollen zuzuweisen. Dann hat der Benutzer die Vereinigung der Berechtigungen aller seiner Rollen. Dies ist sinnvoll, da auch in der Geschäftswelt Personen mit unterschiedlichen Aufgaben betraut werden können, beispielsweise kann ein Mitarbeiter in einer Firma an mehreren Projekten gleichzeitig teilnehmen.

1.1. Betrachtung eines etablierten Frameworks: Spring Security

Im Folgenden wird ein etabliertes rollenbasiertes Zugriffssystem kurz betrachtet. Alle Möglichkeiten des Systems zu evaluieren würde den Rahmen dieser Arbeit sprengen. *Spring Security* ist ein Framework zur Benutzerauthentifikation und -autorisierung, das in vielen auf dem Spring Framework basierenden Anwendungen zum Einsatz kommt. („Spring Security is a powerful and highly customizable authentication

and access-control framework. It is the de-facto standard for securing Spring-based applications.“[ATW, Preface]).

Spring Security bietet verschiedene Möglichkeiten zum Verwalten der Benutzerkonten für die Authentifizierung. Eine Möglichkeit ist die Speicherung in einer Datenbank, wie in Listing 1 veranschaulicht. Eine Relation speichert Benutzernamen und Passwort, eine zweite verknüpft in einer 1:n Relation die Benutzer mit *authorities*(Rollen). Alternativ können die Benutzer und deren Rollen auch in einem XML-Schema definiert werden:

```
create table users(
  username varchar_ignorecase(50) not null primary key,
  password varchar_ignorecase(50) not null
);
create table authorities (
  username varchar_ignorecase(50) not null,
  authority varchar_ignorecase(50) not null,
  constraint fk_authorities_users foreign key(username) references
     users(username)
);
create unique index ix_auth_username on authorities (username,
    authority);
```

Listing 1: Speichern der Benutzer und Rollen in einer relationalen Datenbank, vgl. [ATW, Appendix 1.1]

Der zweite Teil in Listing 2 zeigt, wie man den Zugriff auf URLs beschränken kann. Mit `intercept-url` definiert man ein Pattern (Spring nutzt hier die gleiche Form von Patterns wie das bekannte Buildsystem Apache Ant)[1] und eine oder mehrere Rollen. Ruft jemand diese URL innerhalb der Website auf, so muss der Benutzer eine der angegebenen Rollen haben, ansonsten wird der Zugriff verweigert. Außerdem kann man innerhalb des Java-Codes Methoden mit Annontationen versehen, beispielsweise `@Secured("ROLE_ADMIN")`, sodass diese Methode nur von Benutzern mit der Admin-Rolle aufgerufen werden kann.

Auch eine Vererbungshierarchie zwischen den Rollen ist möglich, wie Listing 3 zeigt. Hier wird konfiguriert, dass jeder Admin auch ein Mitglied der Rolle Staff ist. Zudem ist jedes Staff-Mitglied (und damit transitiv auch die Admins) ein User. Jeder User hat schließlich noch alle Rechte der Gäste.

Neben dieser einfachen Konfiguration mit reinen Namen für Rollen sind auch deutlich komplexere Szenarien möglich, welche das Spring Security ACL Konzept

[1] „The pattern attribute takes an Ant Paths and the most specific URIs should appear first“ [ATW, Web Application Security 1.2]

```
<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="jimi" password="jimispassword"
          authorities="ROLE_USER, ROLE_ADMIN" />
      <user name="bob" password="bobspassword"
          authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
<http>
  <intercept-url pattern="/**" access="ROLE_USER" />
  <form-login />
  <logout />
</http>
```

Listing 2: Konfiguration in XML, vgl. [ATW, Getting Started 4.2.2]

```
<property name="hierarchy">
  <value>
    ROLE_ADMIN > ROLE_STAFF
    ROLE_STAFF > ROLE_USER
    ROLE_USER > ROLE_GUEST
  </value>
</property>
```

Listing 3: Hierarchie der Rollen [ATW, Authorization 1.4]

verwenden. Hierbei wird der rollenbasierte Ansatz jedoch nicht verwendet, sodass viele der zuvor genannten Vorteile dieses Ansatzes verloren gehen. Für weitere Informationen hierzu sei auf [ATW] verwiesen.

1.2. Parametrisierung der Rollen

In manchen Umgebungen sind Tätigkeiten mit organisatorischen Objekten verknüpft. Beispielsweise gibt es an einer Universität unter anderem Dozenten, Assistenten, Tutoren und Studenten, die alle in unterschiedlicher Art und Weise an Lehrveranstaltungen partizipieren. Allerdings gibt es mehrere Lehrveranstaltungen, sodass ein Dozent nicht einfach Dozent, sondern Dozent für eine bestimmte Lehrveranstaltung ist. Ein anderes Beispiel ist ein Unternehmen, in dem mehrere Projekte durchgeführt werden. In jedem Projekt gibt es Projektmanager, Projektleiter, Architekten, Entwickler, Tester, etc.

Diese organisatorischen Einheiten wie Lehrveranstaltungen oder Projekte können dynamisch sein, mit der Zeit werden es immer mehr: Jedes Semester gibt es wieder Vorlesungen und ein Unternehmen führt viele Projekte durch. Es ist durchaus vorstellbar, dass die Tätigkeiten der Dozenten, Assistenten, Tutoren und Studenten in Bezug auf Vorlesungen bzw. der unterschiedlichen Aufgabenträger der Projekte immer gleich sind und daher auch aus Sicht der Autorisierung eine Abbildung gewünscht ist, die dies ermöglicht. Statt für jedes Projekt einzeln die Rolle *Projektleiter für Projekt X* zu beschreiben und entsprechende Berechtigungen für Teile des Projektes X zu vergeben, wäre es denkbar, in einer einzigen Definition alle Projektleiter für alle jemals existierenden Projekte zu beschreiben.

Dies ist jedoch mit Rollen, die durch eine einfache Bezeichnung identifiziert werden, nicht möglich. Man muss unterscheiden zwischen der abstrakt definierten Rolle, die einen oder mehrere Parameter (Vorlesung bzw. Projekt) enthält, also *Assistent für Vorlesung X* oder *Projektleiter für Projekt X*, und der eigentlichen Rolle in der Realität, bei der der Parameter durch einen konkreten Wert ersetzt wurde. Ein solches System ist mit Spring Security auf Basis der Rollenarchitektur nicht möglich. Durch die Mächtigkeit dieses Frameworks im Zusammenhang mit den ACLs lässt sich eine solche Beziehung zwar auch ohne Rollen ausdrücken, jedoch wäre ein intuitiv konfigurierbares System, das den beschriebenen Ansatz verfolgt, in dieser Hinsicht besser. Man könnte den Parameter im ACL System durch eine `ObjectIdentity` abbilden und die einzelnen Benutzer im System als „security identity“ (`SID`) verwalten (vgl. [ATW, Additional Topics 1.2]).

Im Rahmen dieser Arbeit wird ein solches Zugriffssystem mit parametrisierten Rollen am Beispiel eines universitären Informationssystems entwickelt. Wie oben angedeutet geht es dabei um die Verwaltung von Vorlesungen mit den dazugehörigen Übungen und Prüfungen. Dabei wird auf das Framework XCend aufgebaut, welches im nächsten Kapitel vorgestellt wird. Das Beispielsystem wird in Kapitel 2.3 erläutert. Die mit dieser Arbeit entwickelte Lösung beschreibt aber auch einen theoretischen Ansatz, der sich in andere Umgebungen übertragen lässt. Kapitel 3.1 geht auf die nötige Erweiterung der Beschreibungssprache von XCend ein, die Implementierung in Java folgt in Kapitel 3.2. Kapitel 4 fasst die wesentlichen Punkte nochmals zusammen und gibt einen Ausblick auf mögliche Erweiterungen.

2. XCend: Eine Technologie zur Spezifizierung hierarchischer Datenstrukturen

XCend ist eine Technologie, die es erlaubt, hierarchische Datenstrukturen zu definieren und benutzen. Sie ist unabhängig von Sprach- oder Datentypparadigmen und bietet sprachabhängige Backends zur Codegenerierung für verschiedene Sprachen. Der Hauptnutzen von XCend ist die Ergänzung von Integritätsbedingungen zur hierarchischen Datenstruktur [Mic, start].

Die Daten liegen als eine strukturiert abgelegte Sammlung von Werten vor, welche in einem Dokument verwaltet werden. Dabei wird durch XCend sichergestellt, dass alle Zugriffe atomar und Modifikationen unter Beibehaltung der Konsistenz erfolgen. Konsistenz wird hierbei durch die Integritätsbedingungen spezifiziert [Mic14, S. 1-2].

2.1. Datenorganisation

Ein *Dokument* ist definiert als endliche Abbildung von *Orten* nach *Werten*, der Abbildungsreich repräsentiert den eigentlichen Inhalt des Dokumentes, das heißt eine Menge von einfachen Werten wie Integer oder Strings. Ein Wert kann also beispielsweise ein Name einer Person sein. Der Definitionsbereich der Abbildung repräsentiert die Struktur des Dokuments, da er alle Orte der darin enthaltenen Werte beschreibt [Mic14, S. 7].

Orte werden durch *Pfade* beschrieben. Ein Pfad ist eine Aneinanderreihung von *Labels*, welche mit einem *Schlüssel* versehen werden können. Das Zusammenspiel aus den Labels, den Schlüsseln und der hierarchischen Struktur macht jeden Pfad eindeutig [Mic14, S. 8]. Ein Beispiel für einen Pfad in einem Dokument, das Personen verwaltet, ist `/person[4]/name`. Hierbei sind `person` und `name` Labels, die `4` ist ein Schlüssel. Da jede Person exakt einen Namen hat, benötigt dieses Label keinen Schlüssel [Mic14, S. 9].

Existiert ein Pfad in einem Dokument, so muss auch jeder Eltern-Pfad existieren. Das heißt, das Dokument ist unter Pfad-Präfixen abgeschlossen und ein Baum [Mic14, S. 11].

Die Struktur ist XML sehr ähnlich, jedoch gibt es einen gravierenden Unterschied: Während in XML zwei gleiche Geschwisterknoten, die nicht mit einer `id` versehen sind, nur durch die Reihenfolge ihres Auftretens im Dokument unterschieden werden können, ist in XCend der Schlüssel bei Knoten, die gleichartige Geschwister haben können, zwingend erforderlich und sichert die Eindeutigkeit eines Pfades (vgl.

[Mic14, S. 8]). Das XCend-Dokument hat als Abbildung definitionsgemäß keine Ordnung. Die Schlüssel stammen aus einem unendlichen Datenbereich, der nicht geordnet sein muss. Außerdem können mehrere Schlüssel nicht zu einem neuen Schlüssel verrechnet werden [Mic14, S. 9]. Im Rahmen dieser Arbeit kommen natürliche Zahlen als Schlüssel zum Einsatz.

Da Schlüssel auch als Werte verwendet werden dürfen, können sie dazu benutzt werden, um Beziehungen zwischen Orten im Dokument auszudrücken. Allerdings ist ein Schlüssel an sich keine Referenz, da er nicht global eindeutig sein muss. Die Eindeutigkeit ergibt sich erst aus dem Pfad, bei dem sowohl die Labels als auch die Schlüssel der Elternelemente betrachtet werden müssen. Ein Schlüssel kann auch dazu benutzt werden, um an einer anderen Stelle im Dokument einen Wert auszuwählen [Mic14, S. 10].

2.2. Integritätsbedingungen und Prozeduren

XCends Hauptnutzen ist die Möglichkeit, *Integritätsbedingungen* zu definieren. Das sind logische Formeln, die unter Verwendung der eingeführten Pfade Aussagen über das Dokument treffen. Sie definieren Bedingungen, die jedes gültige Dokument erfüllen muss. Zur Spezifikation von Integritätsbedingungen definiert XCend eine eigene Zusicherungssprache, die speziell auf Dokumente zugeschnitten ist [Mic14, S. 2].

Um Konsistenz zu garantierten werden die kleinstmöglichen Änderungen (Einfügen, Aktualisieren und Löschen) als Transaktionen zusammengefasst. Nach der Transaktion müssen alle Integritätsbedingungen erfüllt sein. Während in vielen relationalen Datenbanksystemen dieser Ansatz insofern verfolgt wird, dass beim Abschluss einer Transaktion alle Fremdschlüsselbeziehungen und andere Constraints geprüft werden um die Transaktion bei Verletzung einer Bedingung zurückzurollen, definiert man in XCend vorab alle möglichen Transaktionen als Prozeduren und ermittelt durch statische Analysen die schwächste Vorbedingung, unter der eine Prozedur ausgeführt werden darf. Diese Analyse wird durch das Framework bereitgestellt, sodass der Anwender die Vorbedingungen nicht selbst herleiten muss. Allerdings müssen dazu zur Designzeit alle Schreiboperationen klar als Prozedur spezifiziert werden. Die ermittelte Vorbedingung wird entweder von der implementierenden Programmiersprache zur Laufzeit geprüft oder eine weitere statische Prüfung beweist, dass sie jederzeit beim Prozeduraufruf erfüllt ist, sodass die Prüfung entfallen kann [Mic14, S. 15].

Alle Schreiboperationen auf das Dokument erfolgen über diese Prozeduren, ein direkter Schreibzugriff auf einen Pfad ist nicht zulässig. Dies entspricht einem

relationalen Datenbankschema, bei dem alle Schreiboperationen durch gespeicherte Prozeduren (engl. *stored procedures*) realisiert werden und auf direkte Schreibzugriffe verzichtet wird.

2.3. Fallbeispiel: Softech Achievement Tracking System

Das Beispielsystem[2], für das im Rahmen dieser Arbeit eine Implementierung erarbeitet wurde, ist ein universitäres Informationssystem der AG Softwaretechnik[3].

Das System dient zur Verwaltung von Übungen und Prüfungen. Jede Übung gehört dabei zu einer Vorlesung. Ein Assistent unterstützt den Dozenten bei der Vorlesung und ist insbesondere für den Übungsbetrieb verantwortlich. Jeder Übung ist daher mindestens ein Assistent zugeordnet. In einer Übung existieren mehrere Gruppen. Jede Gruppe wird von einem (meist studentischen) Tutor betreut.

Studenten können sich nach der Registrierung in dem System für Übungen anmelden und in eine der Übungsgruppen eintragen. Während des Semesters müssen sie in Kleingruppen Aufgaben lösen und zusammen abgeben, was kontinuierlich bepunktet wird. Die erhaltenen Punkte können von den Studenten eingesehen werden und sind relevant für die Prüfungszulassung.

Das System dient auch zur Verwaltung von Klausuren. Zu jeder Klausur gibt es mindestens einen Prüfer, der für die Organisation verantwortlich ist. Studenten sehen Ort und Zeit der Prüfung und können sich anmelden. Dem Prüfer steht eine Liste der angemeldeten Studenten zur Verfügung. Nach der Korrektur der Prüfung trägt der Prüfer die erreichten Punkte der Studenten ein. Dabei werden die Punkte je Aufgabe gespeichert. Außerdem legt der Prüfer die Punktegrenzen für die Noten fest. Unterschiedliche Statistiken helfen, die Klausur zu evaluieren. Nach der Freigabe der Ergebnisse durch den Prüfer kann jeder Student seine Punkte und die erreichte Note sowie eine Statistik mit der Häufigkeit der einzelnen Noten einsehen. Den Tutoren steht zudem ein Vergleich der Ergebnisse aus der eigenen Gruppe mit dem Gesamtdurchschnitt zur Verfügung.

Im Folgenden wird genauer auf die einzelnen Personengruppen, die mit dem System interagieren, eingegangen:

[2] https://softech.cs.uni-kl.de/stats/
[3] https://softech.cs.uni-kl.de

Student

Ein Student hört Vorlesungen und nimmt an den dazugehörigen Übungen teil. Am Ende des Semesters nimmt er an der Abschlussprüfung teil und erfährt nach der Korrektur sein Ergebnis.

Bei der ersten Benutzung muss der Student sich im System registrieren. Dazu gibt er seinen Vor- und Zunamen, die Matrikelnummer, seine Mailadresse sowie einen Anmeldenamen und ein Passwort an. Das System stellt dabei sicher, dass die Matrikelnummer und der Anmeldename bislang noch nicht verwendet wurden und schickt dem Studenten eine Mail mit einem Bestätigungslink zu Verifizierung der Mailadresse. Durch einen Klick auf den Link ist die Registrierung abgeschlossen.

Sollte der Student (oder ein anderer Benutzer) seine Anmeldedaten vergessen, kann er durch Eingabe seiner Mailadresse eine Mail mit dem Anmeldenamen und einem Link zum Zurücksetzen des Passwortes anfordern.

Nach dem Login sieht der Student die Liste der aktuellen Übungen und kann sich in die Übung eintragen. Eingetragene Studenten sehen die Übungsgruppen, die für ihre Übung angeboten werden. Dabei erscheinen Ort und Zeit sowie der Name des Tutors, der diese Gruppe betreut. Sobald die Gruppenanmeldung freigeschaltet wurde, kann sich der Student in eine Gruppe eintragen. Die Gruppen haben eine maximale Kapazität. Die Größe der Gruppe und die Anzahl der bereits eingetragenen Studenten wird den Übungsteilnehmern angezeigt. Ist eine Gruppe voll, so sind keine weiteren Anmeldungen möglich. Solange die Gruppenanmeldung geöffnet ist, kann der Student sich auch wieder austragen und ggf. in eine andere Gruppe eintragen.

Während des Semesters kann der Student seine aktuellen Leistungen jederzeit kontrollieren: Er sieht seine erreichte Punktzahl sowie die Maximalpunktzahl aller Übungsblätter.

Möchte der Student an eine der innerhalb des Systems verwalteten Klausuren teilnehmen, so wählt er die entsprechende Klausur aus und kann sich während der Anmeldefrist nach Belieben an- und abmelden. Gegebenenfalls wird die Anmeldung aber auch außerhalb des Systems über das Prüfungsamt durchgeführt und die angemeldeten Studenten werden vom Prüfer eingetragen, sodass sie innerhalb des Systems ihre Anmeldung nochmals kontrollieren können. Nach Veröffentlichung der Ergebnisse kann der Student eine Liste mit den Aufgaben, seiner eigenen Punktzahl sowie der Maximalpunktzahl und die daraus resultierende erreichte Punktzahl in der Klausur einsehen. Darüber hinaus werden die Punktegrenzen und die eigene Note angezeigt. Eine Grafik zeigt die Häufigkeit der einzelnen Noten und hilft

dem Studenten, seine Leistung im Vergleich zu den anderen besser einordnen zu können.

Assistent

Ein Assistent ist für die Organisation seiner Übung zuständig. Er verwaltet die Übungsgruppen, öffnet und schließt die Gruppenanmeldung und setzt Tutoren für die Gruppen ein. Der Assistent kann jederzeit Studenten in die Gruppen ein- und austragen, selbst wenn die Anmeldung geschlossen ist. Eine Liste aller Studenten in der Übung und je Gruppe mit Namen, Mailadresse und Matrikelnummer unterstützen den Assistenten bei seinen organisatorischen Arbeiten. Ein Link öffnet automatisch eine neue E-Mail, bei der alle Studenten als BCC[4] eingetragen sind und ermöglicht so das Kontaktieren aller Übungsteilnehmer für wichtige Bekanntmachungen.

Neben den Gruppen verwaltet der Assistent auch die Übungsblätter: Zu jedem Blatt wird eine Nummer und die Gesamtpunktzahl eingetragen. Zudem stehen dem Assistenten alle Funktionen der Tutoren für die Gruppen der eigenen Übung zur Verfügung, sodass er deren Aufgabe im Krankheitsfall übernehmen kann. Alternativ kann ein zweiter Tutor für eine Gruppe eingesetzt werden.

Tutor

Ein (meist studentischer) Tutor betreut eine Übungsgruppe innerhalb einer Übung. Er kann die Namen und Mailadressen seiner Gruppenmitglieder einsehen, nicht jedoch die Matrikelnummern. Auch ihm steht ein Link für eine Mail an die Studenten zur Verfügung, sie wird jedoch nur an die eigenen Gruppenmitglieder gesandt.

Der Tutor verwaltet die Abgabegruppen innerhalb seiner Gruppe. Dazu weist er jedem Studenten eine Team-Id zu. Nach Bepunktung der Abgaben trägt er die erreichte Punktzahl in die vom Assistenten erstellten Übungsblätter ein. Dabei wird vom System sichergestellt, dass immer ein Wert zwischen 0 und der Maximalpunktzahl angegeben wird.

Prüfer

Ein Prüfer ist für die Organisation und Durchführung einer Klausur verantwortlich. Er legt Ort und Zeit der Klausur fest und kann die Anmeldung zur Klausur

[4] Blindkopie - damit sieht niemand die eigentlichen Empfänger der Mail

öffnen und schließen. Ihm steht eine Liste der angemeldeten Studenten mit Namen, Matrikelnummer und Mailadresse zur Verfügung. Außerdem kann er Studenten selbst für die Klausur anmelden, auch wenn die Anmeldung für die Studenten geschlossen ist.

Der Prüfer trägt im System die in der Klausur geprüften Aufgaben (Nummer und Bezeichnung) sowie die jeweilige Höchstpunktzahl ein. Nach der Korrektur gibt er für jeden Studenten die erreichten Punkte in den jeweiligen Aufgaben ein, wobei auch hier sichergestellt wird, dass die Höchstpunktzahl nicht überschritten werden kann.

Umfangreiche Statistiken wie zum Beispiel eine Tabelle, die zu jeder Aufgabe die Häufigkeit der einzelnen Punktzahlen zeigt oder ein Diagramm, das die Häufigkeit über die Gesamtpunktzahlen aufträgt, helfen bei der Evaluierung und beim Festlegen der Punktegrenzen. Der Prüfer gibt die Noten mit der jeweiligen Mindestpunktzahl ein und erhält daraufhin auch eine Notenverteilung. Ist alles fertiggestellt und überprüft gibt er die Ergebnisse für die Studenten frei.

Administrator

Der Administrator kann neue Nicht-Studenten Benutzeraccounts erstellen, welche keine Matrikelnummer haben, und beliebige Accounts als Assistent oder Prüfer einsetzen und weitere Administratoren ernennen sowie diese Berechtigungen auch widerrufen. Außerdem ist er für das Erstellen neuer Übungen und Klausuren verantwortlich. Er kann alle Accounts einsehen und den Anmeldenamen, Namen und das Passwort aller Benutzer modifizieren sowie Accounts löschen.

Besonderheiten dieses Systems

Das System ist von einer hohen Dynamik der Daten geprägt, denn jedes Semester gibt es neue Veranstaltungen. Die Rollen-Rechte sind jedoch klar definiert und ändern sich nicht. Das System verwaltet personenbezogene Daten, wobei vor allem die Matrikelnummern und Prüfungsergebnisse besonders schützenswert sind und angemessene Mechanismen den Schutz dieser Daten sicherstellen müssen.

3. Erweiterung von XCend um eine Sicherheitsschicht

Ziel ist es, die Beschreibungssprache von XCend so zu erweitern, dass parametrisierte Benutzerrollen definiert werden können. Dabei soll eine Vererbungsstruktur von Rechten zwischen den Rollen möglich sein. Zudem soll eine Autorisierung von Lese- und Schreibzugriffen basierend auf diesen Rollen spezifiziert werden können.

Diese Spracherweiterung soll anschließend in einer zusätzlichen Sicherheitsschicht auf die bisherige Implementierung aufsetzen und die Berechtigungen entsprechend prüfen.

3.1. Spracherweiterung der Definitionssprache

Die in diesem Kapitel verwendete Darstellung der Syntax ist eine vereinfachte Form der *Erweiterten Backus-Naur Form*, bei der wie im Standard [ISO96] Alternativen durch den senkrechten Strich sowie Optionen durch eckige, optionale Wiederholungen durch geschweifte und Gruppierungen durch runde Klammern dargestellt werden, aber auf das Komma für Konkatenation und das abschließende Semikolon verzichtet wird. Nichtterminale sind zur besseren Erkennbarkeit zudem immer in spitze Klammern eingeschlossen. Abbildung 1 zeigt ein Beispiel, in dem eine Bachelorarbeit bestehend aus einem Abstract, Inhaltsverzeichnis, Hauptteil und Anhang definiert wird. Der Hauptteil ist untergliedert in mindestens ein Kapitel. Ein Kapitel kann aus einem reinen Text bestehen oder einem optinalen Text gefolgt von mindestens zwei Unterkapiteln.

⟨*Bachelorarbeit*⟩ ::= ⟨*Abstract*⟩ ⟨*Inhaltsverzeichnis*⟩ ⟨*Hauptteil*⟩ ⟨*Anhang*⟩

⟨*Hauptteil*⟩ ::= ⟨*Kapitel*⟩ { ⟨*Kapitel*⟩ }

⟨*Kapitel*⟩ ::= ⟨*Text*⟩ | ([⟨*Text*⟩] ⟨*Unterkapitel*⟩ ⟨*Unterkapitel*⟩ {⟨*Unterkapitel*⟩})

Abbildung 1: Beispiel für eine EBNF in der genannten Syntax

3.1.1. Konzept: Leserechte & Prozeduren

Konzeptionell gilt es, zwei verschiedene Arten von Zugriffen zu unterscheiden: Zum einen gibt es *Lesezugriffe*, die den Zustand des Systems nicht verändern, und zum

anderen *schreibende Zugriffe*, die den Zustand modifizieren können. Das Design von XCend sieht vor, dass keine direkten Schreibzugriffe auf die Daten getätigt werden. Um die definierten Integritätsbedingungen auf effiziente Art und Weise sicherstellen zu können, werden Schreibzugriffe in Prozeduren gekapselt, zu denen durch statische Analysen die schwächste Vorbedingung ermittelt wird. Ist sie erfüllt und wird anschließend die Prozedur ausgeführt, so sind danach alle Integritätsbedingungen erfüllt. Lesende Zugriffe hingegen können keine Integritätsbedingungen verletzen, sodass hier keine gesonderte Prüfung und Kapselung erforderlich ist.

Das Konzept, Schreibzugriffe in Prozeduren zu kapseln, bringt aber auch aus der Sicht der Rechteverwaltung seine Vorteile: In der Regel möchte man einem Benutzer nicht die Blanko-Erlaubnis geben, an einer bestimmten Stelle in der Datenhierarchie etwas nach Belieben zu verändern. Oftmals sind nur bestimmte Arten von Änderungen zulässig oder die neuen Daten müssen bestimmte Bedingungen erfüllen. Beispielsweise kann ein Kontoinhaber den Kontostand eines anderen Kontos erhöhen, sofern er seinen eigenen Kontostand um den gleichen Betrag verringert, also eine Überweisung tätigt. Zudem könnte hier die Einschränkung herrschen, dass der eigene Kontostand dadurch nicht negativ werden darf. Dies muss jedoch nicht automatisch eine Integritätsbedingung sein, denn die eigene Bank könnte trotzdem eine Abbuchung vornehmen und damit das Konto ins Minus ziehen.

Das Konzept der Vorbedingungen in XCend lässt sich dazu benutzen, Schreibzugriffe an bestimmte Berechtigungen zu knüpfen. Dazu fügt man der Prozedur einen Parameter für die Kennung des ausführenden Benutzers hinzu und ergänzt geeignete Zusicherungen zu dieser Benutzerkennung. Dies führt allerdings dazu, dass die eigentlichen Vorbedingungen der Prozedur und die Zugriffsberechtigungen miteinander vermischt werden, sodass nicht mehr klar erkennbar ist, welche Benutzer eine Prozedur ausführen können. Ziel der Erweiterung ist daher unter anderem, diese Berechtigungen in eine eigene Beschreibung auszulagern.

Für lesende Zugriffe hingegen bietet XCend keine Möglichkeit der Rechtekontrolle an. Hier soll die Beschreibungssprache so erweitert werden, dass man für jeden Pfad in der Datenhierarchie genau festlegen kann, welche Rollen lesenden Zugriff erhalten sollen.

Wann immer man es mit mehreren Datensätzen zu tun hat kann es auch interessant sein, bestimmte abgeleitete Werte zu erstellen. Einfache Operationen sind zum Beispiel das Zählen von Datensätzen oder die Berechnung des Mittelwertes aus einer Menge von Zahlen. Dies wird in der Realität tatsächlich oft benötigt, z.B. wenn eine Notenverteilung und die Durchschnittsnote zu einer Klausur bekanntgegeben werden. Es ist durchaus denkbar, dass man jemandem aus Datenschutzgründen einen bestimmten statistischen Wert wie den Durchschnitt zugänglich machen

möchte, nicht jedoch alle einzelnen Datensätze, aus denen der Wert berechnet wurde. Um dies auch in der zu entwickelnden Sicherheitsschicht umsetzen zu können, wird das Konzept der *Leseprozeduren* eingeführt. Die Leseprozedur führt die geforderte Berechnung durch und liefert das gewünschte Ergebnis an den Aufrufer zurück, ohne die zugrunde liegenden Daten preiszugeben. Auf diese Weise kann man einer Rolle die Erlaubnis erteilen, eine Leseprozedur zu verwenden, ohne ihr die Leseberechtigung für die benötigten Daten zu geben.

Dabei darf der Programmcode innerhalb von Prozeduren (Lese- und Schreibprozeduren) auch auf Daten zugreifen, die der aufrufende Benutzer nicht direkt auslesen kann. Diese konzeptionelle Entscheidung eröffnet vorgenannte Möglichkeiten, einer Rolle beispielsweise einen Notendurchschnitt ohne Einzelnoten preiszugeben. Auch relationale Datenbanksysteme verfolgen mit gespeicherten Prozeduren diesen Ansatz: Unter der Annahme, dass der Ersteller der Prozedur die nötigen Rechte hat, kann jeder Benutzer die Prozedur verwenden, sofern er die Aufrufberechtigung hat. Für die innerhalb der Prozedur gekapselten Operationen benötigt er keine Berechtigung. Es ist jedoch wichtig, diese Tatsache beim Definieren der Prozeduren zu beachten: Ermöglicht eine Schreibprozedur es beispielsweise, Werte in der Datenhierarchie an eine andere Stelle zu kopieren, so kann ein aufrufender Benutzer, der das Ziel auslesen kann, nicht jedoch die Quelle, mit Hilfe dieser Prozedur Zugriff auf die eigentlich nicht für ihn sichtbaren Daten erlangen. Außerdem sollte man sicherstellen, dass die Leseprozeduren eine ausreichende Verschleierung der Daten durchführen, um nicht ungewollt doch die Originaldaten preiszugeben. Insbesondere sollte bei einer Durchschnittsberechnung die Kardinalität der Quelldatenmenge größer als Eins sein. In Kapitel 3.1.3 wird darauf eingegangen, wie dies definiert werden kann.

Während die Kapselung von Schreiboperationen in Prozeduren zur Sicherstellung der Integrität erforderlich ist, benutzt die Erweiterung das Konzept von Lese- und Schreibprozeduren als Hilfsmittel zur Autorisierung.

3.1.2. Rollendefinition

Das Kernstück der Spracherweiterung sind die Rollendefinitionen. Eine konkrete Rolle wird bestimmt durch ihren Namen und den Werten ihrer Parameter, wobei im Schema über die Parameterwerte abstrahiert wird. So wird eine reale Rolle *Tutor für Übung SE-1, Gruppe 7* beschrieben durch eine Rolle *Tutor* mit den Parametern *Übung* und *Gruppe*. Diese Abstraktion über die Parameter ermöglicht es, Rollen in Verbindung mit Objekten aus dem Datenschema des Systems zu verknüpfen, sodass für jedes vorhandene Objekt eine entsprechende Rolle existiert. Das Anlegen

einer neuen Übungsgruppe führt automatisch eine entsprechende Tutorrolle ein, ohne dass diese explizit im System verwaltet wird.

Da sich somit die tatsächlich vorhandenen Rollen mit dem Datenschema verändern wäre ein klassisches Berechtigungssystem, bei dem einem Benutzer innerhalb der Rechteverwaltung Rollen zugewiesen werden, eher unpraktikabel. XCend macht keine Vorgaben, wie Benutzer verwaltet werden. Vielmehr ist es sinnvoll, auch die Zuordnung der Benutzer zu den Rollen im Datenschema zu modellieren. Die Tatsache, dass ein Benutzer des Systems Tutor für eine bestimmte Übungsgruppe ist, ist keine allein der Berechtigung dienliche Information: Andere Benutzer, insbesondere Studenten der Übung, könnten am Namen des Tutors der eigenen Übung interessiert sein.

Der zu implementierenden Sicherheitsschicht ist jederzeit der derzeit angemeldete Benutzer bekannt, der Schlüssel des entsprechenden Elements aus dem Datenbaum steht in der impliziten Variablen `uid` (engl. Abkürzung für *User Identifier*) zur Verfügung. Die Beschreibungssprache muss eine Verbindung der angemeldeten `uid` zu allen Rollen dieses Benutzers ermöglichen. Dazu sollen logische Formeln der von XCend eingeführten Zusicherungssprache verwendet werden: In jeder Rolle werden Bedingungen vermerkt. Sind alle erfüllt, so hat der Benutzer diese Rolle. Die Bedingungen können dabei neben der Variablen `uid` auch alle Parameter-Variablen der definierten parametrisierten Rolle verwenden. Dies löst zugleich das Problem, dass die verschiedenen Rolleninkarnationen zum Zeitpunkt der Definition nicht bekannt sind, da keine explizite Zuordnung von Benutzern zu ihren Rollen verwaltet werden muss, sondern das Innehaben einer Rolle zur Laufzeit anhand der Bedingungen geprüft werden kann.

Ein weiterer Aspekt ist die Vererbungshierarchie zwischen den Rollen. Eine derartiges Übertragen von Rechten vereinfacht die Administration. Beispielsweise könnte man in einer Firma definieren, dass die Rolle *Vorstandsmitglied* alle Rechte der Rollen *Abteilungsleiter für Abteilung X* erben soll, wobei X der Parameter für die Abteilung ist. So ist sichergestellt, dass kein Abteilungsleiter mehr Rechte hat, als ein Vorstandsmitglied - oder anders ausgedrückt: Man muss die Berechtigungen der Abteilungsleiter-Rolle nicht nochmals in der Vorstandsmitglied-Rolle setzen.

Konzeptionell gibt es hier zwei verschiedene Lösungsansätze, wie man eine derartige Vererbung modellieren kann:

1. Man benennt in jeder Rolle, von welchen anderen Rollen die Rechte geerbt werden sollen.

2. In jeder Rolle wird definiert, an welche anderen Rollen die Rechte dieser Rolle übertragen werden sollen.

Beide Varianten haben sowohl Vor- als auch Nachteile. Der zweite Ansatz ermöglicht es, sehr schnell zu einer gegebenen Rolle R alle Gruppen von Bedingungen zu finden, bei deren Erfüllung die Rechte der Rolle R erteilt werden. Dazu betrachtet man zuerst die Bedingungen der Rolle R und dann die aller Rollen, an die die Rechte von R übertragen werden. Rekursiv führt man dieses Vorgehen auch für diese Rollen aus. Nicht auf den ersten Blick sichtbar ist hingegen die Aggregation aller Berechtigungen, die mit einer Rolle R verknüpft sind. Man muss alle anderen Rollen danach durchsuchen, ob sie ihre Rechte an die Rolle R übertragen. Bei der ersten Variante verhält es sich umgekehrt: Um die Aggregation aller Berechtigungen einer Rolle R zu erhalten betrachtet man zunächst die Berechtigungen der Rolle R und ergänzt rekursiv alle Berechtigungen der mit eingeschlossenen Rollen. Die andere Fragestellung erfordert hier wiederum das Durchsuchen aller Rollen.

Da eines der Ziele war, die Berechtigungen klarer herauszustellen und Variante 1 in dieser Hinsicht überlegen ist, wurde diese Modellierung gewählt. Die Rollen werden aufgeführt durch ihren Namen und deren Parameter. Die Parameter der angegebenen Rolle können anschließend wie in Abbildung 2 dargestellt mit Bedingungen versehen werden, beispielsweise die Einschränkung, dass ein Parameter der anderen Rolle dem der eigenen Rolle übereinstimmt. Zur Vermeidung von Namenskonflikten müssen die als Parameter angegebenen Bezeichner disjunkt zu den formalen Parametern der eigenen Rolle sein. In den Bedingungen dürfen die Parameter der eigenen Rolle, die der eingeschlossenen Rolle und der implizite Parameter `uid` verwendet werden.

⟨*include-rule*⟩ ::= ‘`include`’ ⟨*role-name*⟩ ‘`(`’ [⟨*role-parameters*⟩] ‘`)`’
‘{’ { ⟨*assertion*⟩ } ‘}’

⟨*role-parameters*⟩ ::= ⟨*role-parameter*⟩ {‘`,`’ ⟨*role-parameter*⟩ }

Abbildung 2: Definition von weiteren Rollen, deren Berechtigungen geerbt werden

Neben den Bedingungen, die das Innehaben einer Rolle spezifizieren und den Regeln, die Rechte anderer Rollen importieren, sind natürlich auch die eigentlichen Berechtigungen der Rolle zu modellieren. Im System unterscheiden wir zwischen Leserechten für direkte Zugriffe auf den Datenbaum und der Berechtigung, eine Prozedur aufzurufen. Prozeduren können dabei sowohl die bereits vorhandenen modifizierenden Prozeduren als auch die neu eingeführten Leseprozeduren sein. Der Aufbau einer Rollendefinition wird im Folgenden schrittweise erläutert, am Ende dieses Unterkapitels zeigt Abbildung 5 die vollständige Syntax.

Für die Prozeduren bietet es sich an, alle Prozeduren, die die Benutzer mit dieser

Rolle aufrufen dürfen, aufzulisten. Allerdings möchte man nicht immer Aufrufe für beliebige Parameter erlauben, ggf. sollen hier je nach Rolle gewisse Einschränkungen herrschen. Dies lässt sich mit Bedingungen, die neben der `uid` und den Parametern der Rolle auch die Parameter der Prozedur verwenden dürfen gut beschreiben. Sind alle Bedingungen erfüllt, so ist ein Prozeduraufruf mit diesen Rollen- und Prozedurparametern gestattet. Abbildung 3 zeigt die Syntax für Definition einer Prozeduraufrufsberechtigung. Um eine Kollision der Variablennamen in den Bedingungen zu vermeiden, müssen die Parameternamen in den runden Klammern disjunkt zu den Rollenparametern gewählt werden und dürfen `uid` nicht enthalten. Sie müssen nicht die gleichen Namen tragen wie in der Definition der Prozedur, lediglich die Anzahl und die Typen der Parameter muss übereinstimmen.

⟨*procedure-permission*⟩ ::= ‘`call`’ ⟨*procedure-name*⟩
‘`(`’ ⟨*procedure-parameters*⟩ ‘`)`’ ‘{’ { ⟨*assertion*⟩ } ‘}’

Abbildung 3: Definition einer Prozeduraufrufsberechtigung

Die Rechteverwaltung für direkte Lesezugriffe auf den Datenbaum wird mit der Erweiterung erstmals ermöglicht. Dazu soll der zu lesende Ort durch einen Pfadausdruck beschrieben werden. Der Ausdruck darf Labels, die oben bereits eingeführte Variable `uid` und die Parameter-Variablen der parametrisierten Rolle benutzen. Eine reine Auflistung solcher Pfade wäre allerdings nicht mächtig genug: Zusätzlich sollen neue Schlüssel-Variablen im Pfadausdruck definiert werden können und anschließend mit Bedingungen versehen werden, siehe Abbildung 4. Die in runden Klammern angegebenen Parameter sollen dabei disjunkt zu den Rollenparametern sein und auch die Variable `uid` nicht enthalten. In den Bedingungen können beide Parametertypen und `uid` verwendet werden. Zum Zeitpunkt des Lesezugriffs stehen alle Schlüssel auf dem Pfad von der Wurzel zum zu lesenden Knoten bereit, sodass diese Zusicherungen dann geprüft werden können. Sind alle Zusicherungen erfüllt, so wird der Lesezugriff gestattet. Auf diese Weise lässt sich ein Lesezugriff auf eine Stelle im Baum auch abhängig von Daten beschreiben, die nicht auf dem Pfad zum zu lesenden Knoten liegen, beispielsweise das Leserecht eines Prüfers auf die Übungsergebnisse seiner Klausurteilnehmer in Listing 4.

⟨*read-permission*⟩ ::= ‘`read`’ [‘`(`’ ⟨*parameter*⟩ { ‘`,`’ ⟨*parameter*⟩ } ‘`)`’]
⟨*pathexpression*⟩ ‘{’ { ⟨*assertion*⟩ } ‘}’

Abbildung 4: Definition einer Leseberechtigung (Syntax)

```
read(exerciseKey) /stats/exercise[exerciseKey]/student/result {
  assert exerciseKey = /stats/exam[exam]/exercise
}
```

Listing 4: Definition einer Leseberechtigung (Beispiel)

Eine komplette Rollendefinition hat folgende Syntax:

⟨*roledef*⟩ ::= 'role' ⟨*role-name*⟩ ['(' ⟨*role-parameter*⟩ {',' ⟨*role-parameter*⟩} ')'] '{' { ⟨*assertion*⟩ } {⟨*include-rule*⟩}
{ ⟨*procedure-permission*⟩ | ⟨*read-permission*⟩ } '}'

Abbildung 5: Rollendefinition

3.1.3. Leseprozeduren

Wie in Kapitel 3.1.1 bereits eingeführt sollen Leseprozeduren es ermöglichen, den Zugriff auf abgeleitete Werte zu gewähren, ohne dabei das Auslesen der zur Berechnung nötigen Datenbasis zu erlauben. Dafür wird diese Berechnung in eine Prozedur gekapselt.

Die alternative Lösung, zu berechnenden Werte dauerhaft redundant vorzuhalten, um mit reinen Leseberechtigungen auszukommen, bringt mehrere Nachteile mit: Die redundante Speicherung vergrößert das Datenschema und macht es damit unübersichtlicher. Zudem muss bei jeder Änderungsoperation der abgeleitete Wert entsprechend aktualisiert werden. XCend bietet nicht genügend Operationen für diese Aktualisierungen an, um alle denkbaren Berechnungen umzusetzen, die mit den hier definierten Leseprozeduren möglich werden. Eine Erweiterung wäre also auch hier nötig, sodass die Leseprozeduren klar im Vorteil sind.

Um die Komplexität der Erweiterung möglichst gering zu halten machen im Wesentlichen nur zwei Sachen die Mächtigkeit der Leseprozeduren aus: Zum einen erlauben lokale Variablen es, Zahlenwerte zwischenzuspeichern und zum anderen kann eine `foreach`-Schleife ausgehend von einem Knoten im Datenbaum über die Schlüssel aller Kind-Knoten iterieren.

Bei den Berechnungen soll sich dabei auf unbeschränkt große ganze Zahlen, die auch bereits als Typ im Datenschema auftauchen, und auf Gleitkommazahlen doppelter Genauigkeit ([Ins08]) beschränkt werden. Dies ist eine konzeptionelle Entscheidung. Die Beschränkung auf Zahlenwerte, für die es klar definierte arithmetische Operationen gibt, vereinfacht zum Einen die Implementierung eines Generators, der

⟨*read-procedure*⟩ ::= ⟨*rptype*⟩ ⟨*procedure-name*⟩ '`(`' ⟨*procedure-parameters*⟩ '`)`' '{'
{⟨*declaration*⟩} {⟨*rpstatement*⟩} '}'

⟨*rptype*⟩ ::= '`int`' | '`double`'

⟨*declaration*⟩ ::= ⟨*rptype*⟩ ⟨*variable-name*⟩ ['=' ⟨*number*⟩]

⟨*rpstatement*⟩ ::= ⟨*assignment*⟩ | ⟨*foreach*⟩ | ⟨*assertion*⟩ | ⟨*conditional*⟩ | ⟨*return*⟩

⟨*assignment*⟩ ::= ⟨*variable-name*⟩ '=' ⟨*value-expression*⟩

⟨*foreach*⟩ ::= '`foreach`' ⟨*multiset-path-expression*⟩ '`do`' {⟨*rpstatement*⟩} '`done`'

⟨*conditional*⟩ ::= '`if`' ⟨*boolean-expression*⟩ '`then`' {⟨*rpstatement*⟩}
['`else`' {⟨*rpstatement*⟩}] '`fi`'

⟨*boolean-expression*⟩ ::= (⟨*boolean-expression*⟩ ⟨*bin-boolean-op*⟩
⟨*boolean-expression*⟩) | ('`!`' ⟨*boolean-expression*⟩) |
(⟨*value-expression*⟩ ⟨*comparison-op*⟩ ⟨*value-expression*⟩)

⟨*bin-boolean-op*⟩ ::= '`&&`' | '`||`'

⟨*comparison-op*⟩ ::= '<' | '>' | '<=' | '>=' | '`!=`' | '=='

⟨*value-expression*⟩ ::= ⟨*number*⟩ | ⟨*path-expression*⟩ | ⟨*variable-name*⟩
| ('`(`' ⟨*value-expression*⟩ ⟨*arithmetic-op*⟩ ⟨*value-expression*⟩ '`)`')

⟨*arithmetic-op*⟩ ::= '+' | '−' | '*' | '/'

⟨*return*⟩ ::= '`return`' ⟨*value-expression*⟩

Abbildung 6: Syntax einer Leseprozedur

aus der Beschreibung in der Schemasprache Javacode erzeugen kann, und zum anderen gibt es für das Ableiten von Werten aus Zeichenketten (Strings) unter dem Hintergrund der Zugriffskontrolle kaum sinnvolle Anwendungsmöglichkeiten. Zudem sind Schlüssel per Definition nicht miteinander verrechenbar. Dementsprechend sind ganze Zahlen (`int`) und Gleitkommazahlen (`double`) auch die möglichen Typen für lokale Variablen und für den Rückgabewert der Leseprozedur. Abbildung 6 veranschaulicht die Syntax einer vollständigen Leseprozedur.

Die verwendeten lokalen Variablen werden direkt zu Beginn der Prozedur deklariert. Dabei wird der Typ der Variablen, der Name der Variablen und ggf. ein Startwert

angegeben. Wird kein Startwert definiert, so wird die Variable implizit mit 0 initialisiert. Prozedurparameter und lokale Variablen müssen paarweise verschiedene Namen haben, zudem ist das Dollarzeichen als Präfix nicht gestattet, da es für quantifizierte Variablen innerhalb von `foreach`-Schleifen reserviert ist.

Als Anweisungen in Leseprozeduren sind Zuweisungen von Werten an lokale Variablen, die bereits genannten `foreach`-Schleifen, Zusicherungen, Fallunterscheidungen und `return`-Anweisungen zugelassen.

Als Werte gelten explizit angegebene Zahlen, Pfadausdrücke, welche im Datenschema zu einem `int` führen oder mit der `size`-Funktion einen `int` liefern, Variablennamen von lokalen Variablen sowie den Parametern der Prozedur (sofern der Typ `int` ist) und die üblichen vier arithmetischen Verknüpfungen zweier Werte. Wird das Ergebnis einer Division einer lokalen Variable vom Typ `double` zugewiesen oder als Wert in einer `return`-Anweisung verwendet, so wird die Division als `double`-Division gemäß IEE754 ([Ins08]) ausgeführt, anderenfalls als Ganzzahl-Division mit Richtung Null gerundetem Ergebnis entsprechend der Java Language Specification [GJS+, Kap. 15.17.2].

Eine `foreach`-Schleife iteriert über ein Multiset. Dazu wird ein spezieller Pfadausdruck angegeben, in dem die Schlüssel an einen bisher ungenutzten Bezeichner mit einem Dollarzeichen als Präfix gebunden werden. Bisher ungenutzt heißt in diesem Fall, dass der Bezeichner von keiner äußeren `foreach`-Schleife verwendet wird. Die Schleife iteriert durch dieses Multiset und stellt unter dem angegebenen Variablennamen den entsprechenden Wert in jedem Schleifendurchlauf zur Verfügung. Dabei kann das Multiset auch durch Quantifizierung mehrerer Variablen auf einmal gebildet werden.

In Leseprozeduren sind Zusicherungen auch nach anderen Anweisungen erlaubt, während die Vorbedingungen bei schreibenden Prozeduren vor allen anderen Anweisungen geprüft werden müssen. Da eine Leseprozedur per Definition keine Daten ändern kann, gibt es auch keine Inkonsistenzen bei vorzeitigem Abbruch der Prozedur - dies ist bei schreibenden Zugriffen nicht gegeben, sodass dort diese Einschränkung besteht. Eine nicht erfüllte Zusicherung bricht die Prozedur ergebnislos ab. Dies kann zum Beispiel dann sinnvoll sein, wenn ein Durchschnitt erst ab einer bestimmten Größe der Datenbasis preisgegeben werden soll. Da das Konzept der `foreach`-Schleife in Zusammenhang mit den lokalen Variablen komplexere Berechnungen als die Funktionen `size` und `count` ermöglichen, wäre eine Beschränkung auf Vorbedingungen in Leseprozeduren eine echte Einschränkung.

Die Semantik einer Fallunterscheidung entspricht der intuitiven Semantik, die auch von Java verwendet wird.

Eine `return`-Anweisung beendet die Prozedur und liefert den angegebenen Wert als Ergebnis zurück. Die letzte Anweisung einer Prozedur muss stets eine `return`-Anweisung sein oder eine Fallunterscheidung, bei der sowohl im `then`-Zweig als auch im in diesem Falle obligatorischem `else`-Zweig die letzte Anweisung ein `return` ist.

Wird innerhalb der Prozedur zur Laufzeit eine Division durch Null durchgeführt, so wird die Prozedur ergebnislos mit einem Fehler beendet, in der Java-Implementierung entsprechend mit einer `ArithmeticException`.

3.1.4. Erweiterung des Datenschemas

Obwohl eine Änderung des Datenschemas eigentlich nicht nötig ist, kann eine kleine Anpassung die Definition der Leserechte jedoch erheblich vereinfachen. Dazu sollen zwei Schlüsselwörter `private` und `public` eingeführt werden. Wie in vielen Programmiersprachen definieren diese Modifikatoren die Sichtbarkeit einzelner Elemente.

Jedes in XCend beschriebene Datenschema hat eine baumartige Struktur. Möchte ein Benutzer auf einen Knoten wie zum Beispiel `/root/sub1/sub2` zugreifen, so benötigt er eine Leseberechtigung für den gewünschten Knoten und für alle Präfixe auf dem Pfad von der Wurzel zu diesem Knoten, also `/root`, `/root/sub1` und `/root/sub1/sub2`. Diese Entscheidung beruht auf der Erkenntnis, dass ein vorhandener Pfad die Existenz aller Präfixe impliziert und man durch die Leseberechtigung somit die Existenz weiterer Pfade im Dokument ableiten könnte. Um jedem Benutzer, der auf `/root/sub1` Zugriff hat, auch den Lesezugriff auf `/root/sub1/sub2` zu gewähren, wird das Element `sub2` in `sub1` als `public` deklariert. Somit spart man sich die Definition der entsprechenden Leserechte auf `sub2`.

XCend unterscheidet bei den Knoten im Baum-Schema zwischen Elementen (`element`) und Attributen (`attribute`). Ein Attribut ist immer ein Blatt. Es speichert einen Wert eines vorgegebenen Typs: Entweder eine Zeichenkette (`string`) oder eine ganze Zahl (`int`). Ein Element hingegen speichert keinen Wert, kann jedoch ein, mehrere oder keine weiteren Elemente und Attribute enthalten.

Die Semantik der hier definierten Spracherweiterung sei so gewählt, dass Attribute standardmäßig öffentlich sind. Das heißt, dass jeder, der ein Element lesen darf, auch auf alle direkt darin enthaltenen Attribute lesend zugreifen kann. Um dies zu verhindern, muss der Modifikator `private` ergänzt werden. Elemente hingegen sind standardmäßig nicht öffentlich, man benötigt also eine explizite Leseberechtigung

für ein Kind-Element oder man deklariert das Element mit dem Schlüsselwort `public` als öffentlich. Damit ergibt sich die in Abbildung 7 angedeutete Syntax.

⟨*node*⟩ ::= ⟨*element*⟩ | ⟨*attribute*⟩

⟨*element*⟩ ::= ['`public`'] '`element`' ⟨*elementdefinition*⟩

⟨*attribute*⟩ ::= ['`private`'] '`attribute`' ⟨*attributedefinition*⟩

Abbildung 7: Erweiterung der Datenschemabeschreibung

3.1.5. Zusammenfassung der Spracherweiterung

Zusammengefasst sieht eine XCend-Schemadatei nun wie folgt aus:

⟨*xcend-schema*⟩ ::= ⟨*root-element*⟩ { ⟨*roledef*⟩ } { ⟨*procedure*⟩ }

⟨*procedure*⟩ ::= ⟨*read-procedure*⟩ | ⟨*write-procedure*⟩

Abbildung 8: Format der XCend-Schemadatei mit der Spracherweiterung

3.2. Implementierung der Sicherheitsschicht in Java

Für die in Kapitel 3.1 definierte Erweiterung der Beschreibungssprache soll am Beispiel des in Kapitel 2.3 vorgestellten STAT Systems eine Implementierung in Java erstellt werden. Das dazu verwendete Schema befindet sich im Anhang A.

XCend bietet einen Codegenerator für Java an, welcher ein sogenanntes *Binding* erzeugt, das den Zugriff auf das Dokument abstrahiert.

> „To work with the data structures defined by these schemata in programming languages, so-called data binding techniques are applied. In an object-oriented environment, these techniques usually supply a set of classes in the target language which represent the schema definitions, or allow definition of a mapping between elements within the XML schema and user-defined classes. Code is subsequently generated to map between the two representations of data. When reading from XML documents matching the schema, objects are instantiated.“[Fis12, S. 5]

In der angegebenen Quelle wird auch näher auf diesen Generator eingegangen.

3.2.1. Positionierung der Sicherheitsschicht im System

Um die Komplexität der Sicherheitsschicht möglichst gering zu halten, soll sie nicht zu sehr mit dem restlichen System interagieren müssen. Daher bietet sich eine Implementierung als eigene Schicht mit fest definierten Schnittstellen zum restlichen System an. Insbesondere soll der Code der GUI komplett unabhängig von der Sicherheitsschicht arbeiten, sodass die Zugriffsrechte auf einer tiefen Ebene überprüft werden.

Die Sicherheitsschicht wird zwischen ein bestehendes Binding und z.B. der GUI geschaltet. Das heißt, dass die Sicherheitsschicht auf einem Interface für das Binding arbeitet und sich gleichzeitig nach außen wie ein Binding verhält. Das von der Sicherheitsschicht zu implementierende Interface kann man mit dem bestehenden Codegenerator einfach erzeugen: Man entfernt aus der Datendefinition in der Schema-Datei die neu eingefügten Modifikatoren `private` und `public`, entfernt die Rollendefinitionen komplett und die Leseprozeduren. Damit generiert der Generator ein Binding, dass keinerlei Benutzerauthentifizierungen durchführt. Das im Beispielsystem zuvor angewendete Vorgehen, die Authentifizierung mit einem Prozedurparameter `uid` zu überprüfen, wurde durch die Spracherweiterung ersetzt, sodass dieser Parameter weggefallen ist. Die Implementierung im Generat wird vorerst ignoriert, lediglich das Interface dient als Grundlage für die Sicherheitsschicht, da sie selbiges implementieren muss.

Die Definition der Berechtigungen für den Aufruf von schreibenden Prozeduren lässt sich auf recht einfache Art und Weise in ein Schema ohne Erweiterungen zurückführen. Daher besteht die Möglichkeit, aus dem Schema mit Erweiterung eine im Bezug auf schreibende Prozeduren äquivalentes Schema ohne Erweiterung zu generieren. Dazu muss die Vererbungshierarchie zwischen den Rollen aufgelöst werden und die entsprechende Zusicherungen an den Kopf der Prozedurrümpfe gesetzt werden. Die Variable `uid` muss entsprechend wieder als Prozedurparameter definiert werden. Das mit dem so transformierten Schema generierte Binding kann als Grundlage für die Implementierung der Sicherheitsschicht genommen werden: Die Schreibrechte werden durch dieses Binding geprüft, lediglich die Leserechte und die Leseprozeduren müssen noch implementiert werden. Das Interface, das von der Sicherheitsschicht implementiert werden muss, unterscheidet sich jedoch vom intern verwendeten Interface: Der Parameter `uid` taucht in der Sicherheitsschicht nur einmalig auf, nicht bei jedem Prozeduraufruf. Dem Benutzer wird ein Objekt der Sicherheitsschicht zur Verfügung gestellt, das eine Schnittstelle zum echten Binding darstellt. Die `uid` ist in diesem Objekt gespeichert, sodass bei jedem Prozeduraufruf und bei jedem Lesezugriff geprüft werden kann, ob der Benutzer, dem dieses Objekt zugeordnet ist, die nötigen Rechte hat. Falls ja, wird die Anfrage (Lesezugriff bzw.

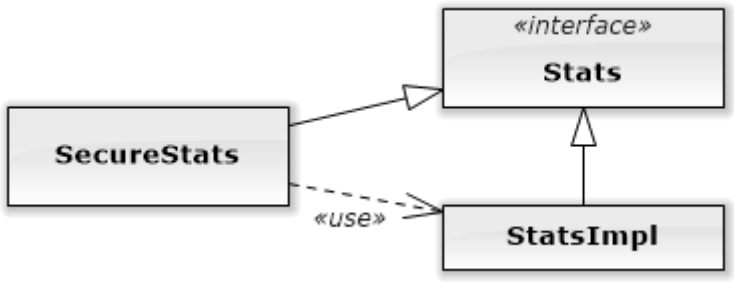

Abbildung 9: Positionierung der Sicherheitsschicht (1)

```
package de.xcend.stats;
import de.xcend.stats.StatsBinding.Stats;

public class SecureStats implements Stats {
  private final Stats realStats;
  private final Account.KeyIF uid;

  public SecureStats(Stats realStats, Account.KeyIF uid) {
    this.realStats = realStats;
    this.uid = uid;
  }
}

// Verwendung:
Stats realStats = ...;
Account.KeyIF currentUser = ...;
SecureStats stats = new SecureStats(realStats, currentUser);
```

Listing 5: Positionierung der Sicherheitsschicht

Prozeduraufruf) wie im Forwarding Pattern (siehe [PH, Kap. 2.2.3, S. 39]) an das intern verwaltete Binding weitergegeben. Die gespeicherte `uid` wird dabei von der Sicherungsschicht als zusätzlicher Parameter hinzugefügt. Da der Codegenerator allerdings ein einzelnes Interface mit hierarchisch strukturierten inneren Klassen, Interfaces und Enums liefert, passen die Typen der Parameter und der Exceptions nicht zueinander. Außerdem lässt sich durch diese verschachtelte Struktur nur schwer eine Ersetzung durchführen, um die Typen des intern verwendete Bindings an die des zu implementierenden Inferaces anzupassen.

Für die Prüfung der Leserechte ist eine entsprechende Implementierung bereitzustellen. Auf die gleiche Weise können jedoch auch die Rechte für den Prodezuraufruf überprüft werden, sodass die Einsparung durch ein internes Binding, das lediglich

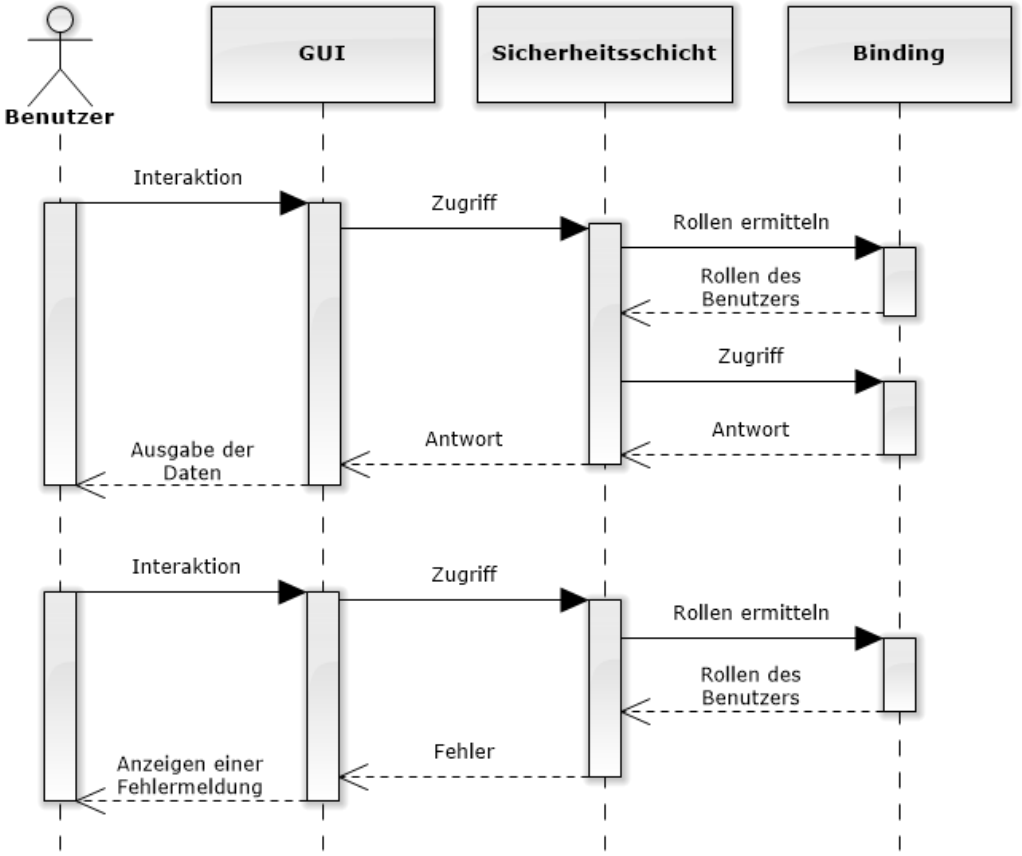

Abbildung 10: Positionierung der Sicherheitsschicht (2)

die Schreibrechte prüft, im Vergleich zu den damit verbundenen technischen Problemen der verschiedenen Interfaces eher gering ausfällt. Zudem würde die Prüfung der Zugriffsrechte bei zuvor genanntem Ansatz sich noch unter der Sicherheitsschicht befinden. Dies widerspricht jedoch dem Anspruch, diese Überprüfung in einer dedizierten Schicht zu kapseln. Daher wird folgender Ansatz verfolgt:

Nach außen implementiert die Sicherheitsschicht das zu Beginn dieses Unterkapitels vorgestellte Binding ohne den Parameter `uid`. Intern verwaltet sie das Binding aus dem gleichen Generat, also auch das selbe Interface (Abbildung 9). Im Konstruktor wird einmalig die `uid` und das echte Binding übergeben. Das Objekt ist fortan dem Benutzer mit dieser `uid` zugeordnet und prüft bei jedem Aufruf, ob er weitergeleitet werden darf. Nur wenn die Berichtigungen ausreichen, wird der Aufruf unverändert an die darunterliegende Schicht weitergegeben. Abbildung 10 veranschaulicht zwei Interaktionen eines Benutzers, bei der die erste berechtigt erfolgt und die zweite wegen eines nicht autorisierten Zugriffs fehlschlägt. Listing 5

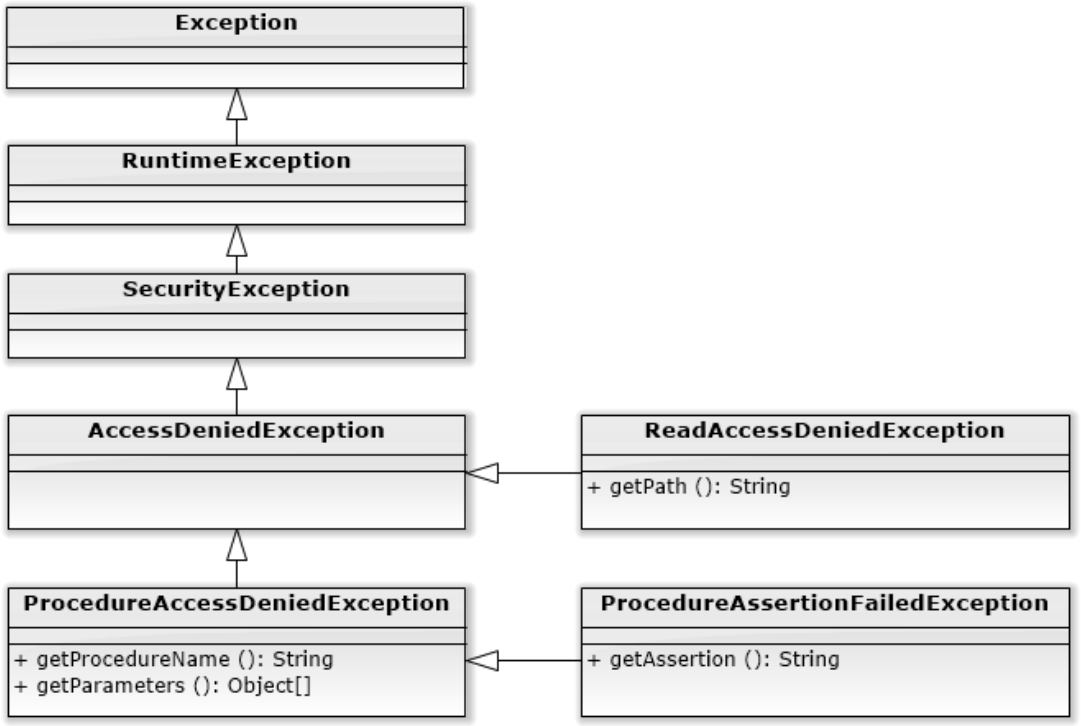

Abbildung 11: Exceptions in der Sicherheitsschicht

zeigt, wie die Sicherheitsschicht auf dem vorhandenen Binding aufbaut und dann verwendet werden kann. In der GUI-Schicht kann dann an einer Stelle die `uid` aus der Session ausgelesen werden und das echte Binding, welches das Dokument verwaltet, in der Sicherheitsschicht damit verkapselt werden. Der Rest der GUI benutzt nur noch dieses abgesicherte Objekt.

Durch die Implementierung desselben Interfaces können keine zusätzlichen Checked Exceptions geworfen werden. Für Zugriffsverletzungen ist in der Java API die Klasse `SecurityException` vorgesehen („Thrown by the security manager to indicate a security violation.“ [Ora]), welche selbst eine Unchecked Exception ist. Daher werden als Exceptions der Sicherheitsschicht Spezialisierungen von SecurityException gewählt. Diese Wahl hat zwar den Nachteil, dass der Compiler das Behandeln von Zugriffsverletzungen nicht erzwingt, jedoch sollte eine GUI sowieso keine Funktionen bereitstellen, die der aktuell angemeldete Benutzer gar nicht verwenden darf, und daher auch keinen solchen Aufruf auf der Sicherheitsschicht tätigen. Unter dieser Annahme ist das Auftreten einer Zugriffsverletzung innerhalb der Sicherheitsschicht ein Fehler innerhalb der darüberliegenden Schicht und es ist fraglich, ob eine Checked Exception diesen Fehler vermieden hätte, bzw. ob ein sinnvoller catch-

Block zum Einsatz gekommen wäre. Die Unchecked Exception ermöglicht es, den Fehler ganz nach oben durchzureichen und dort dem Anwender eine generische Fehlermeldung anzuzeigen, ohne sie explizit in jeder Prozedur innerhalb der throws-Klausel anzugeben. Sollte jedoch die Schicht über der Sicherheitsschicht keine eigene Zugriffskontrolle durchführen wollen, so kann ein Ansatz mit `try-catch` natürlich auch bei Unchecked Exceptions gewählt werden, sodass hier kein gravierender Nachteil entsteht.

Abbildung 11 zeigt die Typhierarchie der Exceptions. Die gemeinsame Superklasse `AccessDeniedException` ist beispielsweise hilfreich, wenn alle Zugriffsverletzungen gefangen werden sollen. Geworfen werden hingegen nur die Kindklassen. `ProcedureAccessDeniedException` steht für einen unberechtigten Prozeduraufruf (Lese- oder Schreibprozedur). Die Exception bietet die Möglichkeit, den Namen der aufgerufenen Prozedur sowie die verwendeten Parameter auszulesen, was für Debuggingzwecke hilfreich sein kann. Als Spezialsisierung hiervon weist die `ProcedureAssertionFailedException` auf eine fehlgeschlagene Zusicherung innerhalb einer Leseprozedur hin, was beispielsweise eine zu kleine Basismenge für eine Durchschnittsberechnung bedeuten kann. Die fehlgeschlagene Zusicherung ist in der Exception als String enthalten, sodass mit einer Fallunterscheidung eine angepasste Fehlermeldung an den Benutzer ausgegeben werden kann, beispielsweise „Unter 5 Einzelnoten kein Durchschnitt verfügbar!". Ein fehlgeschlagener direkter Lesezugriff generiert eine `ReadAccessDeniedException`. Sie enthält den Pfad zu dem Knoten, der unberechtigter Weise gelesen wurde, als String ohne Angabe der Schlüssel, also zum Beispiel `/stats/account/student`.

3.2.2. Abbildung der Rollen-Vererbungsstruktur

Konzeptionell soll jede Rollendefinition (`role`-Block im erweiterten XCend Schema) in einer Klasse abgebildet werden. Objekte dieser Klasse sind dann konkrete Rollen mit instantiierten Parametern. Dieses Objekt soll für eine gegebene `uid` prüfen können, ob der Benutzer diese Rolle hat. Dabei muss die Vererbungshierarchie der Rollen untereinander in Betracht gezogen werden. Die Hilfsklasse in Listing 6 übernimmt diese Aufgabe, sodass jede Klasse nur noch die Bedingungen der eigenen Rolle prüfen muss (`checkConstraint`).

```
abstract class Role {
  Role(Stats realStats) { ... }
  abstract boolean checkConstraint(Stats realStats,
      Account.KeyIF uid);
  final boolean hasRole(Account.KeyIF uid) { ... }
  final void addIncludedBy(Collection<Role> roles) { ... }
  final void addIncludedBy(Role... roles) { ... }
}
```

Listing 6: Hilfsklasse für Rollen

Dabei wird die Rechtevererbung zwischen den Rollen in inverser Richtung zum Schema verwaltet. Im Schema war es wichtig, einen Überblick zu haben, welche Rolle was darf. Daher ist eine Auflistung aller Rollen, deren Rechte geerbt werden sollen, im Schema sinnvoll. In der Implementierung ist jedoch von Interesse, ob ein gegebener Benutzer eine bestimmte Rolle hat. Daher wird hier die andere Richtung benötigt. Dies erfordert einmaligen Aufwand beim Erstellen des Codes, denn alle `include`-Anweisungen aller anderen Rollen müssen nach der aktuell betrachteten Rolle abgesucht werden. Diese werden dann im Konstruktor einer Rolle an die Superklasse `Role` übergeben, welche die Rollen in einem `Set` verwaltet. Dabei löst `addIncludedBy` die rekursive Vererbung von Rollen auf und die Methode `hasRole` übernimmt dann die Prüfung der Zusicherungen der eigenen Rolle sowie aller in diesem `Set` verwalteten Rollen.

Die Parameter der Rollen sind Schlüssel von Elementen im Datenschema. In der Regel beschreibt der Parameter einer Rolle ein Objekt aus dem Datenbaum, das durch seinen Schlüssel identifiziert werden kann. Möchte man eine Rolle instantiieren so gibt es verschiedene Szenarien:

1. Es soll geprüft werden, ob der aktuelle Benutzer eine Rolle mit vorgegebenen Parameter hat. Zum Beispiel „Ist der aktuelle Benutzer ein Tutor für die Übung SE-1 Gruppe 7?“

```
public interface KeyParameter<K extends KeyInterface> {
  public K getKey();
  public boolean isWildcard();
}

public class Key<K extends KeyInterface> implements KeyParameter<K> {
  private final K key;
  public Key(K key) {
    this.key = key;
  }
  public K getKey() {
    return this.key;
  }
  public boolean isWildcard() {
    return false;
  }
}

public class Wildcard<K extends KeyInterface>
      implements KeyParameter<K> {
  public K getKey() {
    throw new NullPointerException("KeyParameter is Wildcard!");
  }
  public boolean isWildcard() {
    return true;
  }
}
```

Listing 7: Klasse zur Darstellung der Rollenparameter

2. Es soll geprüft werden, ob der aktuelle Benutzer eine Rolle hat, aber mit einem beliebigen Parameter. Zum Beispiel „Ist der aktuelle Benutzer ein Tutor in der Übung SE-1, für egal welche Gruppe?“

3. Der Parameter ist undefiniert und kann daher nicht explizit angegeben werden. Zum Beispiel: Der Tutor darf die Punkte eines Studenten ändern, wenn dieser in seiner Gruppe eingetragen ist. Der aktuell betrachtete Student ist jedoch in keiner Gruppe eingetragen. Wenn jemand Tutor für alle möglichen Gruppen wäre, dann hätte er diese Berechtigung. Dies trifft aber durch die Rechtevererbung zum Beispiel auf die Rolle Assistent zu. Daher macht die Fragestellung „Ist der aktuelle Benutzer ein Tutor für alle Gruppen in der Übung SE-1?“ durchaus Sinn.

Um die ersten beiden Fälle auseinander halten zu können wird der in Listing 7 vorgestellte Typ `KeyParameter` eingeführt. Fall 1 wird durch `Key` mit entspre-

```
public class TutorRole extends Role {
  private final KeyParameter<Exercise.KeyIF> exercise;
  private final KeyParameter<Exercise.Group.KeyIF> group;

  public TutorRole(Stats realStats, KeyParameter<Exercise.KeyIF>
        exercise, KeyParameter<Exercise.Group.KeyIF> group) {
    super(realStats);
    this.exercise = exercise;
    this.group = group;
    addIncludedBy(new AssistantRole(realStats, exercise));
  }

  public TutorRole(Stats realStats,
        KeyParameter<Exercise.KeyIF> exercise) {
    super(realStats);
    this.exercise = exercise;
    this.group = null;
    addIncludedBy(new AssistantRole(realStats, exercise));
  }

  public TutorRole(Stats realStats) {
    super(realStats);
    this.exercise = null;
    this.group = null;
    addIncludedBy(new AssistantRole(realStats));
  }

  @Override
  boolean checkConstraint(Stats realStats, Account.KeyIF uid) {
    if (uid == null || this.exercise == null || this.group == null) {
      return false;
    }

    // ...
  }
}
```

Listing 8: Tutor-Rolle (gekürzt)

chendem Schlüssel und Fall 2 durch `Wildcard` realisiert. Der seltenere Fall 3 kann durch einen Nullpointer repräsentiert werden.

Listing 8 zeigt exemplarisch die Implementierung der Tutor-Rolle. Der Konstruktor kann mit allen Parametern aufgerufen werden. Je nachdem, ob als `KeyParameter` ein `Key` oder eine `Wildcard` zum Einsatz kommt, entspricht dies Fall 1 bzw. Fall 2. Durch das Weglassen der Gruppe werden die Berechtigungen nicht gewährt, da `checkConstraint` direkt false ausgibt. Allerdings könnte der Benutzer die Rolle

Assistent für die angegebene Übung haben, womit er die Berechtigung bekommen würde. Dies entspricht Fall 3. Verzichtet man auch auf die Angabe der Übung, so greift die Rolle Assistent ebenfalls nicht. Allerdings fügt diese wiederum die Rolle Admin hinzu, sodass ein Admin die Rechte eines Tutors hat, selbst wenn die angefragte Übung und Gruppe nicht existieren. Dieses Konzept macht insofern Sinn, dass es hierbei allein um Berechtigungen geht. Bindet eine Rolle (Assistent) ein andere (Tutor) ein, ohne dabei einen der Parameter (Gruppe) zu beschränken, so ist das eine übergeordnete Autorität. Falls es überhaupt nicht möglich sein soll, Punkte für Studenten einzutragen, die in keiner Gruppe sind, wäre dies als Integritätsbedingung im Datenschema oder als Zusicherung innerhalb der Prozedur anzubringen, nicht jedoch auf Basis der Berechtigungen.

Die Prüfung von `uid` auf `null` ergibt sich daraus, dass nicht angemeldete Benutzer mit einem Nullpointer als `uid` realisiert werden. Immer wenn die `uid` innerhalb der Rollen-Bedingungen verwendet wird, muss diese Überprüfung zuvor durchgeführt werden, um eine `NullPointerException` zu vermeiden.

Um nicht jeden Schlüssel immer explizit in einen `KeyParameter` verpacken zu müssen bietet es sich an, entsprechende Alias-Konstruktoren in den Rollen zu definieren, die diese Aufgabe übernehmen. Da man aber bei einem solchen Alias-Konstruktor keinen Nullpointer mehr angeben darf („The constructor [...] is ambiguous“), wird hier noch ein zusätzlicher Konstruktor ohne den entsprechenden Parameter definiert.

3.2.3. Kapselung der Objekte

Die Architektur des generierten Bindings sieht vor, dass für jeden Knoten im Datenschema ein Interface und eine Klasse existiert. Es werden Methoden angeboten, um zum Beispiel an die Kinder dieses Knotens zu kommen oder die Attribute auszulesen. Hier muss in der Sicherheitsschicht bei jedem Aufruf die Berechtigung geprüft werden. Die Methode `exam()` in `Stats` liefert beispielsweise alle Klausuren. Würde die Sicherheitsschicht nach dem erfolgreichen Prüfen der Leseberechtigung auf den Pfad `/stats/exam` einfach den Aufruf an das interne Binding weiterleiten und dieses Ergebnis unverändert zurückgeben, so könnten keine weiteren Überprüfungen durchgeführt werden, wenn beispielsweise mit der Methode `participant()` die Liste aller Klausurteilnehmer abgefragt wird.

Die Lösung hierfür besteht darin, dass für jedes dieser Interfaces eine sichere Klasse innerhalb der Sicherheitsschicht definiert wird, wobei die hierarchische Strukturierung des generierten Bindings beibehalten wird.

```
public static class SecureAccount implements Stats.Account {
  private final Stats realStats;
  private final Account.KeyIF uid;
  private final SecureStats secureStats;
  private final Account realAccount;
  SecureAccount(Stats realStats, Account.KeyIF uid,
        Account realAccount) {
    this.realStats = realStats;
    this.uid = uid;
    this.secureStats = new SecureStats(realStats, uid);
    this.realAccount = realAccount;
  }
  // ...
}

// In SecureStats:
Account realAccount = ...;
return new SecureAccount(this.realStats, this.uid, realAccount);
```

Listing 9: Kapselung eines Accounts in der Klasse SecureAccount

Listing 9 zeigt am Beispiel von Accounts, wie jedes Objekt in ein zusätzliches Objekt der Sicherheitsschicht gekapselt wird. Diese Objekte übergeben sich gegenseitig jeweils im Konstruktor eine Referenz auf das intern verwaltete Binding (`realStats`), die `uid` des aktuellen Benutzers und das zu kapselnde Objekt.

Neben den Interfaces und Klassen für einzelne Knoten gibt es für jeden Knoten auch ein Collection-Interface und die dazugehörige Implementierung. Auch diese müssen gekapselt werden, sodass sie nicht die ungeschützten Objekte herausgeben. Verfolgt man hier den gleichen Ansatz, dass die eigentliche Collection in einem Objekt gekapselt wird und die `get`-Methode sowie der Iterator alle herausgegebenen Objekte dynamisch in ein abgesichertes Objekt kapselt, so stößt man beispielsweise bei der `add`-Methode auf das Problem, dass sie das erhaltene gekapselte Objekt entkapseln müsste, was bei einer guten Kapselung und verbotener Reflection jedoch nicht möglich ist. Ein anderer Ansatz ist es, bei Anforderung der Collection alle darin enthaltenen Objekte zu kapseln und eine Collection mit den bereits gekapselten Objekten zu erstellen und zurückzugeben. Bei diesem Ansatz ist kein Entkapseln der Objekte mehr nötig.

Um die Implementierung aller im Collection-Interface vorgesehenen Methoden nicht in jeder einzelnen Collection-Klasse wiederholen zu müssen, kann die in Listing 10 gezeigte Klasse verwendet werden. Es muss lediglich eine Methode `encapsulate` bereitgestellt werden, die die Kapselung der Objekte durchführt.

```
public abstract class AbstractSecureMultisetCollection<SEC, NONSEC>
      extends AbstractCollection<SEC> {
  protected abstract SEC encapsulate(Stats realStats,
        Account.KeyIF uid, NONSEC o);
  protected AbstractSecureMultisetCollection(Stats realStats,
        Account.KeyIF uid,
        Collection<? extends NONSEC> realCollection) {
    super(new HashMultiset<SEC>());
    for (NONSEC e : realCollection) {
      this.add(this.encapsulate(realStats, uid, e));
    }
  }
  protected AbstractSecureMultisetCollection(
        Multiset<SEC> secureMultiset) {
    super(secureMultiset);
  }
  protected AbstractSecureMultisetCollection(
        AbstractSecureMultisetCollection<SEC, ?> orig,
        Filter<? super SEC> filter) {
    // ...
  }
}
```

Listing 10: Hilfsklasse für gekapselte Collections

Listing 11 zeigt, wie diese Klasse verwendet werden kann, um eine Collection zu implementieren. Der erste Konstruktor wird verwendet, wenn eine dem echten Binding entnommene Collection gekapselt werden soll. Der zweite Konstruktor wird benötigt, wenn ein Multiset aus bereits abgesicherten Objekten zusammengestellt wurde und als Collection benötigt wird - diese Objekte müssen nicht mehr gekapselt werden. Dies ist zum Beispiel dann nötig, wenn eine abgesicherte Collection die Collection mit Kindern aller Objekte darin liefern soll. Die gesicherte Collection enthält nur die abgesicherten Objekte und kann demnach keine ungeschützte Collection mehr erhalten. Stattdessen iteriert sie über alle Elemente und ruft dort die entsprechende Methode für den Kindknoten auf, was das bereits gekapselte Objekt liefert. Dieses iterierende Vorgehen entspricht im Übrigen auch der Implementierung des originalen Bindings. Der dritte Konstruktor dient lediglich dazu, den Aufruf von `filter` in geeignete Weise an die Hilfsklasse weiterleiten zu können, damit diese die geforderte Collection zusammenstellen kann. Das Binding erzeugt in jedem Collection-Interface eine derartige `filter`-Methode.

```
public static class SecureAccountCollection<E extends SecureAccount>
    extends AbstractSecureMultisetCollection
      <SecureAccount, Stats.Account>
    implements Stats.AccountCollection<SecureAccount> {
  public SecureAccountCollection(Stats realStats, Account.KeyIF uid,
      Stats.AccountCollection<? extends Account>
         realAccountCollection) {
    super(realStats, uid, realAccountCollection);
  }
  public SecureAccountCollection(
        Multiset<SecureAccount> secureAccountMultiset) {
    super(secureAccountMultiset);
  }
  private SecureAccountCollection(SecureAccountCollection<E> orig,
        Filter<Account> filter) {
    super(orig, filter);
  }
  @Override
  protected SecureAccount encapsulate(Stats realStats,
        Account.KeyIF uid, Account realAccount) {
    return new SecureAccount(realStats, uid, realAccount);
  }
  @Override
  public SecureAccountCollection<? extends SecureAccount> filter(
        Filter<Account> filter) {
    return new SecureAccountCollection<E>(this, filter);
  }
  // ...
}
```

Listing 11: Implementierung der Collection für gekapselte Accounts (Auszug)

3.2.4. Prüfen der Prozedurrechte

Wie im Kapitel 3.2.1 erläutert, soll die Überprüfung der Prozeduraufrufberechtigungen innerhalb der Sicherheitsschicht erfolgen und nicht an das intern verwaltete Binding ausgelagert werden.

Der Generator legt Prozeduren in der Klasse des Wurzelelements an, im Beispielsystem also in der Klasse `Stats`. Zusätzlich werden die Prozeduren zur einfacheren Handhabung auch in den Klassen, deren Schlüssel als Prozedur-Parameter vorkommen, bereitgestellt. Die Prozedur `changeAttributes_account` mit den Parametern `key accountKey, string username, string firstName, string lastName` wird beispielsweise auch in der Klasse `Stats.Account` angeboten, dann aber ohne den Parameter `accountKey`, da sich ein Aufruf der

```
try {
  if (new TutorRole(this.realStats, this.realStudent.parent().key(),
    new Stats.Exercise.Group.Key(this.realStudent.group().keyValue())
    ).hasRole(this.uid)) {
      allowed = true;
  }
} catch (NoSuchElementException e) {
  if (!allowed && new TutorRole(this.realStats,
      this.realStudent.parent().key()
    ).hasRole(this.uid)) {
      allowed = true;
  }
}
```

Listing 12: Behandeln von Rollen für das nicht existente Objekt

Methode auf einem Account-Objekt immer auf diesen Account bezieht. Diese zusätzlichen Prozeduren rufen jedoch lediglich die Prozedur im übergeordneten Objekt auf und setzen dabei ihren eigenen Schlüssel als fehlenden Parameter ein.

In der Sicherheitsschicht wird diese Weiterleitung der Aufrufe ebenfalls umgesetzt, allerdings wird an das abgesicherte Wurzelelement (`SecureStats`) weitergeleitet, wo dann die Prüfung der Prozeduraufrufberechtigungen erfolgt. Auch die neu hinzugekommenen Leseprozeduren werden hier implementiert.

Um die Berechtigung eines Prozeduraufrufs prüfen zu können benötigt man zuerst alle Rollen, die eine `call`-Berechtigung für diese Prozedur haben. Nun prüft man, ob der aktuelle Benutzer eine dieser Rollen hat und ob die innerhalb der `call`-Berechtigung angegebenen Bedingungen erfüllt sind. Bedingungen der Form `assert rollenparameter = ...` können direkt dazu benutzt werden, die Rolle mit passenden Parametern zu instantiieren. Dabei wird der Ausdruck ausgewertet und eingesetzt. Kommt ein Rollenparameter in keiner Bedingung vor, so wird eine Wildcard eingesetzt. Liefert der Ausdruck eine `NosuchElementException`, so wird für diesen Parameter wie in Kapitel 3.2.2 diskutiert ein Nullpointer gewählt, also der Konstruktor ohne den entsprechenden Parameter (Listing 12). Kommt ein Rollenparameter hingegen nur innerhalb eines Pfadausdrucks vor, so muss über diesen Parameter mit einer Schleife quantifiziert werden und für jeden Treffer der Besitz der Rolle und die Gültigkeit der Zusicherungen geprüft werden. Ist keine der Prüfungen erfolgreich, so ist der Aufruf unzulässig und eine `ProcedureAccessDeniedException` wird geworfen. Listing 13 zeigt die Implementierung anhand der Prozedur `registerStudent`.

Es ist wichtig, dass der Rolle im Konstruktor das `realStats` übergeben wird. Das Auslesen von Attributen zur Prüfung der Berechtigungen unterliegt keiner Zugriffs-

```
public void registerStudent(Stats.Exercise.KeyIF exerciseKey, Stats.
    Account.KeyIF studentKey) throws ... {
  boolean allowed = false;
  if (new AssistantRole(this.realStats, exerciseKey).hasRole(this.uid
      )) {
    allowed = true;
  }
  if (!allowed && new StudentRole(this.realStats).hasRole(this.uid))
      {
    if (this.uid.equals(studentKey)
        && this.realStats.exercise(exerciseKey).hasOpen()) {
      allowed = true;
    }
  }

  if (!allowed) {
    throw new ProcedureAccessDeniedException("registerStudent",
        new Object[] { exerciseKey, studentKey });
  }

  this.realStats.registerStudent(exerciseKey, studentKey);
}
```

Listing 13: Implementierung der Rechteüberprüfung in registerStudent

kontrolle, daher darf die Rolle nicht mit der abgesicherten Klasse `SecureStats` arbeiten. Ansonsten würde es gegebenenfalls ungewollte Zugriffsverletzungen bei den von `checkConstraint` getätigten Leseoperationen geben.

3.2.5. Prüfen der Leserechte

Die Prüfung der Leserechte erfolgt nach dem gleichen Schema wie die der Prozeduraufrufberechtigungen. Betrachtet wird der Fall eines Elements x im Datenschema, das ein anderes sich wiederholendes Element y enthält. y enthält wiederum ein eindeutiges Attribut id vom Typ String. Dies entspricht folgender Situation: `element x {element y * id { }}`. Daraus resultieren die folgenden Methoden in der Klasse X:

- `YCollection y()` liefert alle y-Kinder in x. Dies ist erlaubt, sofern man eine Leseberechtigung für y hat, die nicht abhängig vom Schlüssel ist.
- `Y y(Y.KeyIF key)` liefert das y-Kind in x mit dem angegebenen Schlüssel. Dies ist erlaubt, sofern man eine Leseberechtigung für y mit diesem Schlüssel hat.

- `YCollection y(Y.KeyCollectionIF keyCollection)` liefert alle y-Kinder in x mit den angegebenen Schlüsseln. Dies ist erlaubt, sofern man eine Leseberechtigung für alle y hat, deren Schlüssel in der `keyCollection` enthalten sind.
- `Y yById(String id)` liefert das y-Kind in x mit dem angegebenen Wert im Attribut `id`. Dies ist erlaubt, sofern man eine Leseberechtigung für das durch `id` identifizierte y hat.
- `YCollection yById(Y.IdCollection id)` liefert die y-Kinder in x mit den angegebenen Werten im Attribut `id`. Dies ist erlaubt, sofern man eine Leseberechtigung für alle durch die angegebenen Attribute in `id` identifizierte y hat.
- `boolean hasY(Y.KeyIF key)` prüft das Vorhandensein eines y mit dem angegebenen Schlüssel. Hat der Benutzer eine Leseberechtigung für das y mit diesem Schlüssel, so ist der Zugriff erlaubt.
- `boolean hasYById(String id)` prüft das Vorhandensein eines y mit dem angegebenen Attribut `id`. Existiert ein solches y und der Benutzer hat eine Leseberechtigung für dieses y ist der Zugriff gestattet (Ausgabe `true`). Existiert kein solches y und der Benutzer hat eine Leseberechtigung für alle y unabhängig vom Schlüssel, ist der Zugriff gestattet (Ausgabe `false`). Anderenfalls ist der Zugriff nicht gestattet.

Durch eine Integritätsbedingung `count(...) = 1` für ein Attribut z in y, also der Definition, dass z eindeutig ist, kommt eine weitere Methode `Y yByZ(String z)` und `boolean hasYByZ(String z)` hinzu. Hier muss zusätzlich die Leseberechtigung auf das Attribut z geprüft werden. Darf der Benutzer das Attribut y nicht lesen, so ist auch dies eine Rechteverletzung.

3.2.6. Implementierung der Leseprozeduren

Die Leseprozeduren aus der Schemaerweiterung müssen in entsprechenden Java-Quellcode umgesetzt werden. Listing 15 zeigt, wie dies am Beispiel von `getNumberGrades` möglich ist.

Die Deklaration von lokalen Variablen ist in Java entsprechend auch eine Variablendeklaration. Der Typ `int` steht in XCend für eine unbeschränkt große ganze Zahl, er wird daher in Java mit einem `BigInteger` realisiert. Der in der Spracherweiterung neu eingeführte Typ `double` entspricht genau dem Typ `double` in Java, kann also entsprechend übernommen werden. Die Semantik der Leseprozeduren sieht vor, dass nicht explizit mit einem Zahlenwert initialisierte Variablen den

```
if (!(/* Ausdruck der Zusicherung */)) {
  throw new ProcedureAssertionFailedException("nameDerProzedur",
    new Object[] { prozedurParameter1, prozedurParameter2, ... },
    "Ausdruck der Zusicherung"
  );
}
```

Listing 14: Überprüfung einer Zusicherung

Startwert 0 erhalten. In Java ist die Benutzung einer nicht initialisierten Variable hingegen nicht erlaubt (vgl. [GJS+, Kap. 4.12.5, S. 82]). Durch diese Wahl der impliziten Initialisierung wird das Problem umgangen, dass ein Compiler für die Spracherweiterung erst prüfen müsste, ob jede lokale Variable vor ihrer Verwendung auch initialisiert wurde. Bei der Übertragung der Variablennamen in Java muss sichergestellt werden, dass jeder Name ein gültiger Bezeichner in Java ist. Ein automatisiert arbeitender Compiler könnte einfach den Präfix „_“ voranstellen, wenn der Variablenname beispielsweise mit einer Zahl beginnt oder ein reserviertes Schlüsselwort ist.

Zuweisungen an lokale Variablen in XCend sind in Java ebenfalls Zuweisungen. Für arithmetische Operation können die Operationen der Klasse `BigInteger` bzw. die der JLS für `double`-Werte eingesetzt werden. Beim Vermischen verschiedener Typen ist eine Umwandlung entsprechend der in 3.1.3 genannten Regeln durchzuführen.

Quantifizierungen über XCend Pfadausdrücke sowie die `foreach`-Schleife werden in Java mit einer `for`-Schleife der neuen Syntax („enhanced for statement“, [GJS+, Kap. 14.14.2, S. 390]) realisiert. Die `sum`-Funktion muss ebenfalls mit einer Schleife realisiert werden. Falls das Ergebnis keinen Bezeichner erhält muss hierfür eine temporäre lokale Variable eingeführt werden.

Fallunterscheidungen und `return`-Anweisungen können direkt übernommen werden, da die Semantik der Spracherweiterung so gewählt wurde, dass sie mit der JLS in diesen Punkten übereinstimmt.

Zusicherungen werden wie in Listing 14 dargestellt überprüft.

```
public BigInteger getNumberGrades(Stats.Exam.KeyIF examKey,
      Stats.Exam.Grade.KeyIF gradeKey) {
  BigInteger lowerBound = BigInteger.valueOf(0); // int lowerBound
  BigInteger upperBound = BigInteger.valueOf(0); // int upperBound
  BigInteger bound = BigInteger.valueOf(0); // int bound
  BigInteger partSum = BigInteger.valueOf(0); // int partSum
  BigInteger result = BigInteger.valueOf(0); // int result

  // lowerBound = /stats/exam[examKey]/grade[gradeKey]/minPoints
  lowerBound = this.realStats.exam(examKey).grade(gradeKey)
               .minPoints();

  // upperBound = sum(/stats/exam[examKey]/task/maxPoints) + 1
  upperBound = BigInteger.valueOf(0);
  for (Stats.Exam.Task task : this.realStats.exam(examKey).task()) {
    upperBound = upperBound.add(task.maxPoints());
  }
  upperBound = upperBound.add(BigInteger.valueOf(1));
  // foreach /stats/exam[examKey]/grade[$x]/minPoints do
  for (Stats.Exam.Grade x : this.realStats.exam(examKey).grade()) {
    // bound = /stats/exam[examKey]/grade[$x]/minPoints
    bound = x.minPoints();
    // if bound > lowerBound && bound < upperBound then
    if (bound.compareTo(lowerBound) > 0
          && bound.compareTo(upperBound) < 0) {
      upperBound = bound;
    } // fi
  } // done

  // foreach /stats/exam[examKey]/participant[$x] do
  for (Stats.Exam.Participant x :
        this.realStats.exam(examKey).participant()) {
    // partSum =
    //   sum(/stats/exam[examKey]/participant[$x]/result/points)
    partSum = BigInteger.valueOf(0);
    for (Stats.Exam.Participant.Result r : x.result()) {
      partSum = partSum.add(r.points());
    }
    // if partSum >= lowerBound && partSum < upperBound then
    if (partSum.compareTo(lowerBound) >= 0
          && partSum.compareTo(upperBound) < 0) {
      // result = result + 1
      result = result.add(BigInteger.valueOf(1));
    } // fi
  } // done
  return result;
}
```

Listing 15: Leseprozedur getNumberGrades (ohne Berechtigungsüberprüfung)

4. Zusammenfassung und Ausblick

Es wurde ein System vorgestellt, das Benutzerautorisierung mit parametrisierten Rollen ermöglicht. Eine dafür entworfene Beschreibungssprache erlaubt es, die Berechtigungen auf eine Art und Weise zu definieren, in der ein menschlichen Leser schnell und übersichtlich die Berechtigungen der einzelnen Personengruppen sehen kann. Für die Vergabe von Berechtigungen auf abgeleitete Werte wurden Leseprozeduren eingeführt, welche mit lokalen Variablen und einer quantifizierenden Schleife quasi beliebige Berechnungen ermöglichen. Die Kennung des aktuell angemeldeten Benutzers wird dabei einmalig an die Sicherheitsschicht übergeben, welche alle folgenden Zugriffe mit diesen Daten prüft. Die exemplarisch getätigte Implementierung beweist die Umsetzbarkeit eines solchen Systems.

Diese Arbeit befasst sich mit der Benutzerautorisierung. Ein weiterer Aspekt ist die Authentifizierung, bei der die Identität eines Benutzers festgestellt und verifiziert wird. Um auch diesen Aspekt aus der darüberliegenden Schicht heraus zu trennen könnte eine zusätzliche Anweisung das setzen der `uid` des aktuellen Benutzers erlauben. Der Sicherheitsschicht würde so nicht die Benutzerkennung vor der Verwendung (zum Beispiel im Konstruktor der entsprechenden Klasse) übergeben werden, sondern sie befindet sich initial in einem unauthentifizierten Modus und eine entsprechende Prozedur bekommt Anmeldenamen und Passwort übergeben, damit sie diese Daten prüfen und die `uid` entsprechend setzen kann. Dabei wäre auch die Einführung von Hashfunktionen eine Überlegung wert, da sie eine sichere Speicherung von Passwörtern ermöglichen würden.

Eine Erweiterungsmöglichkeit dieser Arbeit ist ein Generator, der ein gegebenes XCend Schema mit Spracherweiterung automatisch in entsprechenden Java Quelltext (oder den Quelltext einer anderen Programmiersprache) umwandelt. Die erarbeiteten Hilfsklassen können dabei weiterhin eingesetzt werden.

Die in Prozeduren gekapselten Operationen benötigen keine gesonderten Berechtigungen, hier wird nur die Berechtigung für die Prozedur überprüft. Auf diese Weise könnten jedoch implizit Informationen ausgelesen werden: Eine fehlgeschlagene Vorbedingung, die eine Verletzung einer Eindeutigkeit geschadet ist, verrät beispielsweise das Vorhandensein eines Knotens mit einem bestimmten Schlüssel. Ein weiterer Aspekt ist das von Prozeduren bereitgestellte Kopieren nicht zugreifbarer Daten innerhalb des Dokuments an eine andere Stelle, der der Benutzer auslesen kann. Man könnte versuchen, eine statische Analyse zu erarbeiten, die derartige implizite Leseberechtigungen ermittelt und dem Designer zur Überprüfung vorlegt. Auch für Schreibrechte wäre eine derartige Analyse denkbar, also eine Zusammenstellung von Pfaden, auf die eine Rolle durch die Verwendung der Prozeduren Schreibrechte hat.

A. Das STAT System in der erweiterten XCend Schemadefinition

```
element stats { @[ size(./account/admin) > 0 ]
  attribute revision { int }

  public element account * username {
    private attribute lastName  { string }
    private attribute firstName { string }
    private attribute email     { string }
    private attribute password  { string }
    private attribute code ?    { string }
    private attribute reset ?   { string }

    element admin ?             { }
    element examiner *          {@[ exists /stats/exam[#examiner]/
                                      examiner[#account] ]}
    element assistant *         {@[ exists /stats/exercise[#assistant]/
                                      assistant[#account] ]}
    element tutor * {
      public element group *    {@[ exists /stats/exercise[#tutor]/
                                      group[#group]/tutor[#account] ]}
    }
    element student ? {
      attribute id              { string @[ count(.,
                                    /stats/account/student/id) = 1 ]}
    }
  }

  element exercise * id {
    attribute lecture           { string }
    attribute term              { string }
    public element open ?       { }

    element assistant *         {@[ exists /stats/account[#assistant]/
                                      assistant[#exercise] ]}

    element group * id          { @[ count(#group, ..exercise/
                                      student/group) <= ./maxSize ]
      attribute day             { "Monday" | "Tuesday" | "Wednesday"
                                  | "Thursday" | "Friday"
                                  | "Saturday" | "Sunday" }
      attribute time            { string }
      attribute location        { string }
      attribute curSize         { int @[ . = count(#group,
                                    ..exercise/student/group) ]}
      attribute maxSize         { int @[ . >= 0 ]}
```

```
    public element tutor *  {@[ exists /stats/account[#tutor]/
                                tutor[#exercise]/group[#group] ]}
  }

  element sheet * id {
    attribute maxPoints     { int @[ . >= 0 ]}
  }

  element student * {       @[ exists /stats/account[#student]/
                                student/id ]
    attribute group ?       { key @[ exists ..exercise/group[.] ]}
    attribute team ?        { int @[ exists ../group ]}
    element result * {      @[ exists ..exercise/sheet[#result] ]
      attribute points      { int @[ . >= 0 && . <= ..exercise/
                                sheet[#result]/maxPoints ]}
    }
  }
}

element exam * id {
  attribute title           { string }
  attribute date            { string }
  attribute time            { string }
  attribute location        { string }
  public element free ?     { }
  public element published ? { }

  attribute exercise        { key @[ exists /stats/exercise[.] ]}

  element examiner *        {@[ exists /stats/account[#examiner]/
                                examiner[#exam] ]}

  element task * id {
    attribute maxPoints     { int @[ . >= 0 ]}
  }
  element grade * name {
    attribute value         { int }
    attribute minPoints     { int }
  }         @[ {exists ./grade[$x], exists ./grade[$y], $x != $y}
                 ./grade[$x]/value != ./grade[$y]/value ]
            @[ {exists ./grade[$x], exists ./grade[$y], $x != $y}
                 ./grade[$x]/minPoints != ./grade[$y]/minPoints ]
            @[ {exists ./grade[$x], exists ./grade[$y],
                ./grade[$x]/value < ./grade[$y]/value}
                 ./grade[$x]/minPoints > ./grade[$y]/minPoints ]

  element participant * {   @[ exists /stats/account[#participant]/
                                student/id ]
    public element result * { @[ exists ..exam/task[#result] ]
```

```
        attribute points      { int @[ . >= 0 && . <=
                                  ..exam/task[#result]/maxPoints ]}
      }
    }
  }
}

role admin {
  assert exists /stats/account[uid]/admin

  include assistant(assExercise) { }
  include examiner(exExam) { }

  call createAccount(key accountKey, string username,
         string lastName, string firstName, string email,
         string password) { }
  call addStudentId(key accountKey, String studentId) { }
  call changeAttributes_account(key accountKey, string username,
         string firstName, string lastName) { }
  call changePassword(key accountKey, string password) { }
  call createExam(key examKey, string id, key eid, string title,
         string date, string time, string location) { }
  call createExercise(key exerciseKey, string id, string lecture,
         string term) { }
  call deleteAccount(key accountKey) {
    assert uid != accountKey
  }
  call deleteExam(key examKey) { }
  call deleteExercise(key exerciseKey) { }
  call grantAdminRights(key accountKey) { }
  call grantAssistantRights(key accountKey, key exerciseKey) { }
  call grantExaminerRights(key accountKey, key examKey) { }
  call removeStudentId(key accountKey) { }
  call revokeAdminRights(key accountKey) { }
  call revokeAssistantRights(key accountKey, key exerciseKey) { }
  call revokeExaminerRights(key accountKey, key examKey) { }

  read /stats/account/lastName { }
  read /stats/account/firstName { }
  read /stats/account/email { }
  read /stats/account/student { }
  read /stats/account/admin { }
  read /stats/account/assistant { }
  read /stats/account/examiner { }
  read /stats/account/tutor { }
  read /stats/exercise { }
  read /stats/exam { }
  read /stats/exam/examiner { }
}
```

```
role assistant(exercise) {
  assert exists /stats/account[uid]/assistant[exercise]

  include tutor(tutExercise, tutGroup) {
    assert tutExercise = exercise
  }
  include examiner(exExam) {
    assert /stats/exam[exExam]/exercise = exercise
  }

  call changeAttributes_exercise(key exerciseKey, string exerciseId,
         string lecture, string term) {
    assert exercise = exerciseKey
  }
  call changeAttributes_group(key exerciseKey, key groupKey,
         string groupId, string day, string time, string location,
         int maxSize) {
    assert exercise = exerciseKey
  }
  call changeAttributes_sheet(key exerciseKey, key sheetKey,
         string sheetId, int maxPoints) {
    assert exercise = exerciseKey
  }
  call closeSignUp(key exerciseKey) {
    assert exercise = exerciseKey
  }
  call createGroup(key exerciseKey, key groupKey, string groupId,
         string day, string time, string location, int maxSize) {
    assert exercise = exerciseKey
  }
  call createSheet(key exerciseId, key sheetKey, string sheetId,
         int maxPoints) {
    assert exercise = exerciseKey
  }
  call deleteGroup(key exerciseKey, key groupKey) {
    assert exercise = exerciseKey
  }
  call deleteSheet(key exerciseKey, key sheetKey) {
    assert exercise = exerciseKey
  }
  call grantTutorRights(key accountKey, key exerciseKey,
         key groupKey) {
    assert exercise = exerciseKey
  }
  call openSignUp(key exerciseId) {
    assert exercise = exerciseKey
  }
```

```
  call registerStudent(key exerciseKey, key studentKey) {
    assert exercise = exerciseKey
  }
  call revokeTutorRights(key accountKey, key exerciseKey,
         key groupKey) {
    assert exercise = exerciseKey
  }
  call signOutGroup(key exerciseKey, key studentKey) {
    assert exercise = exerciseKey
  }
  call signUpGroup(key exerciseKey, key studentKey, key groupKey) {
    assert exercise = exerciseKey
  }
  call unregisterStudent(key exerciseKey, key studentKey) {
    assert exercise = exerciseKey
  }
  call grantExaminerRights(key accountKey, key examKey) {
    assert exercise = /stats/exam[examKey]/exercise
  }
  call revokeExaminerRights(key accountKey, key examKey) {
    assert exercise = /stats/exam[examKey]/exercise
  }
  call createExam(key examKey, string id, key eid, string title,
         string date, string time, string location) {
    assert exercise = eid
  }
  call deleteExam(key examKey) {
    assert exercise = /stats/exam[examKey]/exercise
  }

  read /stats/exercise { }
  read /stats/exercise[exercise]/assistant { }
  read /stats/exercise[exercise]/group { }
  read /stats/exercise[exercise]/sheet { }
  read /stats/exercise[exercise]/student { }
  read /stats/exercise[exercise]/student/result { }
  read /stats/exam { }
  read(examKey) /stats/exam[examKey]/examiner {
    assert exercise = /stats/exam[examKey]/exercise
  }

  // useful to see name before adding account into own exercise
  read /stats/account/lastName { }
  read /stats/account/firstName { }
  read /stats/account/email { }
  read /stats/account/student { }
}
```

```
role examiner(exam) {
  assert exists /stats/account[uid]/examiner[examKey]

  call addParticipant(key examKey, key studentKey) {
    assert exam = examKey
  }
  call addResult_task(key examKey, key studentKey, key taskKey,
         int points) {
    assert exam = examKey
  }
  call changeAttributes_exam(key examKey, string examId,
         string title, string date, string time, string location) {
    assert exam = examKey
  }
  call changeAttributes_task(key examKey, key taskKey, string taskId,
         int maxPoints) {
    assert exam = examKey
  }
  call changeResult_task(key examKey, key studentKey, key taskKey,
         int points) {
    assert exam = examKey
  }
  call closeRegistration(key examKey) {
    assert exam = examKey
  }
  call createGrade(key examKey, key gradeKey, string name, int value,
         int minPoints) {
    assert exam = examKey
  }
  call createTask(key examKey, key taskKey, string taskId,
         int maxPoints) {
    assert exam = examKey
  }
  call deleteGrade(key examKey, key gradeKey) {
    assert exam = examKey
  }
  call deleteTask(key examKey, key taskKey) {
    assert exam = examKey
  }
  call hideResults(key examKey) {
    assert exam = examKey
  }
  call openRegistration(key examKey) {
    assert exam = examKey
  }
  call publishResults(key examKey) {
    assert exam = examKey
  }
```

```
  call removeParticipant(key examKey, key studentKey) {
    assert exam = examKey
  }
  call removeResult_task(key examKey, key studentKey, key taskKey) {
    assert exam = examKey
  }
  call getNumberRegisteredExamParticipants(key examKey) {
    assert exam = examKey
  }
  call getNumberParticipatedExamParticipants(key examKey) {kk
    assert exam = examKey
  }
  call getNumberGrades(key examKey, key gradeKey) {
    assert exam = examKey
  }
  call getAverageTaskPointsByGroup(key examKey, key taskKey,
         key groupKey) {
    assert exam = examKey
  }
  call getAverageTaskPoints(key examKey, key taskKey) {
    assert exam = examKey
  }

  read /stats/exam { }
  read /stats/exam[exam]/examiner { }
  read /stats/exam[exam]/task { }
  read /stats/exam[exam]/grade { }
  read /stats/exam[exam]/participant { }

  // useful to see name before adding account into own exam
  read /stats/account/lastName { }
  read /stats/account/firstName { }
  read /stats/account/email { }

  read /stats/account/student { }
  read(exerciseKey) /stats/exercise[exerciseKey]/sheet {
    assert exerciseKey = /stats/exam[exam]/exercise
  }
  read(exerciseKey) /stats/exercise[exerciseKey]/student {
    assert exerciseKey = /stats/exam[exam]/exercise
  }
  read(exerciseKey) /stats/exercise[exerciseKey]/student/result {
    assert exerciseKey = /stats/exam[exam]/exercise
  }
}
```

```
role tutor(exercise, group) {
  assert exists /stats/account[uid]/tutor[exercise]/group[group]

  call addResult_sheet(key exerciseKey, key studentKey, key sheetKey,
         int points) {
    assert exercise = exerciseKey
    assert group
             = /stats/exercise[exerciseKey]/student[studentKey]/group
  }
  call assignTeam(key exerciseKey, key studentKey, int teamKey) {
    assert exercise = exerciseKey
    assert group
             = /stats/exercise[exerciseKey]/student[studentKey]/group
  }
  call changeResult_sheet(key exerciseKey, key studentKey,
         key sheetKey, int points) {
    assert exercise = exerciseKey
    assert group
             = /stats/exercise[exerciseKey]/student[studentKey]/group
  }
  call leaveTeam(key exerciseKey, key studentKey) {
    assert exercise = exerciseKey
    assert group
             = /stats/exercise[exerciseKey]/student[studentKey]/group
  }
  call removeResult_sheet(key exerciseKey, key studentKey, key
      sheetKey) {
    assert exercise = exerciseKey
    assert group
             = /stats/exercise[exerciseKey]/student[studentKey]/group
  }
  call getNumberParticipatedExamParticipants(key examKey) {
    assert exercise = /stats/exam[examKey]/exercise
        && exists /stats/exam[examKey]/published
  }
  call getNumberGrades(key examKey, key gradeKey) {
    assert exercise = /stats/exam[examKey]/exercise
        && exists /stats/exam[examKey]/published
  }
  call getAverageTaskPointsByGroup(key examKey, key taskKey,
         key groupKey) {
    assert exercise = /stats/exam[examKey]/exercise
      && group = groupKey
        && exists /stats/exam[examKey]/published
  }
  call getAverageTaskPoints(key examKey, key taskKey) {
    assert exercise = /stats/exam[examKey]/exercise
        && exists /stats/exam[examKey]/published
  }
```

```
  read /stats/exercise
  read /stats/exercise[exercise]/group { }
  read /stats/exercise[exercise]/sheet { }
  read(studentKey) /stats/exercise[exercise]/student[studentKey] {
    assert group
             = /stats/exercise[exercise]/student[studentKey]/group
  }
  read(studentKey) /stats/exercise[exercise]/student[studentKey]/
      result {
    assert group
             = /stats/exercise[exercise]/student[studentKey]/group
  }
  read /stats/exercise[exercise]/assistant { }
  read(accountKey) /stats/account[accountKey]/lastName {
    // see public account info...
    // ...of own group members
    assert group
             = /stats/exercise[exercise]/student[accountKey]/group

    // ...and of other tutors
        || exists /stats/account[accountKey]/tutor[exercise]

    // ...and of the assistants (to contact them, of course.)
        || exists /stats/account[accountKey]/assistant[exercise]
  }
  read(accountKey) /stats/account[accountKey]/firstName {
    assert group
             = /stats/exercise[exercise]/student[accountKey]/group
        || exists /stats/account[accountKey]/tutor[exercise]
        || exists /stats/account[accountKey]/assistant[exercise]
  }
  read(accountKey) /stats/account[accountKey]/email {
    assert group
             = /stats/exercise[exercise]/student[accountKey]/group
        || exists /stats/account[accountKey]/tutor[exercise]
        || exists /stats/account[accountKey]/assistant[exercise]
  }
  read /stats/exam { }
  read(examKey) /stats/exam[examKey]/task {
    assert exercise = /stats/exam[examKey]/exercise
  }
  read(examKey) /stats/exam[examKey]/grade {
    assert exercise = /stats/exam[examKey]/exercise
  }
}
```

```
role student {
  assert exists /stats/account[uid]/student

  call addParticipant(key examKey, key studentKey) {
    assert studentKey = uid
    assert exists /stats/exam[examKey]/free
  }
  call registerStudent(key exerciseKey, key studentKey) {
    assert studentKey = uid
    assert exists /stats/exercise[exerciseKey]
    assert exists /stats/exercise[exerciseKey]/open
  }
  call removeParticipant(key examKey, key studentKey) {
    assert studentKey = uid
    assert exists /stats/exam[examKey]/free
  }
  call signOutGroup(key exerciseKey, key studentKey) {
    assert studentKey = uid
    assert exists /stats/exercise[exerciseKey]
    assert exists /stats/exercise[exerciseKey]/open
  }
  call signUpGroup(key exerciseKey, key studentKey, key groupKey) {
    assert studentKey = uid
    assert exists /stats/exercise[exerciseKey]
    assert exists /stats/exercise[exerciseKey]/open
  }
  call unregisterStudent(key exerciseKey, key studentKey) {
    assert studentKey = uid
    assert exists /stats/exercise[exerciseKey]
    assert exists /stats/exercise[exerciseKey]/open
  }
  call getNumberParticipatedExamParticipants(key examKey) {
    assert exists /stats/exam[examKey]/participant[uid]
        && exists /stats/exam[examKey]/published
  }
  call getNumberGrades(key examKey, key gradeKey) {
    assert exists /stats/exam[examKey]/participant[uid]
        && exists /stats/exam[examKey]/published
  }
  call getAverageTaskPoints(key examKey, key taskKey) {
    assert exists /stats/exam[examKey]/participant[uid]
        && exists /stats/exam[examKey]/published
  }

  read /stats/exercise { }
  read(exerciseKey) /stats/exercise[exerciseKey]/group {
    assert exists /stats/exercise[exerciseKey]/student[uid]
  }
```

```
  read(exerciseKey) /stats/exercise[exerciseKey]/sheet {
    assert exists /stats/exercise[exerciseKey]/student[uid]
  }
  read /stats/exercise/student[uid]/result { }
  read(studentKey) /stats/exercise/student[studentKey] {
    // see who is in own team
    assert /stats/exercise[$x]/student[uid]/group
              = /stats/exercise[$x]/student[studentKey]/group
        && /stats/exercise[$x]/student[uid]/team
              = /stats/exercise[$x]/student[studentKey]/team
  }
  read(studentKey) /stats/account[studentKey]/lastName {
    // see public account info...
    // ...of other team members
    assert /stats/exercise[$x]/student[uid]/group
              = /stats/exercise[$x]/student[studentKey]/group
        && /stats/exercise[$x]/student[uid]/team
              = /stats/exercise[$x]/student[studentKey]/team

    // ...and of tutors
        || exists /stats/exercise[$x]/student[uid]
        && exists /stats/exercise[$x]/group/tutor[studentKey]
  }
  read(studentKey) /stats/account[studentKey]/firstName {
    assert /stats/exercise[$x]/student[uid]/group
              = /stats/exercise[$x]/student[studentKey]/group
        && /stats/exercise[$x]/student[uid]/team
              = /stats/exercise[$x]/student[studentKey]/team
        || exists /stats/exercise[$x]/student[uid]
        && exists /stats/exercise[$x]/group/tutor[studentKey]
  }
  read(studentKey) /stats/account[studentKey]/email {
    assert /stats/exercise[$x]/student[uid]/group
              = /stats/exercise[$x]/student[studentKey]/group
        && /stats/exercise[$x]/student[uid]/team
              = /stats/exercise[$x]/student[studentKey]/team
        || exists /stats/exercise[$x]/student[uid]
        && exists /stats/exercise[$x]/group/tutor[studentKey]
  }
  read /stats/exam { }
  read(examKey) /stats/exam[examKey]/task {
    assert exists /stats/exam[examKey]/participant[uid]
  }
  read(examKey) /stats/exam[examKey]/grade {
    assert exists /stats/exam[examKey]/participant[uid]
  }
  read /stats/exam/participant[uid] { }
}
```

```
role guest {
  // No assertion -> always true
  call authenticate(key accountKey, string password) { }
  call createStudentAccount(key accountKey, string username,
         string studentId, string lastName, string firstName,
         string email, string password, string code) { }
  call requestReset(key accountKey, string reset) { }
  call resetPassword(key accountKey, string reset,
         string password) { }
  call validateAccount(key accountKey, string code) { }
}

role user {
  assert exists /account[uid]

  call changeAttributes_account(key accountKey, string username,
         string firstName, string lastName) {
    assert accountKey = uid
  }
  call changePassword(key accountKey, string password) {
    assert accountKey = uid
  }

  read /stats/account[uid]/lastName { }
  read /stats/account[uid]/firstName { }
  read /stats/account[uid]/email { }
  read /stats/account[uid]/student { }
  read /stats/account[uid]/admin { }
  read /stats/account[uid]/examiner { }
  read /stats/account[uid]/tutor { }
}

addAdminUnsafe(key accountKey, string username, string lastName,
      string firstName, string email, string password) {
  insert account[accountKey] at /stats
  insert username   at /stats/account[accountKey] using username
  insert lastName   at /stats/account[accountKey] using lastName
  insert firstName  at /stats/account[accountKey] using firstName
  insert email      at /stats/account[accountKey] using email
  insert password   at /stats/account[accountKey] using password
  insert admin      at /stats/account[accountKey]

  update /stats/revision to 0
}
```

```
addParticipant(key examKey, key studentKey) {
  assert exists /stats/exam[examKey]
  assert exists /stats/account[studentKey]/student/id

  assert !exists /stats/exam[examKey]/participant[studentKey]

  insert participant[studentKey] at /stats/exam[examKey]
  update /stats/revision to /stats/revision + 1
}

addResult_sheet(key exerciseKey, key studentKey, key sheetKey,
      int points) {
  assert exists /stats/exercise[exerciseKey]/student[studentKey]
  assert exists /stats/exercise[exerciseKey]/sheet[sheetKey]

  assert !exists /stats/exercise[exerciseKey]/student[studentKey]/
                  result[sheetKey]

  assert points >= 0
  assert points <= /stats/exercise[exerciseKey]/sheet[sheetKey]/
                    maxPoints

  insert result[sheetKey]
    at /stats/exercise[exerciseKey]/student[studentKey]
  insert points at /stats/exercise[exerciseKey]/student[studentKey]/
                    result[sheetKey] using points
  update /stats/revision to /stats/revision + 1
}


addResult_task(key examKey, key studentKey, key taskKey,
      int points) {
  assert exists /stats/exam[examKey]/participant[studentKey]
  assert exists /stats/exam[examKey]/task[taskKey]

  assert !exists /stats/exam[examKey]/participant[studentKey]/
           result[taskKey]

  assert points >= 0
  assert points <= /stats/exam[examKey]/task[taskKey]/maxPoints

  insert result[taskKey]
    at /stats/exam[examKey]/participant[studentKey]
  insert points at /stats/exam[examKey]/participant[studentKey]/
                    result[taskKey] using points
  update /stats/revision to /stats/revision + 1
}
```

```
addStudentId(key accountKey, String studentId) {
  assert  exists /stats/account[accountKey]
  assert !exists /stats/account[accountKey]/student

  assert count(studentId, /stats/account/student/id) = 0

  insert student at /stats/account[accountKey]
  insert id at /stats/account[accountKey]/student using studentId
  update /stats/revision to /stats/revision + 1
}

assignTeam(key exerciseKey, key studentKey, int teamKey) {
  assert exists /stats/exercise[exerciseKey]/student[studentKey]/
           group
  assert !exists /stats/exercise[exerciseKey]/student[studentKey]/
           team

  insert team at /stats/exercise[exerciseKey]/student[studentKey]
    using teamKey
  update /stats/revision to /stats/revision + 1
}

authenticate(key accountKey, string password) {
  assert exists /stats/account[accountKey]

  assert /stats/account[accountKey]/password = password
  update /stats/revision to /stats/revision + 1
}

changeAttributes_account(key accountKey, string username, string
    firstName, string lastName) {
  assert exists /stats/account[accountKey]

  assert username = /stats/account[accountKey]/username
      || count(username, /stats/account/username) = 0

  update /stats/account[accountKey]/username  to username
  update /stats/account[accountKey]/firstName to firstName
  update /stats/account[accountKey]/lastName  to lastName
  update /stats/revision to /stats/revision + 1
}

changeAttributes_exam(key examKey, string examId, string title,
      string date, string time, string location) {
  assert exists /stats/exam[examKey]

  assert examId = /stats/exam[examKey]/id
      || count(examId, /stats/exam/id) = 0
```

```
  update /stats/exam[examKey]/id       to examId
  update /stats/exam[examKey]/title    to title
  update /stats/exam[examKey]/date     to date
  update /stats/exam[examKey]/time     to time
  update /stats/exam[examKey]/location to location
  update /stats/revision to /stats/revision + 1
}

changeAttributes_exercise(key exerciseKey, string exerciseId,
      string lecture, string term) {
  assert exists /stats/exercise[exerciseKey]

  assert exerciseId = /stats/exercise[exerciseKey]/id
      || count(exerciseId, /stats/exercise/id) = 0

  update /stats/exercise[exerciseKey]/id      to exerciseId
  update /stats/exercise[exerciseKey]/lecture to lecture
  update /stats/exercise[exerciseKey]/term    to term
  update /stats/revision to /stats/revision + 1
}

changeAttributes_group(key exerciseKey, key groupKey, string groupId,
      string day, string time, string location, int maxSize) {
  assert exists /stats/exercise[exerciseKey]/group[groupKey]

  assert maxSize >= 0
  assert /stats/exercise[exerciseKey]/group[groupKey]/curSize
           <= maxSize

  assert groupId = /stats/exercise[exerciseKey]/group[groupKey]/id
      || count(groupId, /stats/exercise[exerciseKey]/group/id) = 0

  update /stats/exercise[exerciseKey]/group[groupKey]/id   to groupId
  update /stats/exercise[exerciseKey]/group[groupKey]/day  to day
  update /stats/exercise[exerciseKey]/group[groupKey]/time to time
  update /stats/exercise[exerciseKey]/group[groupKey]/location
    to location
  update /stats/exercise[exerciseKey]/group[groupKey]/maxSize
    to maxSize
  update /stats/revision to /stats/revision + 1
}
```

```
changeAttributes_sheet(key exerciseKey, key sheetKey, string sheetId,
      int maxPoints) {
  assert exists /stats/exercise[exerciseKey]/sheet[sheetKey]

  assert maxPoints >= 0
  assert { exists /stats/exercise[exerciseKey]/student[$x] }
    -> /stats/exercise[exerciseKey]/student[$x]/
         result[sheetKey]/points <= maxPoints

  assert sheetId = /stats/exercise[exerciseKey]/sheet[sheetKey]/id
      || count(sheetId, /stats/exercise[exerciseKey]/sheet/id) = 0

  update /stats/exercise[exerciseKey]/sheet[sheetKey]/id to sheetId
  update /stats/exercise[exerciseKey]/sheet[sheetKey]/maxPoints
    to maxPoints
  update /stats/revision to /stats/revision + 1
}

changeAttributes_task(key examKey, key taskKey, string taskId,
      int maxPoints) {
  assert exists /stats/exam[examKey]/task[taskKey]

  assert maxPoints >= 0
  assert { exists /stats/exam[examKey]/participant[$x] }
    -> /stats/exam[examKey]/participant[$x]/
         result[taskKey]/points <= maxPoints

  assert taskId = /stats/exam[examKey]/task[taskKey]/id
      || count(taskId, /stats/exam[examKey]/task/id) = 0

  update /stats/exam[examKey]/task[taskKey]/id        to taskId
  update /stats/exam[examKey]/task[taskKey]/maxPoints to maxPoints
  update /stats/revision to /stats/revision + 1
}

changePassword(key accountKey, string password) {
  assert exists /stats/account[accountKey]

  update /stats/account[accountKey]/password to password
  update /stats/revision to /stats/revision + 1
}

changeResult_sheet(key exerciseKey, key studentKey, key sheetKey,
      int points) {
  assert exists /stats/exercise[exerciseKey]/student[studentKey]
  assert exists /stats/exercise[exerciseKey]/sheet[sheetKey]

  assert points >= 0 && points
           <= /stats/exercise[exerciseKey]/sheet[sheetKey]/maxPoints
```

```
  assert exists /stats/exercise[exerciseKey]/student[studentKey]/
           result[sheetKey]

  update /stats/exercise[exerciseKey]/student[studentKey]/
           result[sheetKey]/points to points
  update /stats/revision to /stats/revision + 1
}

changeResult_task(key examKey, key studentKey, key taskKey,
      int points) {
  assert exists /stats/exam[examKey]/participant[studentKey]
  assert exists /stats/exam[examKey]/task[taskKey]

  assert points >= 0 && points
           <= /stats/exam[examKey]/task[taskKey]/maxPoints

  assert exists /stats/exam[examKey]/participant[studentKey]/
           result[taskKey]

  update /stats/exam[examKey]/participant[studentKey]/
           result[taskKey]/points to points
  update /stats/revision to /stats/revision + 1
}

closeRegistration(key examKey) {
  assert exists /stats/exam[examKey]
  assert exists /stats/exam[examKey]/free

  delete /stats/exam[examKey]/free
  update /stats/revision to /stats/revision + 1
}

closeSignUp(key exerciseKey) {
  assert exists /stats/exercise[exerciseKey]
  assert exists /stats/exercise[exerciseKey]/open

  delete /stats/exercise[exerciseKey]/open
  update /stats/revision to /stats/revision + 1
}
```

```
createAccount(key accountKey, string username, string lastName,
      string firstName, string email, string password) {
  assert !exists /stats[]/account[accountKey]

  assert count(username, /stats[]/account/username[]) = 0

  insert account[accountKey] at /stats
  insert username  at /stats/account[accountKey] using username
  insert lastName  at /stats/account[accountKey] using lastName
  insert firstName at /stats/account[accountKey] using firstName
  insert email     at /stats/account[accountKey] using email
  insert password  at /stats/account[accountKey] using password
  update /stats/revision to /stats/revision + 1
}

createExam(key examKey, string id, key eid, string title,
      string date, string time, string location) {
  assert !exists /stats/exam[examKey]
  assert exists /stats/exercise[eid]

  assert count(id, /stats/exam/id) = 0

  insert exam[examKey] at /stats
  insert id            at /stats/exam[examKey] using id
  insert title         at /stats/exam[examKey] using title
  insert date          at /stats/exam[examKey] using date
  insert time          at /stats/exam[examKey] using time
  insert location      at /stats/exam[examKey] using location
  insert exercise      at /stats/exam[examKey] using eid
  update /stats/revision to /stats/revision + 1
}

createExercise(key exerciseKey, string id, string lecture,
      string term) {
  assert !exists /stats/exercise[exerciseKey]

  assert count(id, /stats/exercise/id) = 0

  insert exercise[exerciseKey] at /stats
  insert id      at /stats/exercise[exerciseKey] using id
  insert lecture at /stats/exercise[exerciseKey] using lecture
  insert term    at /stats/exercise[exerciseKey] using term
  update /stats/revision to /stats/revision + 1
}

createGrade(key examKey, key gradeKey, string name, int value,
      int minPoints) {
  assert  exists /stats/exam[examKey]
  assert !exists /stats/exam[examKey]/grade[gradeKey]
```

```
  assert count(value,     /stats/exam[examKey]/grade/value) = 0
  assert count(minPoints, /stats/exam[examKey]/grade/minPoints) = 0

  assert { /stats/exam[examKey]/grade[$x]/value < value,
           exists /stats/exam[examKey]/grade[$x] }
           -> /stats/exam[examKey]/grade[$x]/minPoints > minPoints
  assert { value < /stats/exam[examKey]/grade[$x]/value,
           exists /stats/exam[examKey]/grade[$x] }
           -> minPoints > /stats/exam[examKey]/grade[$x]/minPoints

  assert count(name, /stats/exam[examKey]/grade/name) = 0

  insert grade[gradeKey] at /stats/exam[examKey]
  insert name  at /stats/exam[examKey]/grade[gradeKey] using name
  insert value at /stats/exam[examKey]/grade[gradeKey] using value
  insert minPoints
    at /stats/exam[examKey]/grade[gradeKey] using minPoints
  update /stats/revision to /stats/revision + 1
}

createGroup(key exerciseKey, key groupKey, string groupId,
      string day, string time, string location, int maxSize) {
  assert  exists /stats/exercise[exerciseKey]
  assert !exists /stats/exercise[exerciseKey]/group[groupKey]

  assert maxSize >= 0
  assert day = "Monday" || day = "Tuesday" || day = "Wednesday"
      || day = "Thursday" || day = "Friday" || day = "Saturday"
      || day = "Sunday"

  assert count(groupId, /stats/exercise[exerciseKey]/group/id) = 0

  insert group[groupKey] at /stats/exercise[exerciseKey]
  insert id       at /stats/exercise[exerciseKey]/group[groupKey]
    using groupId
  insert day      at /stats/exercise[exerciseKey]/group[groupKey]
    using day
  insert time     at /stats/exercise[exerciseKey]/group[groupKey]
    using time
  insert location at /stats/exercise[exerciseKey]/group[groupKey]
    using location
  insert maxSize  at /stats/exercise[exerciseKey]/group[groupKey]
    using maxSize
  insert curSize  at /stats/exercise[exerciseKey]/group[groupKey]
    using 0
  update /stats/revision to /stats/revision + 1
}
```

```
createSheet(key exerciseId, key sheetKey, string sheetId,
      int maxPoints) {
  assert  exists /stats/exercise[exerciseId]
  assert !exists /stats/exercise[exerciseId]/sheet[sheetKey]

  assert maxPoints >= 0

  assert count(sheetId, /stats/exercise[exerciseId]/sheet/id) = 0

  insert sheet[sheetKey] at /stats/exercise[exerciseId]
  insert id       at /stats/exercise[exerciseId]/sheet[sheetKey]
    using sheetId
  insert maxPoints at /stats/exercise[exerciseId]/sheet[sheetKey]
    using maxPoints

  update /stats/revision to /stats/revision + 1
}

createStudentAccount(key accountKey, string username,
      string studentId, string lastName, string firstName,
      string email, string password, string code) {
  assert !exists /stats/account[accountKey]

  assert count(username,  /stats/account/username  ) = 0
  assert count(studentId, /stats/account/student/id) = 0

  insert account[accountKey] at /stats
  insert username  at /stats/account[accountKey] using username
  insert lastName  at /stats/account[accountKey] using lastName
  insert firstName at /stats/account[accountKey] using firstName
  insert email     at /stats/account[accountKey] using email
  insert password  at /stats/account[accountKey] using password
  insert code      at /stats/account[accountKey] using code
  insert student   at /stats/account[accountKey]
  insert id at /stats/account[accountKey]/student using studentId
  update /stats/revision to /stats/revision + 1
}

createTask(key examKey, key taskKey, string taskId, int maxPoints) {
  assert  exists /stats/exam[examKey]
  assert !exists /stats/exam[examKey]/task[taskKey]

  assert maxPoints >= 0

  assert count(taskId, /stats/exam[examKey]/task/id) = 0

  insert task[taskKey] at /stats/exam[examKey]
  insert id at /stats/exam[examKey]/task[taskKey] using taskId
```

```
  insert maxPoints at /stats/exam[examKey]/task[taskKey]
    using maxPoints
  update /stats/revision to /stats/revision + 1
}

deleteAccount(key accountKey) {
  assert exists /stats/account[accountKey]

  assert size(/stats/account[accountKey]/examiner) = 0
  assert size(/stats/account[accountKey]/assistant) = 0
  assert size(/stats/account[accountKey]/tutor) = 0

  assert { exists /stats/account[accountKey]/admin }
           -> size(/stats/account/admin) > 1

  assert { exists /stats/account[accountKey]/student }
           -> size(/stats/exercise/student[accountKey]) = 0
  assert { exists /stats/account[accountKey]/student }
           -> size(/stats/exam/participant[accountKey]) = 0

  delete /stats/account[accountKey]
  update /stats/revision to /stats/revision + 1
}

deleteExam(key examKey) {
  assert exists /stats/exam[examKey]

  assert size(/stats/exam[examKey]/examiner) = 0

  assert size(/stats/exam[examKey]/participant) = 0

  delete /stats/exam[examKey]
  update /stats/revision to /stats/revision + 1
}

deleteExercise(key exerciseKey) {
  assert exists /stats/exercise[exerciseKey]

  assert size(/stats/exercise[exerciseKey]/assistant) = 0
  assert size(/stats/exercise[exerciseKey]/group/tutor) = 0

  assert size(/stats/exercise[exerciseKey]/student) = 0

  delete /stats/exercise[exerciseKey]
  update /stats/revision to /stats/revision + 1
}
```

```
deleteGrade(key examKey, key gradeKey) {
  assert exists /stats/exam[examKey]/grade[gradeKey]

  delete /stats/exam[examKey]/grade[gradeKey]
  update /stats/revision to /stats/revision + 1
}

deleteGroup(key exerciseKey, key groupKey) {
  assert exists /stats/exercise[exerciseKey]/group[groupKey]

  assert size(/stats/exercise[exerciseKey]/group[groupKey]/tutor) = 0
  assert count(groupKey, /stats/exercise[exerciseKey]/student/group)
           = 0

  delete /stats/exercise[exerciseKey]/group[groupKey]
  update /stats/revision to /stats/revision + 1
}

deleteSheet(key exerciseKey, key sheetKey) {
  assert exists /stats/exercise[exerciseKey]/sheet[sheetKey]

  assert size(/stats/exercise[exerciseKey]/student/result[sheetKey])
     = 0

  delete /stats/exercise[exerciseKey]/sheet[sheetKey]
  update /stats/revision to /stats/revision + 1
}

deleteTask(key examKey, key taskKey) {
  assert exists /stats/exam[examKey]/task[taskKey]

  assert size(/stats/exam[examKey]/participant/result[taskKey]) = 0

  delete /stats/exam[examKey]/task[taskKey]
  update /stats/revision to /stats/revision + 1
}

grantAdminRights(key accountKey) {
  assert  exists /stats/account[accountKey]
  assert !exists /stats/account[accountKey]/admin

  insert admin at /stats/account[accountKey]
  update /stats/revision to /stats/revision + 1
}

grantAssistantRights(key accountKey, key exerciseKey) {
  assert exists /stats/account[accountKey]
  assert exists /stats/exercise[exerciseKey]
```

```
  assert !exists /stats/account[accountKey]/assistant[exerciseKey]
  assert !exists /stats/exercise[exerciseKey]/assistant[accountKey]

  insert assistant[exerciseKey] at /stats/account[accountKey]
  insert assistant[accountKey]  at /stats/exercise[exerciseKey]
  update /stats/revision to /stats/revision + 1
}

grantExaminerRights(key accountKey, key examKey) {
  assert exists /stats/account[accountKey]
  assert exists /stats/exam[examKey]

  assert !exists /stats/account[accountKey]/examiner[examKey]
  assert !exists /stats/exam[examKey]/examiner[accountKey]

  insert examiner[examKey]    at /stats/account[accountKey]
  insert examiner[accountKey] at /stats/exam[examKey]
  update /stats/revision to /stats/revision + 1
}

grantTutorRights(key accountKey, key exerciseKey, key groupKey) {
  assert exists /stats/exercise[exerciseKey]/group[groupKey]
  assert exists /stats/account[accountKey]

  assert !exists /stats/account[accountKey]/tutor[exerciseKey]/
           group[groupKey]
  assert !exists /stats/exercise[exerciseKey]/group[groupKey]/
           tutor[accountKey]

  if !exists /stats/account[accountKey]/tutor[exerciseKey] then
    insert tutor[exerciseKey] at /stats/account[accountKey]
  fi
  insert group[groupKey]
    at /stats/account[accountKey]/tutor[exerciseKey]
  insert tutor[accountKey]
    at /stats/exercise[exerciseKey]/group[groupKey]
  update /stats/revision to /stats/revision + 1
}

hideResults(key examKey) {
  assert exists /stats/exam[examKey]
  assert exists /stats/exam[examKey]/published

  delete /stats/exam[examKey]/published
  update /stats/revision to /stats/revision + 1
}
```

```
leaveTeam(key exerciseKey, key studentKey) {
  assert exists /stats/exercise[exerciseKey]/student[studentKey]/team

  delete /stats/exercise[exerciseKey]/student[studentKey]/team
  update /stats/revision to /stats/revision + 1
}

openRegistration(key examKey) {
  assert  exists /stats/exam[examKey]
  assert !exists /stats/exam[examKey]/free

  insert free at /stats/exam[examKey]
  update /stats/revision to /stats/revision + 1
}

openSignUp(key exerciseId) {
  assert  exists /stats/exercise[exerciseId]
  assert !exists /stats/exercise[exerciseId]/open

  insert open at /stats/exercise[exerciseId]
  update /stats/revision to /stats/revision + 1
}

publishResults(key examKey) {
  assert  exists /stats/exam[examKey]
  assert !exists /stats/exam[examKey]/published

  insert published at /stats/exam[examKey]
  update /stats/revision to /stats/revision + 1
}

registerStudent(key exerciseKey, key studentKey) {
  assert exists /stats/account[studentKey]/student

  assert  exists /stats/exercise[exerciseKey]
  assert !exists /stats/exercise[exerciseKey]/student[studentKey]

  insert student[studentKey] at /stats/exercise[exerciseKey]
  update /stats/revision to /stats/revision + 1
}

removeParticipant(key examKey, key studentKey) {
  assert exists /stats/exam[examKey]/participant[studentKey]
  assert size(/stats/exam[examKey]/participant[studentKey]/result)
           = 0

  delete /stats/exam[examKey]/participant[studentKey]
  update /stats/revision to /stats/revision + 1
}
```

```
removeResult_sheet(key exerciseKey, key studentKey, key sheetKey) {
  assert exists /stats/exercise[exerciseKey]/student[studentKey]/
           result[sheetKey]

  delete /stats/exercise[exerciseKey]/student[studentKey]/
           result[sheetKey]
  update /stats/revision to /stats/revision + 1
}

removeResult_task(key examKey, key studentKey, key taskKey) {
  assert exists /stats/exam[examKey]/participant[studentKey]/
           result[taskKey]

  delete /stats/exam[examKey]/participant[studentKey]/result[taskKey]
}

removeStudentId(key accountKey) {
  assert exists /stats/account[accountKey]/student

  assert !exists /stats/exercise/student[accountKey]
  assert !exists /stats/exam/participant[accountKey]

  delete /stats/account[accountKey]/student
  update /stats/revision to /stats/revision + 1
}

requestReset(key accountKey, string reset) {
  assert exists /stats/account[accountKey]

  assert !exists /stats/account[accountKey]/code

  if !exists /stats/account[accountKey]/reset then
    insert reset at /stats/account[accountKey] using reset
  else
    update /stats/account[accountKey]/reset to reset
  fi

  update /stats/revision to /stats/revision + 1
}

resetPassword(key accountKey, string reset, string password) {
  assert exists /stats/account[accountKey]/reset
  assert /stats/account[accountKey]/reset = reset

  delete /stats/account[accountKey]/reset
  update /stats/account[accountKey]/password to password
  update /stats/revision to /stats/revision + 1
}
```

```
revokeAdminRights(key accountKey) {
  assert exists /stats/account[accountKey]/admin

  assert size(/stats/account/admin) > 1

  delete /stats/account[accountKey]/admin
  update /stats/revision to /stats/revision + 1
}

revokeAssistantRights(key accountKey, key exerciseKey) {
  assert exists /stats/account[accountKey]/assistant[exerciseKey]
  assert exists /stats/exercise[exerciseKey]/assistant[accountKey]

  delete /stats/account[accountKey]/assistant[exerciseKey]
  delete /stats/exercise[exerciseKey]/assistant[accountKey]
  update /stats/revision to /stats/revision + 1
}

revokeExaminerRights(key accountKey, key examKey) {
  assert exists /stats/account[accountKey]/examiner[examKey]
  assert exists /stats/exam[examKey]/examiner[accountKey]

  delete /stats/account[accountKey]/examiner[examKey]
  delete /stats/exam[examKey]/examiner[accountKey]
  update /stats/revision to /stats/revision + 1
}

revokeTutorRights(key accountKey, key exerciseKey, key groupKey) {
  assert exists /stats/account[accountKey]/tutor[exerciseKey]/
           group[groupKey]
  assert exists /stats/exercise[exerciseKey]/group[groupKey]/
           tutor[accountKey]

  delete /stats/account[accountKey]/tutor[exerciseKey]/
           group[groupKey]
  if size(/stats/account[accountKey]/tutor[exerciseKey]/group) = 0
  then
    delete /stats/account[accountKey]/tutor[exerciseKey]
  fi
  delete /stats/exercise[exerciseKey]/group[groupKey]/
           tutor[accountKey]
  update /stats/revision to /stats/revision + 1
}


signOutGroup(key exerciseKey, key studentKey) {
  assert exists /stats/exercise[exerciseKey]/student[studentKey]/
           group
```

```
  update /stats/exercise[exerciseKey]/group[
           /stats/exercise[exerciseKey]/student[studentKey]/group
         ]/curSize
     to (/stats/exercise[exerciseKey]/group[
           /stats/exercise[exerciseKey]/student[studentKey]/group
         ]/curSize - 1)
  delete /stats/exercise[exerciseKey]/student[studentKey]/group
  if exists /stats/exercise[exerciseKey]/student[studentKey]/team
  then
    delete /stats/exercise[exerciseKey]/student[studentKey]/team
  fi
  update /stats/revision to /stats/revision + 1
}

signUpGroup(key exerciseKey, key studentKey, key groupKey) {
  assert  exists /stats/exercise[exerciseKey]/student[studentKey]
  assert  exists /stats/exercise[exerciseKey]/group[groupKey]

  assert !exists /stats/exercise[exerciseKey]/student[studentKey]/
           group
  assert /stats/exercise[exerciseKey]/group[groupKey]/curSize <
         /stats/exercise[exerciseKey]/group[groupKey]/maxSize

  insert group at /stats/exercise[exerciseKey]/student[studentKey]
    using groupKey
  update /stats/exercise[exerciseKey]/group[groupKey]/curSize
     to (/stats/exercise[exerciseKey]/group[groupKey]/curSize + 1)
  update /stats/revision to /stats/revision + 1
}

unregisterStudent(key exerciseKey, key studentKey) {
  assert  exists /stats/exercise[exerciseKey]/student[studentKey]
  assert !exists /stats/exercise[exerciseKey]/student[studentKey]/
                   group
  assert   size( /stats/exercise[exerciseKey]/student[studentKey]/
                   result) = 0

  delete /stats/exercise[exerciseKey]/student[studentKey]
  update /stats/revision to /stats/revision + 1
}

validateAccount(key accountKey, string code) {
  assert exists /stats/account[accountKey]/code

  assert /stats/account[accountKey]/code = code

  delete /stats/account[accountKey]/code
  update /stats/revision to /stats/revision + 1
}
```

```
int getNumberRegisteredExamParticipants(key examKey) {
  return size(/stats/exam[examKey]/participant)
}

int getNumberParticipatedExamParticipants(key examKey) {
  int result = 0
  foreach /stats/exam[examKey]/participant[$x] do
  if size(/stats/exam[examKey]/participant[$x]/result) > 0 then
    result = result + 1
  fi
  done
  return result
}

int getNumberGrades(key examKey, key gradeKey) {
  int lowerBound
  int upperBound
  int bound
  int partSum
  int result = 0

  // min points for requested grade
  lowerBound = /stats/exam[examKey]/grade[gradeKey]/minPoints

  // min points for next better grade, good default here
  upperBound = sum(/stats/exam[examKey]/task/maxPoints) + 1

  // loop over all grades to find upperBound
  foreach /stats/exam[examKey]/grade[$x]/minPoints do
    bound = /stats/exam[examKey]/grade[$x]/minPoints
    if bound > lowerBound && bound < upperBound then
      upperBound = bound
    fi
  done

  // Count participants having points between bounds
  foreach /stats/exam[examKey]/participant[$x] do
    partSum = sum(/stats/exam[examKey]/participant[$x]/result/points)
    if partSum >= lowerBound && partSum < upperBound then
      result = result + 1
    fi
  done
  return result
}

double getAverageTaskPoints(key examKey, key taskKey) {
  return sum(/stats/exam[examKey]/participant/result[taskKey]/points)
      / size(/stats/exam[examKey]/participant/result[taskKey]/points)
}
```

```
double getAverageTaskPointsByGroup(key examKey, key taskKey,
      key groupKey) {
  int groupsum = 0
  int groupcount = 0
  foreach /stats/exam[examKey]/participant[$x] do
    if exists /stats/exercise[/stats/exam[examkey]/exercise]/
                student[$x]/group[groupKey] then
      if exists /stats/exam[examKey]/participant[$x]/
                  result[taskKey] then
        groupsum = groupsum + /stats/exam[examKey]/participant[$x]/
                                result[taskKey]/points
        groupcount = groupcount + 1
      fi
    fi
  done
  assert groupcount > 0 // increase this for better privacy
  return groupsum / groupcount
}
```

B. Abbildungsverzeichnis

C. Listings

D. Literatur

[ATW] Alex, Ben ; Taylor, Luke ; Winch, Rob: *Spring Security Reference*. http://docs.spring.io/spring-security/site/docs/3.2.3.RELEASE/reference/htmlsingle, Abruf: April 2014

[Fec98] Fecht, Peter: *Zugriffskontrolle für Internet-Informationssysteme: Ein Lösungsbeitrag für eine maritime Informationsinfrastruktur (MARINFO)*, Universität Hamburg, Diplomarbeit, November 1998. https://wwwmatthes.in.tum.de/file/t8jji963imta/Publications/1998/Fech98/Fech98.pdf

[Fis12] Fischer, Thomas: *Data Binding for Schemata with Integrity Constraints and Atomic Procedures*, Technische Universität Kaiserslautern, Masterarbeit, August 2012

[GJS+] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad ; Buckley, Alex ; Oracle America, Inc. (Hrsg.): *The Java Language Specification - Java SE 7 Edition*. http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf, Abruf: April 2014

[Ins08] Institute of Electrical Electronics Engineers: *IEEE Standard for Binary Floating-Point Arithmetic*. 2008

[ISO96] International Organization for Standardization (Hrsg.): *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*. 1996

[Mic] Michel, Patrick: *XCend [Transcend]*. https://xcend.de, Abruf: April 2014

[Mic14] Michel, Patrick: *A Formal Framework for Maintaining the Integrity of Structured Data*, Technische Universität Kaiserslautern, Diss., 2014

[Ora] Oracle America, Inc. (Hrsg.): *Java Platform SE 7*. http://docs.oracle.com/javase/7/docs/api, Abruf: April 2014

[PH] Poetzsch-Heffter, Arnd: *Fortgeschrittene Aspekte objektorientierter Programmierung*. https://softech.cs.uni-kl.de/Homepage/FASOOPSS13, Abruf: April 2014

[WW07] Wortman, Felix ; Winter, Robert: Vorgehensmodelle für die rollenbasierte Autorisierung in heterogenen Systemlandschaften. In: *Wirtschaftsinformatik* 49 (2007), Dezember, Nr. 6, S. 439–447. – ISSN 0937–6429