Universität Koblenz-Landau
Institut für Informatik
Dr. Bernhard Beckert
www.uni-koblenz.de/~beckert

WS '02/03

Praktikum

Entwicklung objektorientierter Software mit formalen Methoden

Aufgabenblatt 7

Um sich die Arbeit zu erleichtern, benützen Sie bitte eine neue Version des KeY-Systems, die Sie von der Praktikums-Seite herunterladen können.

Als Basis für die Verifikationsaufgaben finden Sie auf der Praktikums-Webseite ein Archiv mit Eingabe-Dateien für ein Together-Projekt. Laden Sie es herunter, legen Sie ein Verzeichnis fur das Projekt an, legen Sie die Datei dort hinein und entpacken Sie das Archiv mit dem Befehl:

```
gtar zxf verifprojekt.tgz
```

Starten Sie KeY und öffnen Sie das heruntergeladene Together-Projekt `Konto.tpr`. Es umfasst ein Klassendiagramm, in dem verschiedene Arten von Kontos der Berliner Milliardenbank modelliert sind. Diese verschiedenen Arten von Kontos sind als Unterklassen der Klasse `Konto` modelliert, so gibt es beispielsweise Spendenkonten und Spezial-Spenden-Konten, deren Hauptmerkmal eine besonders kurze Kontonummer bzw. eine Kontonummer ist, die zusätzliche besondere Eigenschaften hat. Die Kontonummer eines Kontos ist in einer extra Klasse gekapselt.

Stellen Sie bitte für die erste Aufgabe und die Teilaufgaben 3 und 4 der zweiten Aufgabe sicher, dass folgende Einstellung getroffen ist: Starten Sie den Beweiser, z.B. indem Sie über das Kontextmenü (rechte Maustaste) der Methode `abheben` in der Klasse `Konto` KeYExtensions | PreservesInvariant aufrufen. Stellen Sie unter Options | LDT Models die Einstellung von Numbers von Arithmetic Semantics (prohibiting overflow) auf Arithmetic Semantics (ignoring overflow) und speichern Sie die Einstellungen. Beenden Sie dann den Beweiser.

Aufgabe 1

Bei der objektorientierten Softwareentwicklung ist es ein wichtiges Merkmal, dass die Invarianten einer jeden Unterklasse stärker sind als die Invarianten ihrer Oberklasse. Nur unter dieser Voraussetzung ist sicher, dass sich eine Unterklasse mindestens so verhält wie die Oberklasse und somit die Unterklasse überall da eingesetzt werden kann, wo die Oberklasse erwartet wird (*Substitutionsprinzip*).

Im geöffneten Together-Projekt finden sie die Klasse `SpendenKonto` und ihre Unterklasse `SpezialSpendenKonto`. In beiden sind Invarianten angegeben. Sie sollen nun beweisen, dass das Substitutionsprinzip in dieser Unterklassenbeziehung gilt.

(a) Lesen Sie die beiden Constraints durch und machen Sie sich klar, warum das Substitutionsprinzip gilt.

(b) Wenn Sie mit der Maus auf die Unterklasse `SpezialSpendenKonto` klicken, das Kontextmenü mit der rechten Maustaste aufrufen und im Untermenü der KeYExtensions den Menüpunkt StructuralSubtyping auswählen, wird eine prädikatenlogische Beweisverpflichtung erzeugt und im KeY-Beweiser angezeigt. Machen Sie sich die Aussage dieser Formel klar!

Wie bei Übungsblatt 3 empfiehlt es sich View | Pretty&Untrue auszuwählen. Achtung: Falls Sie im Beweiser eine Formel interaktiv eingeben wollen, dürfen Sie *nicht* diese Notation verwenden. Insbesondere müssen Sie (in den meisten Fällen), bevor Sie mittels *Copy & Paste* eine Formel in ein Eingabefeld eines Instanziierungsdialoges ziehen, View | Pretty&Untrue vorübergehend wieder ausschalten.

(c) Beweisen Sie die erzeugte Beweisverpflichtung mit dem KeY-Beweiser. Sie dürfen dabei die Heuristiken benützten, die Sie in Options | Heuristics auswählen und mit ApplyHeuristics anwenden können. Der Beweis bedarf einiger kleinerer Interaktionen.

Bemerkung: Die Regel `partition_inequation` erzeugt zu einer Formel $i < j$ auf der linken Seite eine Formel $j < k$ auf der rechten Seite, wenn $i < k$ schon rechts steht.

Aufgabe 2

Während in der letzten Aufgabe allein die Modellebene beteiligt war (sog. *horizontale Verifikation*), beschäftigt sich diese Aufgabe mit der Korrektheit der Implementierung (sog. *vertikale Verifikation*).

In der Klasse `Konto` ist die Methode `abheben` implementiert und mit einer OCL-Vorbedingung versehen. Ferner ist in dieser Klasse eine Invariante angegeben.

(a) Lesen Sie sich die Invariante von `Konto` und die Implementierung von `abheben` durch.

(b) In der Implementierung befindet sich ein kleiner Fehler, der die Invariante verletzt. Korrigieren Sie ihn!

(c) Beweisen Sie nun die Korrektheit der Implementierung bezüglich der Invariante mit dem KeY-Beweiser. Klicken Sie dazu auf die Operation `abheben` im Klassendiagramm, rufen das Kontextmenü auf und wählen Sie aus dem Untermenü der KeYExtensions den Menüpunkt PreservesInvariant aus. Machen Sie sich die Aussage der generierten Beweisverpflichtung in Dynamischer Logik klar und beweisen Sie sie mithilfe der Heuristiken.

Bemerkung: Beachten Sie: Um ein Beweisziel zu schließen, kann man auch *false* auf der linken Seite der Sequenz zeigen! Folgende Regeln sind für diese und / oder die nächste Teilaufgabe von Nutzen:

`less_sub`	multipliziert eine Ungleichung auf beiden Seiten mit -1
`skip_negative_int_*`	lässt auf der *kleiner*-Seite einer Ungleichung ein positives Additionsglied weg.

(d) Man kann die Invariante von `Konto` verschärfen, indem man zusätzlich fordert, dass der Kontostand unter einem bestimmten maximalen Kontostand liegt. Fügen Sie am Ende der alten Invariante folgendes hinzu:

```
and kontostand<maxKontoStand and maxKontoStand>0
```

Beweisen Sie nun erneut, dass die Operation `abheben` die Invariante von `Konto` erhält.

(e) Trotz des obigen Beweises erhält `abheben` die Invariante von `Konto` nicht immer!

Benützen Sie die `main`-Methode, die sie mit (F9) ablaufen lassen können, um sich von dieser Tatsache zu überzeugen:

Falls Sie beim Übersetzen eine Fehlermeldung bekommen, setzen Sie unter dem Menüpunkt Tools | Options | Project Level den Wert von Builder | Built-in Javac | Destination directory auf `$PROJECT_DIR$`.

Das Programm erwartet als Eingabe den Betrag, den Sie vom Konto abheben möchten und ruft damit die `abheben`-Methode auf. Machen Sie einige Durchläufe des Programms mit verschiedenen Eingaben. Bei gewissen Eingaben, wie z.B. 2147483647 wird die Invariante verletzt. Es tritt die Eigenschaft des *overflow* von Ganzzahlen in JAVA auf. Machen Sie sich mit dieser Eigenschaft vertraut.[1] Unter welcher Voraussetzung ist Ihr Beweis aus der letzten Teilaufgabe nur gültig? Was meinen Sie: Ist das Problem in der Praxis relevant?

(f) Um die unerwünschten Effekte, wie sie in der letzten Teilaufgabe entstanden sind, zu vermeiden, bietet der KeY-Beweiser die Möglichkeit an zu beweisen, dass in einem Programm kein *overflow* auftreten kann.

Rufen Sie dazu den KeY-Beweiser noch einmal auf und ändern Sie unter Options | LDT Models die Einstellung von Numbers von Arithmetic Semantics (ignoring overflow) auf Arithmetic Semantics (prohibiting overflow) und speichern Sie die Einstellungen. Beenden Sie dann den Beweiser.

Bemerkung: Im folgenden werden in den Beweisen Prädikate der Form $inInt(x)$ auftauchen. Das Prädikat $inInt(x)$ ist eine Abkürzung für

$$x >= \texttt{Integer.MIN\_VALUE} \wedge x <= \texttt{Integer.MAX\_VALUE},$$

wobei die Konstanten `Integer.MIN_VALUE` bzw. `Integer.MAX_VALUE` den kleinsten bzw. größten Wert der Typs `Integer` enthalten. Informell ausgedrückt wird dieses Prädikat benutzt um zu beweisen, dass der Wert des (evtl. komplexen) Terms x den Wertebereich des Typs `Integer` nicht überschreitet und also kein *overflow* verursacht.

Um Ihnen die Aufgabe zu erleichtern, sollen Sie das folgende auf der Klasse `Konto2` durchführen. Diese Klasse ist wie `Konto` aufgebaut, aber einige Attribute fehlen und die Invariante ist anders.

Versuchen Sie zu beweisen, dass die Methode `abheben` von `Konto2`, die im wesentlichen gleich wie die in `Konto` aufgebaut ist, die Invariante nicht verletzt. Lässt sich der Beweis schließen? Wenn nicht, schauen Sie sich die offenen Ziele an und erläutern Sie, warum sich der Beweis nicht schließen lässt.

(g) Die Klasse `Konto2` enthält noch eine weitere Methode `sicherAbheben`. Diese Methode ist so implementiert, dass kein *overflow* mehr auftreten kann.

Versuchen Sie mit dem KeY-Beweiser zu zeigen, dass `sicherAbheben` die Invariante nicht verletzt und dass kein *overflow* auftreten kann.[2] Wenn man durch Anwendung von Heuristiken nicht mehr weiterkommt, erkennt man, dass noch zwei Ziele offen sind. Da es nicht ganz einfach ist, den Beweis zu schließen, reicht es aus, informell zu argumentieren, warum beide Ziele geschlossen werden können (wer besonders motiviert ist, kann natürlich auch versuchen, den Beweis mit dem KeY-System zu schließen).

---

[1] Beispielsweise mithilfe von `http://www.artima.com/underthehood/logic.html`

[2] Achten Sie darauf, dass Options | LDT Models immer noch auf *Arithmetic Semantics (prohibiting overflow)* gesetzt ist.