Überprüfung von UML/OCL-Spezifikationen mit Prädikatenlogikbeweisern: eine Fallstudie

Eine Bachelorarbeit von

Sandra von Cube

Matrikelnummer 28826

Studiengang IT

12. Oktober 2006

Betreuender Hochschulprofessor: Prof. Dr. Ralf Möller
Betreuer: Miguel Garcia

Institut für Softwaresysteme
Technische Universität Hamburg-Harburg

In dieser Bachelorarbeit stelle ich Überlegungen an, wie es möglich sein könnte, automatische Verfahren zur Verifikation von Software zu erstellen. Dazu wandle ich zum Einen UML-Diagramme in die Sprache TLA^+ um und überprüfe die erstellten Spezifikationen mit dem Model-Checker TLC. Von den Resultaten ausgehend beschreibe ich eine Möglichkeit zur Implementierung. Zum Anderen wandle ich die Diagramme in Prädikatenlogik um und überprüfe sie mit mit dem Theorembeweiser OTTER, der das Verfahren der Deduktion benutzt. Da sich dieses Verfahren jedoch nicht gut für eine Automatisierung eignet, gebe ich hier keinen Ausblick auf eine mögliche Implementierung.

Erklärung

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel oder Quellen dazu verwendet habe.

Hamburg, den 12. Oktober 2006
Sandra von Cube

Inhaltsverzeichnis

Zusammenfassung ii
Abbildungsverzeichnis v
Tabellenverzeichnis vi

1 Einführung 1
1.1 Motivation 1
1.2 Aufgabe 2
1.3 Vorgehensweise 3

2 Model-Checking 4
2.1 Die Sprache TLA$^+$ 4
2.1.1 Allgemeines über TLA$^+$ 4
2.1.2 Einführung in TLA$^+$ 5
2.2 Umformung von UML/OCL-Spezifikationen nach TLA$^+$ 13
2.2.1 Umformung der UML nach TLA$^+$ 13
2.2.2 Umformung der OCL nach TLA$^+$ 17
2.3 Überprüfung mit dem Model-Checker TLC 20
2.3.1 Allgemeines über TLC 20
2.3.2 Einführung in TLC 20
2.4 Resultate 22
2.5 Eine mögliche Implementierung 26
2.5.1 Voraussetzungen 26
2.5.2 Funktionsweise 27
2.5.3 mögliche Erweiterungen 27

3 Deduktion 29
3.1 Otter 29
3.1.1 Allgemeines über Otter 29
3.1.2 Einführung in die Syntax von Otter 30
3.2 Umformung der UML nach Otter 32
3.3 Überprüfung 33
3.4 Resultate 34

4 Zusammenfassung 37
4.1 Resultate 37
4.2 Ausblick 38

A TLA$^+$ Tools 41
A.1 Installation der TLA$^+$ Tools 41
A.2 Benutzung von TLC 41
A.3 Benutzung von TLATeX 42

B Otter **43**
B.1 Installation von Otter 43
B.2 Benutzung von Otter 43

C TLA$^+$-Eingabedateien **45**

D Otter-Eingabedateien **72**

E Verzeichnisstruktur der CD **86**

Abbildungsverzeichnis

1 Modul HourClock in der ASCII-Version 11
2 Modul HourClock in der Typeset-Version 11
3 UML-Diagramm Italians 14
4 UML-Diagramm Persons 15
5 UML-Diagramm Flights 17
6 UML-Diagramm Driver 18
7 UML-Diagramm Person 18
8 Die Ausgabe der Überprüfung der Datei GenOverIncom.tla 22
9 Die Ausgabe der Überprüfung der Datei Persons.tla 23
10 Funktion packageBefore im Visitor PackageToString 27
11 Funktion packageBefore im Visitor PackageToTLA 28
12 Verzeichnisstruktur der CD 86

Tabellenverzeichnis

1 Operatoren der Prädikatenlogik 5
2 Operatoren der Mengenlehre 5
3 Operatoren der temporalen Logik 6
4 Die wichtigsten TLA^+-Operatoren in der Typeset- und ASCII-Version 8
5 vordefinierte Operatoren der OCL Quelle [Sch06OCL] 19
6 gemessene Zeiten TLC 24
7 Operatoren der Prädikatenlogik in der OTTER-Syntax 31
8 benötigte Zeiten Otter 35

1 Einführung

1.1 Motivation

In der heutigen Zeit spielt die Automatisierung der Softwareherstellung eine immer größere Rolle und es ist anzunehmen, dass sie in der Zukunft eine noch bedeutendere Rolle spielen wird. *Model Driven Architecture* (MDA) und *Model Driven Software Development* (MDSD) sind dabei Begriffe, die häufig zu hören sind.

Die Model Driven Architecture ist ein von der *Object Management Group* (OMG) entwickelter Rahmen (*framework*) zur Entwicklung von modellgetriebener (*model driven*) Software. Der Prozess der MDA besteht dabei aus drei Schritten:

1. Erstellung eines plattformunabhängigen Modells (PIM - *Platform Independent Model*) auf einer hohen Abstraktionsebene, welches komplett unabhängig von beliebigen Technologien ist.

2. Transformation des PIMs in ein oder mehrere plattformabhängige Modelle (*Platform Specific Model* - PSM). Ein PSM ist direkt von der Technologie und Implementierung der Plattform abhängig. Hier liegt der größte Nutzen von MDA, denn dieser Schritt war bisher kompliziert und mit viel Arbeit verbunden.

3. Transformation des PSMs in Code. Dieser Schritt ist relativ einfach, denn das PSM ist bereits plattformspezifisch.

Die Transformationen werden von Werkzeugen ausgeführt. Dabei ist neu, dass die Transformation von einem PIM zu einem PSM auch durch Werkzeuge ausgeführt wird.
Ziele von MDA sind Produktivität, Portabilität, Interoperabilität und Dokumentation auf einem hohen Abstraktionsniveau. Auf längere Sicht kommt die Standardisierung von Modellen hinzu. Voraussetzungen dafür sind jedoch eine ausdrucksstarke, eindeutig definierte und standardisierte Sprache, um plattformunabhängige Modelle zu beschreiben, und eine formale, von Werkzeugen ausführbare Definitionssprache, um die Transformation von einem PIM zu einem PSM beschreiben zu können. Ausführlichere Informationen zu der *Model Driven Architecture* sind in [KlWaBa04] zu finden.

Die Grundzüge vom MDSD sind denen von der MDA sehr ähnlich. Der Prozess des MDSDs besteht aus den gleichen drei Schritten. MDA nimmt jedoch teilweise Einschränkungen vor, zum Beispiel eine Fokussierung auf UML-basierte Modellierungssprachen, die im MDSD nicht vorgenommen werden. Zu den Zielen der MDA kommen bei dem MDSD noch die Erhöhung der Qualität

der entwickelten Software, eine bessere Handhabbarkeit von Komplexität durch Abstraktion, die Erhöhung der Möglichkeiten zur Wiederverwendbarkeit und eine bessere Wartbarkeit durch die Vermeidung von Redundanzen hinzu. Einen genaueren Vergleich und weitere Informationen über das *Model Driven Software Development* können in [StVö05] nachgelesen werden.
Im Allgemeinen ist es richtig zu sagen, dass die *Model Driven Architecture* der OMG eine Standardisierungsinitiative für das *Model Driven Software Development* ist.

Da jedoch auch in automatisch erzeugter Software Fehler enthalten sein können, muss auch diese überprüft werden. Es bietet sich an, auch diesen Schritt zu automatisieren. Dabei ist es am sinnvollsten, das PIM zu überprüfen, denn es ist das Modell, von dem ausgegangen wird.
Ein Verfahren zur Überprüfung, oder auch Verifikation genannt, von Systembeschreibungen ist das *Model-Checking*. Dabei wird das Modell in eine formale Sprache umgewandelt und schließlich vollautomatisch geprüft.
Eine andere Möglichkeit ist die *Deduktion*. Auch hier wird die Systembeschreibung in Formeln aufgestellt und mit Hilfe der Resolution überprüft. Da bei der Resolution nur die Unerfüllbarkeit einer Formelmenge bewiesen werden kann, muss mit der Negation der zu beweisenden Formel gearbeitet werden.

In dieser Bachelorarbeit stelle ich einige Überlegungen an, wie es möglich wäre, ein Testverfahren für automatisch erstellte Software zu entwerfen und zu implementieren. Dabei gehe ich eher von dem Ansatz der MDA aus, da auch ich mich auf die Nutzung eines Modells in Form eines UML-Diagramms zusammen mit OCL-Constraints beschränke. Da beide Sprachen von der OMG als Standard festgelegt wurden und zusammen sehr ausdrucksstark sind, sehe ich dies aber nicht unbedingt als Nachteil an. Auch spielt der Fakt, dass andere standardisierte Sprachen, die sich für MDSD eigen würden, noch fehlen, eine bedeutende Rolle. Mit Hilfe einiger einfacher UML-Diagramme und der Verfahren des Model-Checkings und der Deduktion überprüfe ich schließlich meine Überlegungen und versuche, Schlüsse für ein automatisiertes Testverfahren daraus zu ziehen.

1.2 Aufgabe

Aufgabe meiner Bachelorarbeit ist das Umformen von UML-Klassendiagrammen und den dazugehörigen OCL-Constraints in die von Leslie Lamport entwickelte Sprache TLA$^+$. Die generierten TLA$^+$-Formeln überprüfe ich schließlich mit dem Model-Checker TLC. Parallel dazu forme ich die Diagramme in die Syntax des Beweisers OTTER um und überprüfe auch sie. Dabei ist mein Ziel herauszufinden, ob diese Programme dazu fähig sind, die umgeformten Diagramme zu verifizieren.

1.3 Vorgehensweise

Beide der ausgesuchten Verfahren zur Verifikation, also das Model-Checking und die Deduktion, behandle ich in einem eigenen Kapitel.
Ich beginne mit dem Model-Checking in Kapitel 2. Am Anfang des Kapitels gebe ich eine kurze Einführung in das Verfahren. Im ersten Unterabschnitt nenne ich einige allgemeine Informationen über die benutzte Sprache TLA^+ und beschreibe sie näher. In Kapitel 2.2 erläutere ich die Umformung von UML-Diagrammen in die Sprache TLA^+. Kapitel 2.3 handelt von der Überprüfung der erstellten TLA^+-Dateien mit dem Model-Checker TLC. Auch hierzu sind einige allgemeine Informationen und eine kurze Einführung in den Kapiteln 2.3.1 und 2.3.2 nachzulesen. In Kapitel 2.4 stelle ich die Ergebnisse dar. Schließlich gehe ich im letzten Unterabschnitt auf eine mögliche Implementierung zur Automatisierung der Umformung von UML-Diagrammen in die Sprache TLA^+ ein.
Das Verfahren der Deduktion behandle ich in Kapitel 3. Auch hier gebe ich zu Beginn eine Einführung in das Verfahren. In Kapitel 3.1 beschreibe ich das verwendete Programm OTTER und dessen Syntax. Auf die Umformung von UML-Diagrammen in die Syntax von OTTER gehe ich in Kapitel 3.2 ein. Eine Zusammenstellung der Ergebnisse der Überprüfung mit dem gewählten Programm sind in Kapitel 3.4 gegeben.
Meine gesamte Arbeit fasse ich schließlich in Kapitel 4 zusammen und gebe einen Ausblick auf mögliche Verbesserungen und Erweiterungen.
Im Anhang habe ich alle erstellten TLA^+- sowie OTTER-Eingabedateien aufgelistet. Zusätzlich ist dort eine kurze Anleitung zur Installation und Benutzung der von mir verwendeten Programme zu finden.

2 Model-Checking

Beim Simulieren und Testen von Software können zwar Fehler entdeckt werden, es gibt aber keine Garantie, dass keine weiteren Fehler existieren. Dies folgt aus der Gegebenheit, dass nur Untermengen von möglichen Systemabläufen getestet werden. Beim Verifizieren hingegen werden alle möglichen Systemzustände betrachtet. Techniken, die dies automatisch durchführen, werden Verfahren zum Model-Checking genannt. Beim Model-Checking wird eine Systembeschreibung vollautomatisch überprüft. Die Systembeschreibung muss dazu in einer formalen Sprache vorliegen, die von dem Model-Checker interpretiert werden kann. Dieser kann dann ohne Mithilfe des Benutzers die Systembeschreibung prüfen. Der Model-Checker kann Korrektheit feststellen oder aber ein Gegenbeispiel bei einem Widerspruch liefern.

Eine Sprache, in der Systembeschreibungen erstellt werden können, ist TLA$^+$. Für sie gibt es einen Model-Checker namens TLC. Beide wurden von Leslie Lamport und anderen entwickelt.

2.1 Die Sprache TLA$^+$

2.1.1 Allgemeines über TLA$^+$

TLA$^+$ ist eine Sprache, um Systeme zu beschreiben. Sie wurde in den späten 80-er Jahren von Leslie Lamport entwickelt.
1977 spezifizierte Amir Pnueli zum ersten Mal Systeme mit temporaler Logik, indem er das erlaubte Verhalten der Systeme beschrieb. Leslie Lamport entwickelte daraus TLA, the Temporal Logic of Actions.
TLA basiert auf mathematischen Logikgrundlagen: der Prädikatenlogik und der Mengenlehre. Leslie Lamport führte zusätzliche Notationen zum Aufstellen von langen Formeln und die Möglichkeit zur Modularisierung ein. All dies führte zu der Sprache TLA$^+$. Mit ihrer Hilfe können die verschiedensten diskreten Systeme präzise und formal beschrieben werden.
In seinem Buch 'Specifying Systems' [Lam02] beschreibt Leslie Lamport, wie es zu der Entwicklung der Sprache TLA$^+$ kam und wie sie zu gebrauchen ist. Es ist möglich, dieses Buch kostenlos von seiner Homepage im Internet

http://research.microsoft.com/users/lamport/tla/book.html

herunterzuladen.

2.1.2 Einführung in TLA$^+$

Da TLA$^+$ auf der Prädikatenlogik basiert, können alle Operatoren aus der Prädikatenlogik benutzt werden. Diese Operatoren sind in der Tabelle 1 aufgelistet.

Operator	Bezeichnung	Bedeutung
$\wedge$	Konjunktion	UND
$\vee$	Disjunktion	ODER
$\neg$	Negation	NICHT
$\Rightarrow$	Implikation	IMPLIZIERT
$\equiv$	Äquivalenz	IST ÄQUIVALENT ZU
$\forall$	Allquantor	FÜR ALLE
$\exists$	Existenzquantor	ES EXISTIERT

Tabelle 1: Operatoren der Prädikatenlogik

Aus der Mengenlehre kommen weitere Operatoren hinzu, die in Tabelle 2 aufgeführt sind.

Operator	Bezeichnung	Bedeutung
$\cap$	Durchschnitt	DAS ELEMENT IST IN BEIDEN MENGEN ENTHALTEN
$\cup$	Vereinigung	DAS ELEMENT IST IN EINER DER BEIDEN MENGEN ENTHALTEN
$\subseteq$	Teilmenge	ERSTERE MENGE IST IN ZWEITER ENTHALTEN
$\setminus$	Differenz	DAS ELEMENT IST IN ERSTERER, ABER NICHT IN ZWEITER MENGE ENTHALTEN

Tabelle 2: Operatoren der Mengenlehre

Um den zeitlichen Aspekt mit in den Formeln ausdrücken zu können, bediente sich Leslie Lamport der Operatoren der temporalen Logik, die in Tabelle 3 dargestellt sind.
Die Operatoren der temporalen Logik werden hauptsächlich zur Aufstellung von Liveness- und Fairness-Bedingungen benötigt. Liveness-Bedingungen beschreiben, dass etwas geschehen muss. Es wird damit die Situation verhindert, dass sich ein System immer nur in ein und demselben Zustand befinden kann. Mit Fairness-Bedingungen wird Gerechtigkeit beschrieben. Dies könnte zum Beispiel die Gleichbehandlung von *next-state*-Formeln sein.

Operator	Bezeichnung	Bedeutung
$\Box$	Box	IMMER
$\Diamond$	Diamant	IRGENDWANN
$\leadsto$	führt zu	$\Box(F \Rightarrow \Diamond G)$

Tabelle 3: Operatoren der temporalen Logik

Zusätzlich bediente sich Leslie Lamport der Idee der Modularisierung aus Programmiersprachen, um Beschreibungen komplexer Systeme übersichtlich und wiederverwendbar gestalten zu können. Auch aus dem Gebiet der Programmiersprachen verwendet er das IF/THEN/ELSE-Konstrukt und die Fallunterscheidung mit Hilfe des Befehls CASE.

Anhand eines einfachen Beispiels, welches auch Leslie Lamport in seinem Buch 'Specifying Systems' [Lam02] benutzt, ist es leicht nachzuvollziehen, wie die Beschreibung eines Systems in TLA$^+$ aufgebaut wird. Beschrieben wird in diesem Beispiel eine Uhr, die jedoch nur die Stunden anzeigt.

Eine Beschreibung eines Systems wird in TLA$^+$ in einem oder mehreren Modulen zusammengefasst. Der Name der Datei, in der ein Modul enthalten ist, muss identisch mit dem Namen des Moduls sein, wobei die Dateiendung *.tla* lautet. In diesem Fall ist der Dateiname HourClock.tla und der Modulname HourClock.

Es gibt zwei Möglichkeiten ein Modul darzustellen. Die erste Möglichkeit ist die ASCII-Version. Sie kann mit einem einfachen Texteditor geschrieben werden. Die Symbole der Operatoren müssen dabei umschrieben werden. Meist wurde versucht, die Symbole bildhaft darzustellen. War dies nicht möglich, wurden die TeX -Befehle benutzt. Eine Auflistung der ASCII-Schreibweisen ist in Tabelle 4 zu finden. Die zweite Möglichkeit ist die Typeset-Version. Dies ist quasi die „normale“ Darstellung mit Symbolen wie $\in$, $\subseteq$ oder $\neg$. Sie wird mit Hilfe des Programmes TLATeX erstellt. Dazu wird die ASCII-Version konvertiert und es entsteht eine LaTeX -Datei, aus der eine *.pdf*-Datei generiert werden kann.

Ein Modul beginnt mit

MODULE *HourClock*

geschrieben in der Typeset-Version oder mit

```
-------- MODULE HourClock --------
```

in der ASCII-Version, wobei hier je mindestens viermal ein '–' geschrieben werden muss. In der ASCII-Version werden alle Befehle in normalen Großbuchstaben geschrieben, im Gegensatz zu der Typeset-Version, in der die Befehle in Kapitälchen geschrieben werden.

Sollen in ein Modul Module aus anderen Dateien eingebunden werden, muss dies an erster Stelle in dem Modul geschehen. Der Befehl dazu lautet EXTENDS. Dies wird zum Beispiel zum Einbinden der arithmetischen Operatoren wie +, -, * und / benötigt, die in TLA$^+$ selber nicht definiert sind, sondern im Modul *Naturals*. In einem Modul würde dies wie folgt aussehen:

EXTENDS *Naturals*

Es gibt noch weitere, vordefinierte Module, wie zum Beispiel das Modul *FiniteSets* mit zwei Operationen auf endlichen Mengen, das Modul *Sequences*, welches Operationen auf Sequenzen definiert, oder die Module *Integer* und *Real*.

An zweiter Stelle werden Variablen und Konstanten deklariert. Dazu werden die Befehle VARIABLE und CONSTANT verwendet. Synonym können auch die Befehle VARIABLES und CONSTANTS benutzt werden, wenn mehrere Variablen oder Konstanten deklariert werden sollen. Mit der Anweisung

VARIABLE *hr*

wird die Variable *hr* deklariert. Sie stellt in diesem Beispiel die Stunden dar.

Auf die Variablen- und Konstantendeklaration folgen die Invarianten. Sie müssen in jedem Zustand des Systems erfüllt sein. Ist dies nicht der Fall, erfüllt der Zustand nicht die Beschreibung des Systems. Im Beispiel der Uhr lautet die Invariante *HCinv*

$$HCInv \triangleq hr \in (1..12)$$

geschrieben in der Typeset-Version oder

```
HCInv == hr \in (1 .. 12)
```

geschrieben in der ASCII-Version. Sie definiert den Wertbereich der Variablen *hr*. In diesem Fall darf *hr* nur einen ganzzahligen Wert zwischen 1 und 12 annehmen. Der Operator '..' ist im Modul *Naturals* definiert. Das Symbol $\triangleq$ bedeutet *ist definiert als* und wird in der ASCII-Version mit '==' dargestellt. Weiter Operatoren sind jeweils in der Typeset- und ASCII-Version in Tabelle 4 abgebildet.

Operator	ASCII-Schreibweise	Operator	ASCII-Schreibweise
$\land$	`/\` oder `\land`	$\cap$	`\cap` oder `\intersect`
$\lor$	`\/` oder `\lor`	$\cup$	`\cup` oder `\union`
$\neg$	`\neg`, `\lnot` oder `~`	$\exists$	`\E`
$\triangleq$	`==`	$\forall$	`\A`
$\neq$	`#` oder `/=`	$'$	`'`
$\equiv$	`<=>` oder `\equiv`	$\ll$	`\ll`
$\Rightarrow$	`=>`	$\gg$	`\gg`
$\in$	`\in`	$\langle$	`<<`
$\notin$	`\notin`	$\rangle$	`>>`
$\Box$	`[]`	$]_v$	`]_v`
$\Diamond$	`<>`	┌──────	`------`
$\leadsto$	`~>`	──────┐	`------`
$\subseteq$	`\subseteq`	├──────┤	`------`
$\subset$	`\subset`	└──────┘	`======`

Tabelle 4: Die wichtigsten TLA$^+$-Operatoren in der Typeset- und ASCII-Version

Nun fehlt noch die Beschreibung der möglichen Zustandsübergänge. Sie werden in einer oder mehreren *next-state*-Relation/en beschrieben. Eine *next-state*-Relation ist eine Formel, die die Beziehung von einem Zustand zum nächsten Zustand ausdrückt. In ihr sind Variablen und Variablen mit Strichindex enthalten. In dem Beispiel mit der Uhr entspricht dies den Variablen *hr* und *hr'*, wobei *hr'*, also die Variable mit dem Strichindex, den Wert in dem Folgezustand darstellt. Die Anforderung, dass die Uhr nur jeweils um eine Stunde weiterspringen darf und von '12' auf auf '1' springen soll, kann mit folgender Formel *HCNext* ausgedrückt werden

$$HCNext \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$$

beziehungsweise in der ACSII-Version mit der Formel

```
HCNext == hr' = IF hr # 12 THEN hr+1 ELSE 1
```

Die Tatsache, dass es einmal im Jahr vorkommt, dass die Uhr um zwei Stunden nach vorne springt anstelle von nur einer, nämlich bei der Umstellung auf die Sommerzeit im Frühling, wird hier vernachlässigt. Eine solche Formel wird eine *Aktion* (*action*) genannt. Ein *Schritt* (*step*), der diese Aktion erfüllt, wird als *HCNext-Schritt* (*HCNext-step*) beschrieben.

Da die Stundenanzahl einer Uhr aber auch gleichbleiben kann, muss diese Eigenschaft auch in den Formeln festgehalten werden können. Dazu führte Lamport *Stotterschritte* (*stuttering steps*) ein. Bei einem Stotterschritt darf die Variable unverändert bleiben, das heißt

$$hr' = hr$$

darf gelten. In TLA$^+$ steht dafür die Notation

$$[HCNext]_{hr}$$

zur Verfügung. Ausgeschrieben bedeutet sie

$$HCNext \lor (hr' = hr)$$

Es darf also entweder ein *HCNext-Schritt* oder ein *Stotterschritt* stattfinden.
Um als Spezifikation nicht eine Menge von Formeln angeben zu müssen, können die einzelnen Funktionen in einer neuen Funktion zusammengefasst werden. Dabei soll der Anfangswert von *hr* die Bedingung *HCInv* erfüllen und es muss auch angegeben werden, dass jeder Schritt des Systems entweder ein *HCNext-Schritt* oder ein *Stotterschritt* sein soll. Dies kann durch die Formel *HC*

$$HC \triangleq HCInv \land \Box[HCNext]_{hr}$$

ausgedrückt werden.

Um ein Modul übersichtlicher zu gestalten, gibt es die Möglichkeit, Trennlinien einzufügen. Sie dienen nur der Optik und haben keine Bedeutung. In der Typeset-Version werden sie mit

├──┤

und in der ASCII-Version mit

```
-----------------------------------------
```

geschrieben, wobei bei der ASCII-Version mindestens vier '-' geschrieben werden müssen.

Bisher wurde definiert, dass *hr* einen ganzzahligen Wert zwischen 1 und 12 annehmen darf. Dies wurde bisher aber nur für den Anfangszustand festgelegt. Diese Bedingung soll jedoch zu jeder Zeit für das gesamte System gelten.

$$\Box HCInv$$

beschriebt diese Anforderung. Die Formel

$$HC \Rightarrow \Box HCInv$$

soll also von jedem Zustand des Systems erfüllt werden. Eine solche temporale Formel, die von jedem Zustand erfüllt werden soll, wird *Theorem* genannt. Sie folgt aus den Anforderungen, die in der Spezifikation beschrieben wurden. Das Theorem wird am Ende eines Moduls angegeben.

$$\textsc{theorem}\ HC \Rightarrow \Box HCInv$$

beziehungsweise mit

```
THEOREM HC => []HCInv
```

Das Modul selber wird mit

---

in der Typeset-Version oder mit

```
=========================================
```

in der ASCII-Version beendet.
Die komplette Beschreibung der HourClock sind in Abbildung 1 und in Abbildung 2 dargestellt. Der Aspekt, dass die Uhr nur wirklich einmal in der Stunde weiterspringt, wurde in dieser Spezifikation noch nicht beachtet.

Für beliebige Systeme gibt es verschiedene Möglichkeiten, eine Spezifikation zu schreiben. So könnte die Formel

$$HCnxt \triangleq hr' = \textsc{if}\ hr \neq 12\ \textsc{then}\ hr + 1\ \textsc{else}\ 1$$

auch durch die Formel

$$HCNext \triangleq hr' = (hr\%12) + 1$$

ersetzt werden, ohne dass sich in der Anforderung etwas ändert.

Werden in einer Formel mehrere Teilformeln durch Konjunktionen oder Disjunktionen miteinander verbunden, ist darauf zu achten, dass sich die Einrückun-

```
---------------------- MODULE HourClock ----------------------
EXTENDS Naturals
VARIABLE hr
HCInv  ==  hr \in (1 .. 12)
HCNext == hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC == HCInv /\ [][HCNext]_hr
--------------------------------------------------------------
THEOREM  HC => []HCInv
==============================================================
```

Abbildung 1: Modul HourClock in der ASCII-Version

MODULE *HourClock*

EXTENDS *Naturals*
VARIABLE *hr*

$HCInv \quad \triangleq \quad hr \in (1 \,..\, 12)$
$HCNext \quad \triangleq \quad hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$
$HC \quad \triangleq \quad HCInv \land \Box[HCNext]_{hr}$

THEOREM $HC \Rightarrow \Box HCInv$

Abbildung 2: Modul HourClock in der Typeset-Version

gen auf gleicher Höhe befinden. So können Klammern weggelassen werden. Ist dies nicht der Fall, kann es zu Problemen führen. Im Zweifelsfall oder wenn die gesamte Formel in eine Zeile geschrieben wird, sollten jedoch immer Klammern gesetzt werden.
So kann die Formel

$$A \land B \land (C \lor D \lor (E \land F))$$

ganz ohne Klammern als

```
/\ A
/\ B
/\ \/ C
   \/ D
   \/ /\ E
      /\ F
```

geschrieben werden.

Um Kommentare einzufügen gibt es zwei Möglichkeiten. Die erste ist ein Kommentar, der nur bis zum Zeilenende reicht. Er beginnt mit `*`. Die andere Möglichkeit wird benutzt, um ganze Teile von Spezifikationen auszukommentieren. Diese Teile selber dürfen auch Kommentare enthalten. Diese Art von Kommentar wird mit `(*` eingeleitet und endet mit `*)`. Mit dieser Art ist es aber auch möglich, nur einen Teil einer Zeile auszukommentieren, so dass der Rest der Zeile, der hinter dem Kommentar steht, wieder zur Spezifikation gehört.

Neben den hier beschriebenen Möglichkeiten gibt es noch andere, um Spezifikationen zu erstellen. So gibt es zum Beispiel die Möglichkeit, mehrere Variablen durch eine auszudrücken. Dazu werden die Variablen in einem *Kanal* (*channel*) gespeichert. Innerhalb dieses Kanals ist die Reihenfolge jedoch beliebig. Es kann über die Namen der einzelnen Variablen auf sie zugegriffen werden. Eine andere, bisher nicht erwähnte Möglichkeit ist die Instantiierung. Informationen über diese und alle weiteren, bisher hier nicht erklärten Möglichkeiten und Konstrukte, werden in [Lam02] erläutert.

2.2 Umformung von UML/OCL-Spezifikationen nach TLA$^+$

2.2.1 Umformung der UML nach TLA$^+$

Insgesamt habe ich fünf *.tla*-Dateien erstellt. In jeder Datei ist als Kommentar das dazugehörige Diagramm abgebildet. Die Datei Generalisierung.tla repräsentiert zwei Klassen, wobei die Klasse B von der Klasse A vererbt wurde. Das zu der Datei *GenOverIncom.tla* (Generalisierung overlapping, incomplete) gehörende Diagramm umfasst drei Klassen. Klasse A ist die Oberklasse, von denen die Klassen B und C vererbt werden. An die Vererbung ist die Bedingung `{overlapping, incomplete}` geknüpft. Das Diagramm der Datei *GenDisCom.tla* entspricht dem der Datei *GenOverIncom.tla*, nur dass hier an die Vererbung die Bedingung `{disjoint, complete}` geknüpft ist.
Die Datei *Italians.tla* repräsentiert ein Diagramm, welches fünf Klassen umfasst. Von der Klasse *Italian* werden die Klassen *Lazy*, *Mafioso* und *LatinLover* als `{disjoint, complete}` vererbt. Des weiteren wird von der Klasse *Italian* die Klasse *ItalianProf* vererbt. Als zusätzliche Bedingungen sollen die Klassen *Mafioso* und *Lazy* jeweils mit der Klasse *ItalianProf* disjunkt (`disjoint`) sein. Aus dem Diagramm folgt also, dass alle *ItalianProf* *LatinLover* sind. ist. Das Diagramm ist in Abbildung 3 dargestellt.
Die letzte Datei ist die Datei *Persons.tla*. Hier gibt es insgesamt sieben Klassen. Die oberste Klasse ist die Klasse *Person*. Von ihr werden zwei Klassen als `disjoint` vererbt. Das sind die Klassen *Italian* und *English*. Von der Klasse *Italian* werden wieder zwei Klassen, die Klasse *Lazy* und die Klasse *LatinLover*, vererbt. Diesmal als `{disjoint, complete}`. Von der Klasse *English* werden auch zwei Klassen vererbt und zwar die Klassen *Gentleman* und *Hooligan*. Zusätzlich gibt es noch eine weitere Vererbung, die besagt, dass die Klasse *LatinLover* von der Klasse *Gentleman* vererbt wurde. Das dazugehörige Diagramm ist in Abbildung 4 zu sehen.

Die Klassen der UML-Diagramme habe ich als Mengen repräsentiert. Die Elemente, die schon bearbeitet wurden, werden einer Untermenge *allInstances_X* der entsprechenden Menge X hinzugefügt. So kann auch kontrolliert werden, ob alle der vorhandenen Elemente bearbeitet wurden.
Für jede Klasse gibt es Funktionen, die Elemente zu ihrer Untermenge hinzufügen oder entfernen. Ausnahmen bilden abstrakte Klassen. Da es von diesen Klassen selber keine Instanzen gibt, existieren auch keine Funktionen, um Elemente hinzuzufügen oder zu entfernen.

Bei einigen Klassen sind die Mengen *allInstances* keine Untermengen, sondern Obermengen. Die ursprüngliche Menge ist also eine Untermenge der Menge *allInstances*. Das ist bei Generalisierungen der Fall. Ein Beispiel hierfür ist in der Datei *Generalisierung.tla* zu finden. Bereits bei der Invariante ist angegeben, dass

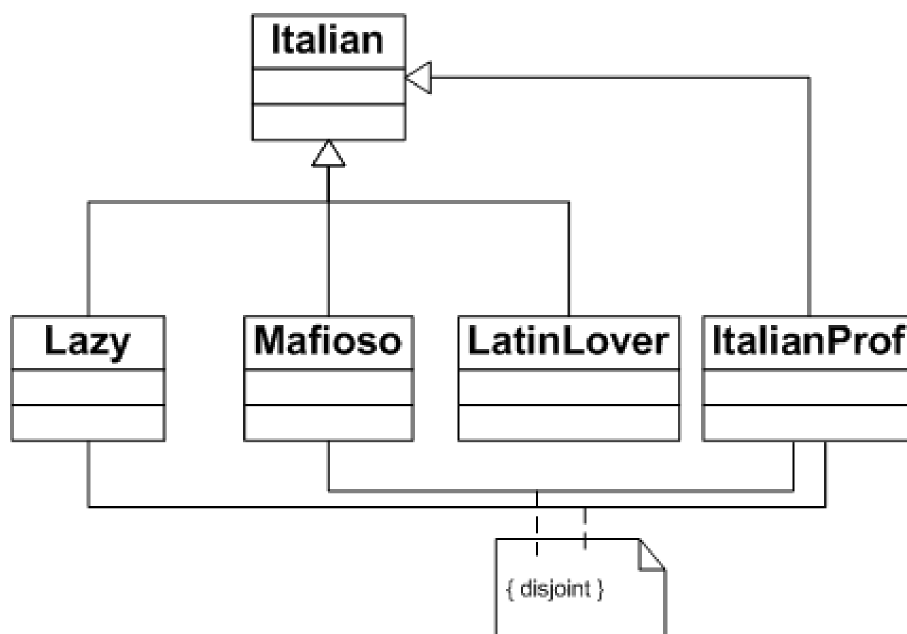


Abbildung 3: UML-Diagramm Italians

die Menge *allInstances_packname_A* eine Teilmenge der Vereinigung der Mengen A und B sein soll. Da B eine Subklasse von A darstellt, sind alle ihre Elemente auch in der Menge A enthalten und werden somit auch der Menge *allInstances_packname_A* hinzugefügt. Daher ist die Menge *allInstances_packname_A* größer als die Menge A. Sind alle Elemente beider Mengen in der Menge *allInstances_packname_A* enthalten, so muss diese Menge gleich der Vereinigung der Mengen A und B sein.

Die Dateien sind vom Aufbau her alle gleich. Zu Beginn werden die Mengen und Variablen deklariert. Die Mengen wurden als konstant (CONSTANT) deklariert und entsprechen den Klassen. Die Variablen repräsentieren die schon oben erwähnten Mengen *allInstances_packname_X*. Es folgen die Invarianten. Mit ihnen werden die Bedingungen wie Vollständigkeit (*complete*) und Disjunktheit (*disjoint*) beschrieben. Anschließend werden die Operationen zum Hinzufügen und Entfernen von Elementen definiert. Zum Schluss kommen die Bedingungen für den Anfangszustand und die Postconditions. Hier werden auch alle vorhandenen *next-state*-Relationen zu einer zusammengefasst, um eine kürzere Schreibweise für die folgende Formel der Gesamtspezifikation zu ermöglichen. Ganz am Ende folgt das Theorem.

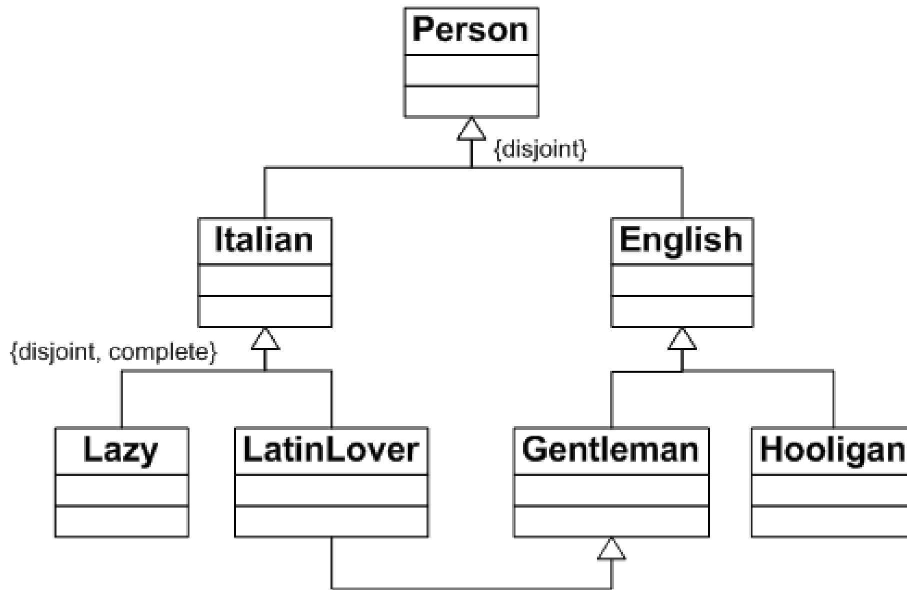


Abbildung 4: UML-Diagramm Persons

Die Bedingungen der Vererbung habe ich als Invarianten angegeben. Dass zum Beispiel eine Klasse A die Basisklasse einer anderen Klasse B ist, kann in TLA^{+} wie folgt angegeben werden

$$InvariantInheritance \triangleq allInstances_B \subseteq allInstances_A$$

Da alle Elemente der vererbten Klasse B auch zu der Menge *allInstances_A* hinzugefügt werden, muss also die reine Menge der Elemente von B, die Menge *allInstances_B*, eine Untermenge der Menge *allInstances_A* sein. Ist die Menge A leer, so sind die beiden Mengen gleich.

Gibt es mehrere vererbte Klassen, so muss für jede Klasse eine solche Invariante angegeben werden. Kommen weitere Bedingungen wie `{disjoint}` hinzu, kann dies wie folgt umgesetzt werden

$$InvariantDisjointness \triangleq allInstances_B \cup allInstances_C = \{\}$$

Die Bedingung `{disjoint}` besagt, dass Elemente nur entweder Element der einen Klasse oder Element der anderen Klasse sein dürfen, jedoch nicht von beiden gleichzeitig. Die Schnittmenge muss also der leere Menge entsprechen.

Die Bedingung `{complete}` sagt aus, dass als Unterklassen nur die aufgelisteten Klassen in Frage kommen. Ein Element kann also nur Element aus einer

der angegebenen Unterklassen sein. Dies impliziert eine abstrakte Basisklasse, also eine Klasse, von der selber es keine Instanzen geben kann. In TLA^+ kann dies durch folgende Invariante festgelegt werden

$$InvariantCompleteness \triangleq \{\}$$

Die *next-state*-Relationen ergaben sich aus der oben beschriebenen Tatsache, dass ich die schon verwendeten Elemente einer gesonderten Menge hinzufüge. Wurden noch nicht alle Elemente einer Klasse verwendet, so wird ein noch nicht verwendetes Element ausgewählt und der Menge *allInstances_Klasse* hinzugefügt. Ist diese Klasse Unterklasse einer oder mehrerer anderer Klassen, so wird das Element auch den entsprechenden anderen Mengen *allInstances* hinzugefügt. Als Precondition gilt dabei, dass das Element noch nicht in der Menge enthalten sein darf. Als Postcondition muss gelten, dass andere Mengen, von denen die Klasse des Elementes keine Untermenge ist, nicht verändert werden dürfen. Analog dazu wird ein Element beim Entfernen aus der Menge *allInstances* auch aus allen anderen Mengen *allInstances* entfernt, denen es hinzugefügt wurde. Also Precondition muss hier gelten, dass das Element in der Menge enthalten ist und als Postcondition muss wieder gelten, dass die Mengen anderer Klassen nicht verändert werden.

Die Bedingung für die Anfangszustände ist bei allen Dateien gleich. Alle Mengen müssen zu Beginn der leeren Menge entsprechen. Auch die Zusammenfassung der einzelnen *next-state*-Relationen zu einer gesamten ist in allen Dateien gleich. Sie besteht aus den Konjunktionen aller benutzten einzelnen *next-state*-Relationen.
Als Postcondition habe ich jeweils die Bedingung angegeben, dass alle Elemente der in der Konfigurationsdatei gegebenen Mengen verarbeitet werden müssen.
Die Formeln für die Spezifikation und das Theorem sind auch bei allen Dateien gleich.

Die TLA^+-Spezifikation zweier Klassen, die durch eine Assoziation verbunden sind, wobei das eine Assoziationsende eine Multiplizität von *amin* bis *amax* und die Bedingungen *ordered* und *nonUnique* besitzt und das andere Ende eine Multiplizität von 0 bis unendlich, ist im Anhang zu finden. Sie wurde von Miguel Garcia geschrieben. Zusammen mit dieser Datei sollten nun die meist verwendeten Elemente eines UML-Klassendiagrammes in die Sprache TLA^+ umgewandelt werden können.

2.2.2 Umformung der OCL nach TLA$^+$

Es gibt drei standard OCL-Constraints. Diese sind Invariante, Pre- und Postcondition. Sie können ohne größeren Aufwand in TLA$^+$-Code umgewandelt und so in die Spezifikation integriert werden.

So können zum Beispiel OCL-Invarianten in der Sprache TLA$^+$ als Invarianten der Spezifikation angegeben werden. Zu einer Klasse können mehrere Invarianten gehören. Die verschiedenen Invarianten aller Klassen werden mit Konjunktionen zusammengefasst und in dem Theorem am Ende des Moduls angegeben. In dem aus [Ca03] entnommenen Beispiel lauten die Invarianten zu dem UML-Diagramm in Abbildung 5:

```
context Flight
inv  : self.type = 'cargo' implies plane.type = Type::cargo
inv  : self.type = 'passenger' implies
                        plane.type = Type::passenger
```

In TLA$^+$-Code können die Invarianten wie folgt geschrieben werden:

$$flight \in \{CargoFlight, PassengerFlight\}$$
$$airplane \in \{CargoPlane, PassengerPlane\}$$

$$InvariantCargoFlight \triangleq CargoFlight \Rightarrow CargoPlane$$
$$InvariantPassengerFlight \triangleq PassengerFlight \Rightarrow PassengerPlane$$

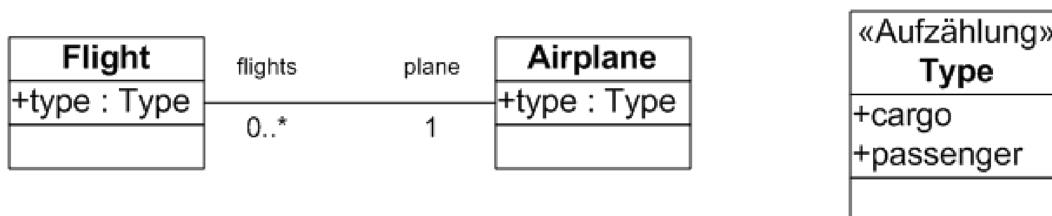


Abbildung 5: UML-Diagramm Flights

Ein anderes, einfaches Beispiel für eine Invariante ist die Bedingung, dass ein Autofahrer mindestens 18 Jahre alt sein muss, um fahren zu dürfen.
Sie lautet als OCL-Constraint

```
context Age
inv : self.age >= 18
```

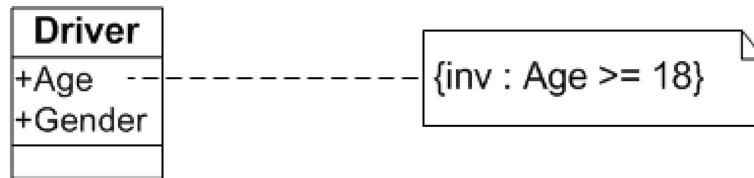


Abbildung 6: UML-Diagramm Driver

In einer TLA$^+$-Spezifikation kann diese Bedingung als

$$InvariantAge \triangleq Age \geq 18$$

formuliert werden.

Mit Hilfe von Invarianten wird auch häufig der Wert einer Variablen im Anfangszustand definiert. Auch hier wird die OCL-Invariante in TLA$^+$ als Invariante angegeben. Die zu Abbildung 7 gehörende OCL-Bedingung lautet

```
context Person::Age : Integer
init : 0
```

In TLA$^+$ lautet sie

$$InvariantPersonAge \triangleq Age = 0$$

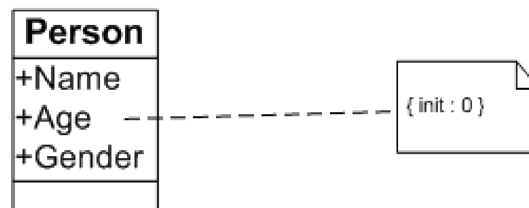


Abbildung 7: UML-Diagramm Person

Pre- und Postcondition können in TLA$^+$ ohne größeren Aufwand direkt in die Formel der *next-state*-Relation eingebracht werden. Dazu werden für die Precondition die Variablen ohne, für die Postcondition die Variablen mit Strichindex verwendet. Die einzelnen Bedingungen und die *next-state*-relation selber werden durch Konjunktionen miteinander verknüpft.

Soll zum Beispiel eine Funktion beschrieben werden, die die Modulodivision von einem festen Wert *val* und einem per Parameter übergebbarem Divisor *div*

berechnen soll, so muss darauf geachtet werden, dass der Divisor *div* von 0 unterschiedlich ist. Als Postcondition muss gelten, dass der errechnete Wert kleiner als der Wert *div* ist.

$$
\begin{aligned}
&modulo(div) \triangleq \\
&\qquad\quad \backslash * Precondition \\
&\qquad \land\ div \neq 0 \\
&\qquad \land\ val' = val\% div \\
&\qquad\quad \backslash * Postcondition \\
&\qquad \land\ val' < div
\end{aligned}
$$

In OCL gibt es einige vordefinierte Funktionen. Auch die meisten dieser Funktionen lassen sich ohne große Mühe nach TLA^{+} umwandeln. In Tabelle 5 sind die wichtigsten dieser Funktionen aufgelistet.

OCL	TLA^{+}	OCL	TLA^{+}
not	$\neg$	x.intersection(y)	$x \cap y$
and	$\land$	x.union(y)	$x \cup y$
or	$\lor$	x.includes(y)	$y \in x$
implies	$\Rightarrow$	x.excludes(y)	$y \notin x$
x.including(y)	$x \cup \{y\}$	x.includesAll(y)	$y \subseteq x$
x.excluding(y)	$x \setminus \{y\}$	x.isEmpty()	$x = \emptyset$
x.excludesAll(y)	$x \cap y = \emptyset$	x.notEmpty()	$x \neq \emptyset$
x.equals(y)	$x = y$	x <> y	$x \neq y$

Tabelle 5: vordefinierte Operatoren der OCL Quelle [Sch06OCL]

Invarianten, Pre- und Postconditions habe ich in den TLA^{+}-Spezifikationen häufig benutzt. So ist zum Beispiel die Bedingung, dass bei einer Generalisierung mit dem Zusatz `{disjoint, complete}` die beiden Subklassen nur die leere Mange als Schnittmenge haben, eine Invariante. Die Bedingung, dass ein zu löschendes Element in einer Menge enthalten sein muss, um es aus dieser Menge löschen zu können, ist eine Precondition. Eine andere Precondition ist die *Init*-Bedingung, die die Werte in dem Anfangszustand definiert. Postconditions sind zum Beispiel die Bedingungen, dass, nachdem eine Funktion durchgeführt wurde, bestimmte andere Mengen unverändert geblieben sein müssen.

2.3 Überprüfung mit dem Model-Checker TLC

2.3.1 Allgemeines über TLC

TLC ist ein Model-Checker, der TLA$^+$-Beschreibungen von Systemen auf Korrektheit überprüfen kann. Dieser Model-Checker wurde von Yuan Yu entworfen und entwickelt, mit Hilfe von Leslie Lamport, Mark Tuttle und Mark Hayden. Die aktuelle Version ist Version 2.0, mit der auch ich gearbeitet habe.
Dieses in Java geschriebene Programm kann von der Seite

http://research.microsoft.com/users/lamport/tla/tools.html

kostenlos heruntergeladen werden. Eine kurze Anleitung zur Installation und Benutzung des Programmes sind im Anhang A zu finden.

2.3.2 Einführung in TLC

TLC erwartet als Eingabe eine Spezifikation in der Form

$$Init \land \Box[Next]_{vars} \land Temporal$$

oder, falls keine temporale Formel benutzt wurde, in der Form

$$Init \land \Box[Next]_{vars}$$

In der Spezifikation sollte jedoch nicht der Versteck-Operator (*hiding operator*) $\exists$ benutzt werden, da TLC Spezifikationen mit diesem Operator nicht richtig überprüfen kann.

TLC überprüft Formeln von links nach rechts. So sollten Bedingungen an erster Stelle stehen und Berechnungen folgen. Ein Beispiel ist, dass bei einer Division der Divisor ungleich von 0 sein muss. Den Ausdruck

$$(x \neq 0) \land (a = 500/x)$$

kann TLC berechnen. Das logische Äquivalent dazu

$$(a = 500/x) \land (x \neq 0)$$

kann hingegen nicht berechnet werden, da zuerst die Division berechnet wird und dort schon der Fehler auftritt. Da der Mensch intuitiv als erstes die Bedingung aufstellt, weil es so einfaches zu verstehen ist, tritt dieser Fehler meist nicht auf.

Intern rechnet TLC mit einem gerichteten Graphen. Ein Knoten dieses Graphen entspricht einem Zustand des Systems. Genauer gesagt entspricht ein Knoten einem Tupel von Werten aller definierten Variablen. Da die Werte dieser Variablen jedoch den Zustand des Systems bestimmen, kann von Zuständen gesprochen werden. In dem Graphen werden alle schon berechneten Zustände des Systems gespeichert. In einer Warteschlange (*queue*) werden zusätzlich alle Knoten des Graphen gespeichert, deren Nachfolgezustände noch nicht berechnet wurden.

Nach einem bestimmten Algorithmus errechnet TLC zu Beginn die Anfangszustände und fügt nach und nach die schon berechneten Zustände mit den entsprechenden Kanten in den Graphen ein, wobei die Zustände, von denen noch nicht alle Nachfolgezustände berechnet wurden, auch in die Warteschlange eingefügt werden. Solange die Warteschlange nicht leer ist, entnimmt TLC der Warteschlange den ersten Zustand und berechnet für ihn alle Nachfolgezustände. Sind sie noch nicht im Graphen enthalten, werden sie mit einer entsprechenden Kante dem Graphen hinzugefügt.
Die Zustände werden als eine 64-bit Zahl gespeichert. Diese Zahl wird durch eine Hashfunktion berechnet. Die Wahrscheinlichkeit dafür, dass zwei unterschiedliche Zustände dieselbe 64-bit Zahl als Hashwert haben, beträgt 2^{-64}. Sie ist sehr gering, es ist aber durchaus möglich, dass ein solcher Fall eintritt. Ist dies jedoch geschehen, so sieht TLC die beiden unterschiedlichen Zustände als einen an. Das heißt, es werden nicht alle möglichen erreichbaren Zustände berechnet, obwohl in der Ausgabe von TLC steht, dass dies geschehen sei. Um die Wahrscheinlichkeit für eine Kollision zweier unterschiedlichen Zustände mit gleichem Hashwert etwas besser bewerten zu können, werden zwei Wahrscheinlichkeiten mit angegeben. Die erste basiert auf der Annahme, dass die Wahrscheinlichkeit für einen gleichen Hashwert 2^{-64} beträgt. Sie wird mit der Formel $m * (n - m) * 2^{-64}$ berechnet, wobei m für die Anzahl der Zustände und n für die Anzahl der unterschiedlichen Hashwerte steht. Die zweite Wahrscheinlichkeit wird aufgrund von Beobachtungen berechnet. In der Praxis hat sich herausgestellt, dass die Wahrscheinlichkeit für zwei gleiche Hashwerte bei unterschiedlichen Zuständen sehr gering ist.
Der verwendete Algorithms in Pseudocode, eine genauere Erklärung der internen Darstellung und eine Erläuterung zu der Berechnung der Wahrscheinlichkeiten, sind in [Lam02] nachzulesen.

2.4 Resultate

Wurden in einer Spezifikation keine Fehler gefunden, so beginnt die Ausgabe des Model-Checkers TLC mit der verwendeten Version. Es folgt der Modus, in dem TLC gestartet wurde. Die nächsten Zeilen geben die Module und ihre Dateien an, die verwendet wurden. Als nächstes werden Angaben über den Anfangszustand ausgegeben, auf die eventuelle Zwischenausgaben folgen. Es folgt die Aussage, dass das Model-Checking beendet und keine Fehler gefunden wurden. Abschließend werden, falls zwei Zustände mit gleichem Hashwert gefunden wurden, die Wahrscheinlichkeiten ausgegeben sowie die Anzahl der berechneten Zustände, der gefundenen unterschiedlichen Zustände und die Tiefe des Graphen. Ein Beispiel für eine Ausgabe von TLC, in der keine Fehler gefunden worden sind, ist in Abbildung 8 dargestellt.

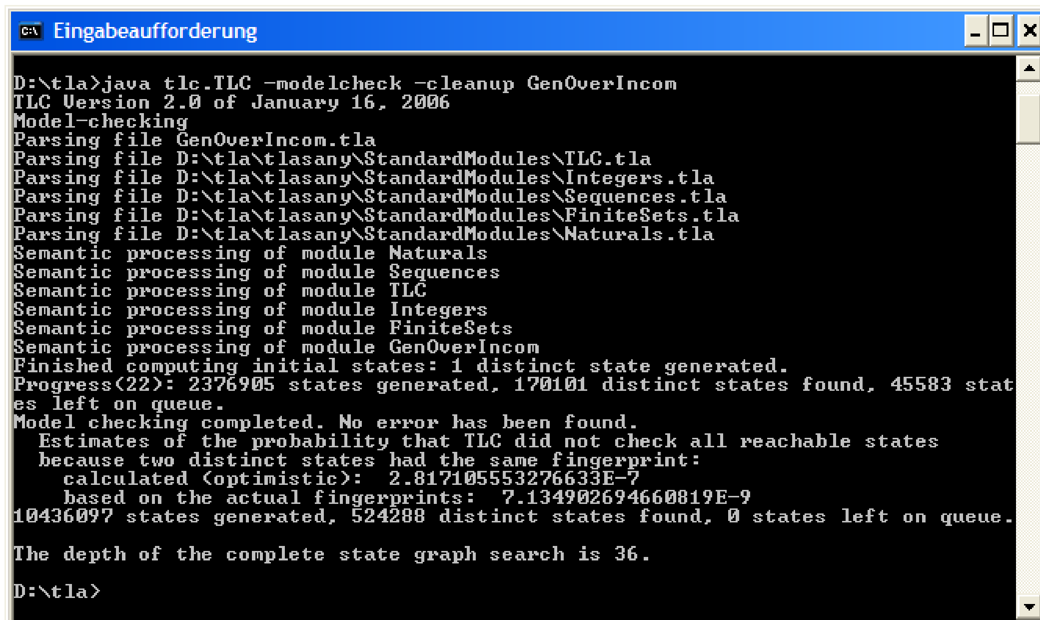


Abbildung 8: Die Ausgabe der Überprüfung der Datei GenOverIncom.tla

Wurden Fehler in einer Spezifikation gefunden, so werden sie an entsprechender Stelle ausgegeben. Die Fehler, die als erstes gefunden werden, sind die Syntaxfehler. Als nächstes folgen eventuelle Fehlermeldungen für einen ungültigen Anfangszustand. Andere Fehler werden mit den letzten paar Zuständen und einer Angabe, wo der Fahler aufgetreten ist, ausgegeben. Eine solche Ausgabe, in der ein Fehler gefunden wurde, ist in Abbildung 9 zu sehen.

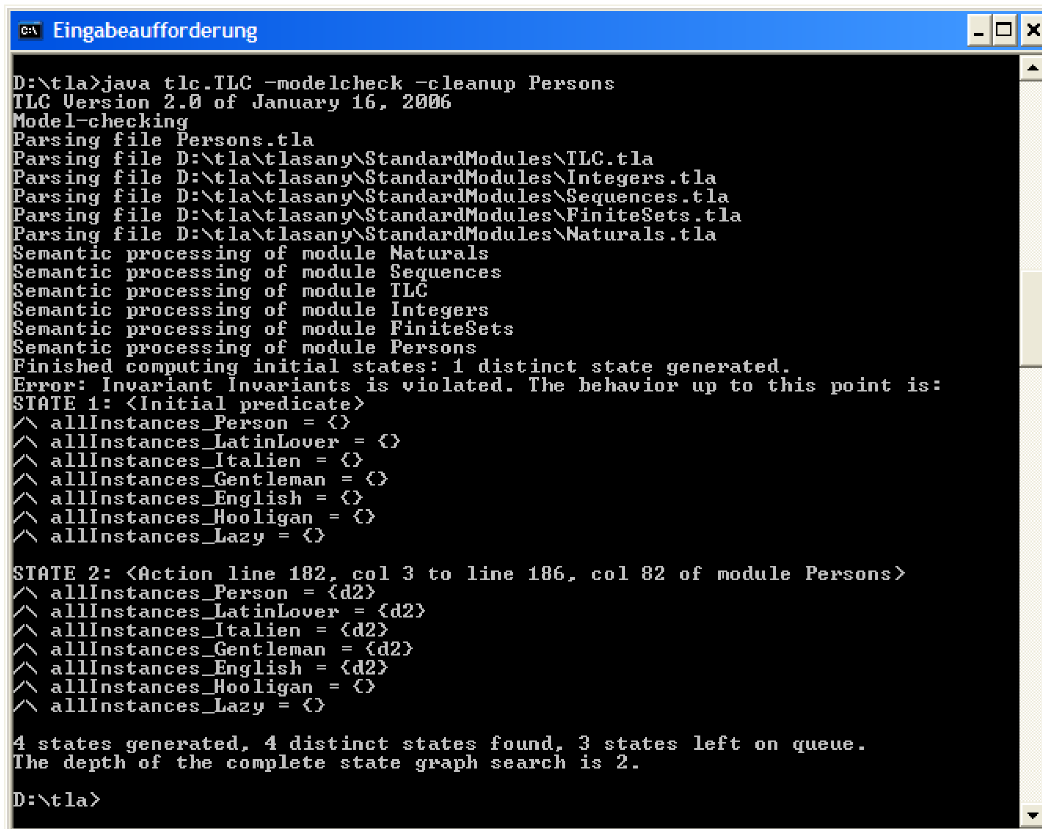


Abbildung 9: Die Ausgabe der Überprüfung der Datei Persons.tla

Ich habe alle von mir erstellten Dateien mit TLC überprüft. Dabei habe ich zuerst die Optionen *-modelcheck* und *-cleanup* verwendet. Die von mir benutzten Konfigurationsdateien sind im Anhang C in den *tla.*-Dateien als Kommentar mit angegeben. Die dazugehörigen Ausgaben sind auf der CD im Ordner `TLA/Ausgabedateien` als *.png*-Dateien zu finden. Die Ausgabe der Überprüfung der Datei *Persons.tla* mit abgeänderter Konfigurationsdatei ist als *AusgabePersons2.png* gespeichert. Die genaue Abänderung beschreibe ich später in diesem Kapitel. Bis auf die Datei *Persons.tla* konnten alle Dateien durch den Model-Checker TLC verifiziert werden. Die Zeiten, die dazu benötigt wurden, sind in Tabelle 6 eingetragen. Sie beruhen auf meinen eigenen Messungen mit einer Stopp-

uhr. Als PC habe ich ein Notebook mit einem *Mobile AMD Sempron 3000+* Prozessor und *512 MB DDR-RAM* Arbeitsspeicher verwendet. Als Betriebssystem benutze ich *Windows XP - Home Edition Version 2002* mit dem *Service Pack 2*. Zusätzlich habe ich die Anzahl der berechneten Zustände, die Anzahl der unterschiedlichen Zustände und die Tiefe des erstellten Graphen mit angegeben.

Datei	benötigte Zeit	Zustände gesamt	Zustände verschieden	Tiefe des Graphen
Generalisierung.tla	4,5s	228.337	16.384	27
GenOverIncom.tla	116s	10.436.097	524.288	36
GenDisCom.tla	6s	310.241	32.768	29
Italians.tla	104,5s	6.848.001	524.288	35
Persons.tla	2,5s	4	4	2
*Persons.tla**	690s	31.752.193	2.097.152	38

Tabelle 6: gemessene Zeiten TLC

Bei der Datei *Persons.tla* wurde nach 2,5 Sekunden die in Abbildung 9 gezeigte Fehlermldung angezeigt. Dies sollte auch so der Fall sein, denn es darf kein Element existieren, das gleichzeitig in den Mengen *allInstances_Italian* und *allInstances_English* vorhanden ist. Die Klasse *LatinLover* wurde von der Klasse *Italian* abgeleitet. So ist jedes Element von *LatinLover* in der Menge *allInstances_Italian* enthalten. Dadurch, dass die Klasse *LatinLover* aber auch eine abgeleitete Klasse der Klasse *Gentleman* ist, die wiederum eine von der Klasse *English* abgeleitet ist, ist jedes Element von *LatinLover* auch Element der Menge *allInstances_English*. Laut Invariante ist dies ungültig.
Wird die Konfigurationsdatei so abgeändert, dass die Menge der *LatinLover* eine leere Menge ist, so verifiziert TLC die Datei *Persons.tla* in circa elf Minuten und 30 Sekunden (in Tabelle 6 mit einem * gekennzeichnet). Es wird jedoch nicht erkannt, dass keine *LatinLover* existieren dürfen. Zusätzlich habe ich diese Datei im Simulationsmodus überprüfen lassen. Dazu habe ich die Optionen *-simulate*, *-depth 2* und *-cleanup* benutzt. Entspricht die Konfigurationsdatei der im Kommentar angegebenen Datei, so wird nach weniger als zwei Sekunden ein Zustand gefunden, in dem die Invariante verletzt wurde. Wird sie jedoch wieder wie oben beschrieben abgeändert, so dauert die Überprüfung wesentlich länger. Nach über acht Stunden wurden 812.201.482 Zustände bearbeitet, ohne einen Widerspruch gefunden oder die Korrektheit bewiesen zu haben. An dieser Stelle sehe ich die Notwendigkeit einen völlig von der Konfigurationsdatei unabhängigen Simulationsmodus zu schaffen, der einsolches Problem erkennen kann. Nur so können auch Fehler in dem Modell selber gefunden werden, die vielleicht sonst nicht direkt erkennbar sind.

Eine andere nötige Erweiterung machte sich beim Überprüfen der Datei *Italians.tla* bemerkbar. Die Spezifikation wurde zwar verifiziert, was auch so sein soll, aber die Tatsache, dass die Mengen *LatinLover* und *ItalianProf* identisch sind, wurde nicht erkannt.

Als einen weiteren Test habe ich die Datei *Generalisierung.tla* im Simulationsmodus mit den Optionen *-simulate* und *-cleanup* laufen lassen. Nach 20 Minuten wurden bereits 107.295.555 Zustände bearbeitet. Nach 40 Minuten waren es 214.540.321 und nach 60 Minuten 321.771.162. Nach etwas über einer Stunde habe ich die Simulation abgebrochen. Da die Datei eine Spezifikation von nur zwei Klassen ohne Attribute darstellt, denke ich, dass bei einer solchen langen benötigten Zeit dieser Modus nicht für eine praktische Anwendung geeignet ist.

Insgesamt denke ich, dass sich TLA^+ in Kombination mit dem Model-Checker TLC recht gut für eine automatisierte Verifikation eignet. Die Tatsache, dass TLC in Java programmiert wurde und frei zugänglich ist, erleichtert eine Einbindung und noch mögliche Erweiterungen. Auch lassen sich die UML-Diagramme relativ einfach in die Sprache TLA^+ umwandeln. OCL-Bedingungen können ohne größeren Aufwand mit integriert werden und die in TLA^+verwendbaren Programmierkonstrukte erleichtern die Umwandlung von Operationen.
Die oben genannten Erweiterungen sollten möglichst umgesetzt werden, damit eine noch größere Sicherheit für die Korrektheit gewährleistet werden kann.
Einen Nachteil wird wahrscheinlich die Laufzeit mit sich bringen. Die UML-Diagramme, die ich bisher getestet habe, bestanden aus maximal sieben Klassen und es waren keine Attribute vorhanden. In den Konfigurationsdateien existierten maximal zehn Elemente pro Menge. Auch gab es keine komplizierten Operationen, die umgeformt werden mussten. Trotzdem wurden, wie aus Tabelle 6 ersichtlich ist, schon teilweise einige Millionen Zustände berechnet. Sollen nun komplexere Diagramme mit Attributen, Funktionen und vielen Constraints überprüft werden, so wird dies wahrscheinlich zu einer im Verhältnis wesentlich längeren Laufzeit führen. Einen weiteren Nachteil bringt die Tatsache mit sich, dass immer eine Konfigurationsdatei mit einer möglichen Belegung erstellt werden muss. Gibt es keine Konfigurationsdatei, kann die Spezifikation weder im Modus ModelChecking noch im Simulationsmodus überprüft werden.

2.5 Eine mögliche Implementierung

Ein schon existierendes MDA-Tool ist Octopus (OCL Tool for Precise UML Specifications). Es wurde von Jos Warmer und Anneke Kleppe erstellt. Mit Hilfe dieses Programmes können OCL-Constraints auf syntaktische Fehler und auf den korrekten Gebrauch überprüft werden. Eine andere wichtige Eigenschaft ist, dass dieses Programm UML-Diagramm zusammen mit OCL-Constraints in Java-Code umwandeln kann.
Da das Programm selber in Java programmiert wurde und der Code frei zugänglich ist, kann Octopus so erweitert werden, dass dieses Programm auch TLA$^+$-Code erstellen und ihn mit TLC überprüfen lassen kann.
In diesem Kapitel beschreibe ich eine denkbare Erweiterung des Programms Octopus, die dies ermöglichen würde.

2.5.1 Voraussetzungen

Octopus ist ein Eclipse Plug-In, daher wird die Eclipse-Plattform benötigt. Sie kann unter

http://www.eclipse.org/downloads/

kostenlos heruntergeladen werden. Die aktuelle Version von Octopus, Version 2.2.0, benötigt Eclipse 3.1 oder höher, jedoch gibt es einige Probleme bei der Version 3.1.1. Eclipse wiederum benötigt ein Java Runtime Environment (JRE). Dieses kann unter anderem von der Seite

http://developers.sun.com/resources/downloads.html

heruntergeladen werden. Dabei ist darauf zu achten, dass Java 1.5 gewählt wird. Mit älteren Versionen von Java funktioniert Octopus nicht.
Wurden das JRE und Eclipse installiert, kann Octopus installiert werden. Die Installationsdatei ist bei SourceForge unter

http://sourceforge.net/projects/octopus/

erhältlich. Eine genaue Anleitung für die Installation ist im Internet unter der Adresse

http://www.klasse.nl/octopus/octopus-install.html

einzusehen.

2.5.2 Funktionsweise

Wurden das UML-Diagramm über die *.xmi*-Datei und die OCL-Constrains über *.ocl*-Dateien in Octopus importiert, werden die einzelnen Elemente und Bedingungen in ein internes Datenformat umgewandelt. Die Interfaces dazu sind im Package *octopus.model* enthalten. Für jedes Element eines UML-Klassendiagramms ist eine entsprechende Klasse mit einem Interface vorhanden. So heißt zum Beispiel das Interface für eine UML-Klasse *IClass*. In den Interfaces werden wichtige Funktionen deklariert, wie zum Beispiel *set*- und *get*-Funktionen. Diese Funktionen werden dann in den entsprechenden Klassen im Package *octopus.model.internal.types* implementiert. So heißt zum Beispiel die Klasse, die das Interface *IClass* implementiert, *ClassImpl*. In diesen Klassen wird auch die *accept*-Funktion für einen Visitor vom Typ *IPackageVisitor* implementiert. Ein solcher Visitor vom Typ *IPackageVisitor* wurde im Package *octopus.modelVisitors* realisiert. Dieser Visitor, *PackageToString*, schreibt, wird er aufgerufen, die Namen aller Packages, Klassen, Attribute, Assoziationen etc. als String in einen Buffer. Dieser Buffer wird von einer anderen Funktion auf der Konsole ausgegeben. Aufgerufen wird der PackageToString-Visitor zum Beispiel, wenn auf den Button *Show UML Model* geklickt wurde, also von einem Plug-In.

2.5.3 mögliche Erweiterungen

Damit ein eingelesenes UML-Modell in TLA$^+$-Code umgewandelt werden kann, müsste ein neues Plug-In hinzugefügt werden. Dieses Plug-In müsste den Code eines neuen Visitors enthalten wie auch die Funktionen, um ihn aufzurufen. Als Grundlage für einen neuen Visitor, zum Beispiel *PackageToTLA*, kann der Visitor *PackageToString* dienen. Anstatt die Namen der Elemente zusammen mit der Elementart in einen Buffer zu schreiben, was der Visitor PackageToString macht, könnten die Namen zusammen mit TLA$^+$-Code erst in einen Buffer und schließlich in eine Datei geschrieben werden. So könnte die Funktion *packageBefore* nicht mehr wie im Visitor PackageToString aussehen, was in Abbildung 10 dargestellt ist, sondern wie in Abbildung 11. Um den Unterschied zu verdeutlichen, ist der ursprüngliche Code auskommentiert.

```
public void package_Before(IPackage p) {
    buffer.append("<package> " + p.getPathName() + "\n");
}
```

Abbildung 10: Funktion packageBefore im Visitor PackageToString

Um die erstellte Datei mit dem Model-Checker TLC ohne eine Eingabe über die Konsole überprüfen zu können, könnte der Code von TLC mit in das Plug-In integriert werden. TLC wurde in Java programmiert und so ist es einfach, seinen

```
public void package_Before(IPackage p) {
    //buffer.append("<package> " + p.getPathName() + "\n");
    buffer.append("------------- MODULE " + p.getPathName() + " -------------" + "\n");
}
```

Abbildung 11: Funktion packageBefore im Visitor PackageToTLA

Code zu integrieren, da auch er frei zugänglich ist. Würde in die *run*-Funktion eines neuen Buttons der Aufruf der *main*-Funktion vom Model-Checker implementiert werden, könnte so eine ausgewählte Datei, also zum Beispiel die neu erstellte TLA-Datei, mit TLC überprüft werden. Analog dazu wäre es möglich, die erstellt TLA-Datei mit TLATEX in eine *.pdf*-Datei zu konvertieren.
Die beiden Schritte, also das Erstellen der TLA-Datei und das Überprüfen, könnten auch in einem Schritt kombiniert werden. Dazu müssten bei Klick auf einen Button nur beide Schritte ohne weitere Aktion des Benutzers hintereinander ausgeführt werden.

3 Deduktion

Das Wort *Deduktion* stammt von dem lateinischen Wort *deducere*, was übersetzt *herabführen* bedeutet. Damit ist die Folgerung von etwas Allgemeinem auf etwas Spezielles gemeint. Aus einer Menge von Formeln kann also auf logischem Wege eine spezielle Formel hergeleitet werden. Für diese Herleitung wird die *Resolution* benutzt. Es handelt sich hierbei um eine syntaktische Umformungsregel, bei der aus zwei Formeln eine entsteht, unter der Bedingung, dass beide Formeln die Voraussetzungen erfüllen. Mit der Resolution ist es möglich zu zeigen, dass eine Menge von Formeln unerfüllbar ist. Es ist jedoch nicht direkt möglich zu beweisen, dass eine Formelmenge erfüllbar ist. Hierzu muss erst die zu beweisende Formel negiert werden und mit der negierten Formel die Unerfüllbarkeit gezeigt werden.
Eine Voraussetzung für dieses Verfahren ist, dass die Formelmenge in der Klauselschreibweise vorliegt. Die andere Voraussetzung ist, dass ein *Resolvent* existiert. Sind K_1, K_2 und R Klauseln, dann heißt R Resolvent von K_1 und K_2, falls R die Form

$$R = (K_1 - \{L\}) \cup (K_2 - \{\overline{L}\})$$

hat und L ein Literal ist, wobei $L \in K_1$ und $\overline{L} \in K_2$ gelten. $\overline{L}$ ist dabei definiert als

$$\overline{L} = \begin{cases} \neg A_i & \textit{falls} \quad L = A_i \\ A_i & \textit{falls} \quad L = \neg A_i \end{cases}$$

.

Tritt die leere Menge, sie wird mit dem Symbol $\square$ gekennzeichnet, als Resolvent auf, ist die Klauselmenge laut Definition unerfüllbar.

Das Verfahren der Deduktion leitet also mit Hilfe der Regeln der Resolution die leere Klausel her, was somit die Unerfüllbarkeit der Klauselmenge zeigt.

Ein Programm, welches die Regeln der Resolution anwenden kann und somit eine mögliche leere Klausel aus einer Menge von Klauseln bzw. sogar Formeln herleiten kann, ist der Beweiser OTTER.

3.1 Otter

3.1.1 Allgemeines über Otter

Das Programm OTTER (*Organized Techniques for Theorem-proving and Effective Research*) wurde zum ersten Mal in der Version 0.9 auf der 9. internationalen Konferenz der automatisierten Deduktion (*9th Conference on Automated Deduction - CADE-9*) im Mai 1988 vorgestellt. Im August 2004 wurde die Version 3.3

veröffentlicht, die bisher letzte. Heute wird OTTER nicht mehr weiterentwickelt und auch nur noch minimal gewartet und unterstützt. Es existiert jedoch ein Nachfolger namens Prover9.
OTTER wurde von der Abteilung für Mathematik und Informatik (*Mathematics and Computer Science Division - MCS Division*) vom Argonne National Laboratory entwickelt. Diese Abteilung wurde hauptsächlich von der wissenschaftlichen Abteilung für Mathematik, Information und Informatik (*Mathematical, Information, and Computational Sciences Division*), der Behörde für fortgeschrittene Computerforschung (*Office of Advanced Scientific Computing Research*) und der Behörde für Wissenschaft (*Office of Science*) von der U.S. Abteilung für Energie (*U.S. Department of Energy*) gegründet. Informationen über dieses und andere Projekte der *MCS Division* können unter

http://www-new.mcs.anl.gov/new/

gefunden werden.

OTTER ist ein Theoremprüfer für Prädikatenlogik mit Gleichheit im Stil der Resolution (resolution-style theorem-prover for first-order logic). Mit diesem Programm ist es möglich, Formeln nach den Regeln der Resolution, Hyperresolution, UR-Resolution und binärer Paramodulation umzuformen. Weitere Eigenschaften sind unter anderem das Umformen von Formeln in Klauseln, vorwärts und rückwärts Subsumierung und das Ordnen von Formeln. Für diese Arbeit habe ich lediglich die Resolution benutzt. Informationen über Klauseln und Resolution können in [Schö05] nachgelesen werden.
Das Programm selber wurde in ANSI C geschrieben, um eine gute Performance, Portabilität und die Möglichkeit zur Erweiterung zu bieten.

3.1.2 Einführung in die Syntax von Otter

Als Eingabe sind prädikatenlogische Formeln oder Klauseln möglich. Das Programm selber arbeitet nur mit Klauseln. Wurden Formeln benutzt, formt OTTER sie automatisch in Klauseln um. Bei diesem Vorgang werden als erstes die Implikationen ($\Rightarrow$) und Äquivalenzen ($\Leftrightarrow$) eliminiert. Dann werden die Negationen ($\neg$) vor die Atome gezogen. Schließlich werden die Quantoren ($\forall$, $\exists$) nach außen gebracht und falls nötig gebundene Variablen umbenannt. Nun befindet sich die Formel in der Pränexnormalform. Von dort aus wird die Formel weiter in die Skolemnormalform gebracht, was bedeutet, dass die Existenzquantoren ($\exists$) nach vorne gebracht und gestrichen werden. Die dann freien Variablen werden durch einen Skolemterm ersetzt. Diese Umformung ist nicht äquivalent. Eine Formel ist aber genau dann erfüllbar, wenn ihre Skolemnormalform erfüllbar ist. Daher kann diese Umformung vorgenommen werden. Dann wird die Matrix der entstandenen Formel in die konjunktive Normalform (KNF) gebracht und die Allquantoren ($\forall$)

werden gestrichen. Schließlich wird die entstandene Formel in die Mengenschreibweise überführt. Mit der so hergeleiteten Klauselmenge kann resolviert werden. Die Eingabe der Formeln kann in Prefix-, Listen- oder (abgekürzter) Infixform erfolgen. Für die Listenform wird dabei die Prolog-Notation verwendet.

Die Syntax von OTTER ist der Syntax der Prädikatenlogik sehr ähnlich. In den Formeln dürfen jedoch keine freien Variablen vorkommen - alle Variablen müssen durch Quantoren gebunden sein. Da die Eingabe als normale Textdatei erfolgt, müssen die Symbole der Prädikatenlogik etwas anders geschrieben werden. Wie genau die Symbole umschrieben werden müssen, ist in Tabelle 7 dargestellt.

<table>
<tr><th>Operator</th><th>Bezeichnung</th><th>OTTER-Syntax</th></tr>
<tr><td>$\neg$</td><td>Negation</td><td>-</td></tr>
<tr><td>$\vee$</td><td>Disjunktion</td><td>|</td></tr>
<tr><td>$\wedge$</td><td>Konjunktion</td><td>&</td></tr>
<tr><td>$\Rightarrow$</td><td>Implikation</td><td>-></td></tr>
<tr><td>$\equiv$</td><td>Äquivalenz</td><td><-></td></tr>
<tr><td>$\exists$</td><td>Existenzquantor</td><td>exists</td></tr>
<tr><td>$\forall$</td><td>Allquantor</td><td>all</td></tr>
</table>

Tabelle 7: Operatoren der Prädikatenlogik in der OTTER-Syntax

Die Formel

$$\forall x P(x)$$

wird in der OTTER-Syntax als

```
all x (P(x)).
```

dargestellt. Werden mehrere Variablen durch jeweils einen Quantoren gleicher Art gebunden, so können in der OTTER-Syntax bis auf den ersten Quantor die Quantoren weggelassen werden. Die Formel

```
all x all y all z (P(x,y,z)).
```

kann so kürzer als

```
all x y z (P(x,y,z)).
```

geschrieben werden.

Operatoren wie der Zähl-Existenzquantor (*counting existential quantifier*) existieren in OTTER nicht.

Kommentare in OTTER-Eingabedateien können mit einem % eingeleitet werden. Sie werden mit dem Ende der Zeile auch beendet.

Bei Variablen wird zwischen drei Arten unterschieden. Die ersten sind Variablen mit gewöhnlichen Namen (*ordinary name*). Der Name darf dabei aus einer Menge von alphanumerischen Zeichen, `$` und `_` bestehen. Das Zeichen `$` darf dabei nicht an erster Stelle stehen, denn solche Namen sind für einen speziellen Zweck reserviert. Die zweite Art der Variablen sind die mit einem speziellen Namen (*special name*). Dieser Name darf aus den Zeichen `* + - / \ ^ < > ' ~ : ? ! ; #` und manchmal auch `|` bestehen. Die dritte und letzte Art von Variablen sind die Variablen mit einem zitiertem Namen (*quoted name*). Der Name selber wird bei diesen Variablen in jeweils zwei “ oder ' eingeschlossen. Das gewählte Zeichen darf dabei nicht in dem Namen selber vorkommen. Es gibt auch keine Möglichkeit, dies mit einem Trick zu umgehen.
Bei den Namen, auch bei den Namen der Prädikate, sollten keine speziellen Symbole verwendet werden. Wird zum Beispiel ein `=` verwendet oder beginnt der Name mit `eq`, `Eq` oder `EQ`, so kann es passieren, dass der Name als Gleichheitseigenschaft interpretiert wird.

Ein weiteres Symbol, das verwendet werden kann, ist `.` (Punkt). Mit ihm werden in der Eingabedatei Ausdrücke beendet. Um Gruppierungen vorzunehmen, können die Symbole `( ) { } [` und `]` verwendet werden. Da in der Syntax von OTTER die gleichen Prioritäten wie in der Prädikatenlogik gelten, können Klammern teilweise weggelassen werden. Der Operator `<->` hat die höchste Priorität. Es folgen mit absteigenden Prioritäten `->`, `|`, `&` und `-`. Im Zweifelsfall oder in langen, komplizierten Formeln ist es aber oft sinnvoll, Klammern zu setzten.

3.2 Umformung der UML nach Otter

Die Umformung von UML-Diagrammen in die Syntax von OTTER ist relativ simpel, da nur prädikatenlogische Formeln verwendet werden können. So kann eine Vererbung als

```
all x (B(x) -> A(x)).
```

dargestellt werden. Die Bedingung, dass zwei vererbte Klassen `{disjoint}` sein sollen, kann mit folgender Formel ausgedrückt werden

```
all x (B(x)-> -C(x)).
```

Dass eine Vererbung als `{complete}` stattfinden soll, kann wie folgt umgewandelt werden

```
all x ((A(x) & B(x)) | (A(x) & C(x))).
```

Zuerst versuchte ich ein Vererbung mit der Bedingung `{complete}` wie folgt umzuwandeln, aber mit dieser Eingabe hatte OTTER Probleme.

```
all x (A(x) -> (B(x) | C(x))).
```

Die dazugehörige Ausgabe von OTTER ist auf der CD im Ordner `Otter/Ausgabedateien` in der Datei *GenDisCom-out.txt* zu finden.

Bei Klassen, die durch eine Assoziation R miteinander verbunden sind, wird die Umformung schwerer bis unmöglich, da in der Syntax von OTTER viele Tatsachen nicht umgeformt werden können. So können zum Beispiel nur Klassen deren Assoziationsenden jeweils die Multiplizität 1 besitzen umgeformt werden. Da die OTTER-Syntax keine Zähl-Quantoren enthält, können Assoziationen mit anderen Multiplizitäten nur sehr kompliziert umgeformt werden.

Eine Beschreibung der beiden Klassen mit jeweils der Multiplizität 1 sieht in der OTTER-Syntax wie folgt aus

```
all x,y (R(x,y) -> A(x) & B(y)).
all x (A(x) -> exists y (R(x,y))).
all y (B(y) -> exists x (R(x,y))).
```

OCL-Constraints oder andere UML-Elemente können nicht in die Syntax von OTTER umgeformt werden

3.3 Überprüfung

Die Überprüfung der OTTER-Eingabedateien erfolgt mit dem Starten von OTTER. Bei den Eingabedateien muss darauf geachtet werden, in welcher der so genannten Listen die Formeln beziehungsweise Klauseln enthalten sind. Es gibt drei verschiedene Listen für Formeln und vier verschiedene für Klauseln. Die drei Listen für die Formeln sind *formula_list(usable)*, *formula_list(sos)* (*set of support*) und *formula_list(passive)*. Alle in der Liste *formula_list(usable)* enthaltenen Formeln können für die Herleitung neuer Formeln verwendet werden. Die Formeln, die in der Liste *formula_list(sos)* enthalten sind, können hingegen nicht zur Herleitung neuer Formeln verwendet werden. Sie können lediglich zur Suche eines Beweises, beziehungsweise genauer gesagt eines Gegenbeweises, herangezogen werden. Die Formeln, die in der Liste *formula_list(passive)* enthalten sind,

werden nur im Falle eine Konflikts oder für eine Subsumierung benutzt. Ich habe für meine Überprüfungen nur die Listen *formula_list(usable)* und *formula_list(sos)* verwendet. In der Liste *usable* sind alle Formeln, die das System beschreiben, enthalten. In der Liste *sos* ist jeweils die Negation der zu beweisenden Formel enthalten.
Genauere Einzelheiten über die Listen der Formeln und Informationen über die verschiedenen Listen der Klauseln können in [McCu] nachgelesen werden.

Für die Überprüfung können verschiedene Flags gesetzt oder gelöscht werden. Alls vorhandenen Flags werden in [McCu] aufgelistet und erklärt. Ich habe lediglich das Flag `binary_res` gesetzt. Ist dieses Flag gesetzt, so wird zur Herleitung neuer Klauseln das Verfahren der Resolution benutzt. Beim Setzen werden auch automatisch die Flags `factor` und `unit_deletion` gesetzt. Das Flag `unit_deletion` bringt OTTER dazu, aus neu hergeleiteten Formeln Literale zu löschen, falls diese in negierter Form als Klausel in der Liste `usable` oder `sos` vorkommen. Das Flag `factor` dient zur Vereinfachung von neu hergeleiteten Klauseln, indem Teile einer Klausel, die die Klausel subsumieren, durch die einfachste Subsumierung ersetzt werden. Dies wird jedoch nur bei neu hergeleiteten Klauseln angewendet.

3.4 Resultate

Bei allen Dateien hat OTTER innerhalb kürzester Zeit als Beweis die leere Klausel herleiten können oder aber brach die Überprüfung ab, da keine verwendbaren Klauseln mehr existierten. Die benötigten Zeiten, die Anzahl der hergeleiteten Klauseln, die Länge und Tiefe des Beweises sind in Tabelle 8 aufgelistet. Alle diese Daten habe ich den Ausgabedateien entnommen. Die kompletten Eingabe- wie auch Ausgabedateien sind auf der beigelegten CD im Ordner `Otter/Eingabedateien` beziehungsweise im Ordner `Otter/Ausgabedateien` zu finden. Die Eingabedateien sind zusätzlich im Anhang D dargestellt.

Die Überprüfung der Umformung einer Vererbung zweier Klassen mit der Bedingung `{overlapping, incomplete}` war in der Hinsicht kompliziert, dass keine Formel existiert, mit der eine leere Klausel hergeleitet werden kann. In diesem Fall überprüfte ich, ob es Elemente geben darf, die nur Element von A, nicht aber von B oder C sein dürfen. Die Ausgabe von OTTER besagte, dass die Resolution abgebrochen wurde, da keine Formeln mehr zur Verfügung standen. Es gab also keinen Beweis, somit darf ein Element nur in A enthalten sein. Die gleiche Ausgabe erschien bei dem Versuch, zu testen, ob ein Element in den Mengen B und C gleichzeitig enthalten sein darf. Laut der Semantik der Bedingung `{overlapping, incomplete}` ist dies gestattet und somit ist die Überprüfung von OTTER korrekt.

Datei	benötigte Zeit	benötigter Speicher	hergeleitete Klauseln	Länge	Level	Bemerkung
Generalisierung-in.txt	0,00s	159kB	2	0	0	Beweis
Generalisierung-in2.txt	0,00s	159kB	2	1	1	Beweis
GenDisCom-in.txt	-	-	-	-	-	Fehler
GenDisCom-in2.txt	0,11s	191kB	1	1	1	Beweis
GenOverIncom-in.txt	0,41s	159kB	2	-	-	*sos empty*
GenOverIncom-in2.txt	0,00s	159kB	0	-	-	*sos empty*
Italians-in.txt	0,01s	191kB	5	4	1	Beweis
Italians-in2.txt	0,00s	191kB	8	-	-	*sos empty*
Persons-in.txt	0,00s	191kB	5	4	2	Beweis
Persons-in2.txt	0,11s	191kB	2	1	1	Beweis
Persons-in3.txt	0,08s	191kB	8	4	2	Beweis

Tabelle 8: benötigte Zeiten Otter

In dem in Abbildung 3 gezeigten Diagramm *Italians* sind alle *ItalianProfs LatinLovers*. Das Konzept der *LatinLover* subsumiert das Konzept der *ItalianProf*. Um diese Tatsache feststellen zu können, muss die Unerfüllbarkeit der folgenden Formel gezeigt werden.

```
all x (-LatinLover(x) & ItalianProf(x)).
```

Diesen Test führte ich in der Datei *Italians-in.txt* aus. Den Gegentest führte ich in der Datei *Italians-in2.txt* durch. Würde auch hier eine leere Klausel hergeleitet werden können, so wären die beiden Konzepte identisch. Dies war aber nicht der Fall, wie die Ausgabedatei *Italians-out2.txt* zeigt.

Den gleichen Test führte ich auch an der Spezifikation für das in Abbildung 4 dargestellte Diagramm durch, nachdem ich zeigte, dass keine *LatinLover* existieren können. Wenn bekannt ist, dass keine *LatinLover* laut diesem Diagramm existieren können, so ist schnell erkennbar, dass das Konzept *Lazy* und das Konzept *Italian* identisch sind. In den Dateien *Persons-in2.txt* und *Persons-in3.txt* führte ich die beiden Tests auf Subsumierung durch. Dass bei beiden die leere Klausel hergeleitet werden konnte, zeigt, dass die Konzepte identisch sind.

Das Programm OTTER hatte, mit einer Ausnahme, nicht die geringsten Probleme die von mir erstellten Dateien zu überprüfen. Alle Laufzeiten lagen bei unter einer Sekunde. Das Problem jedoch ist, die Formel zu finden, die bewie-

sen werden soll. Ohne genau zu wissen, auf was genau das Diagramm überprüft werden soll, ist es nicht möglich, eine solche Formel aufzustellen. Bei einfachen Diagrammen, wie zum Beispiel den Diagrammen *Persons* oder *Italians*, ist noch schnell erkennbar, auf was getestet werden muss. Bei komplexen Diagrammen hingegen wird dies nicht mehr möglich sein. Auch ist es nicht möglich, Attribute oder beliebige Operationen in eine prädikatenlogische Formel umzuformen. Es kann also lediglich auf eine Subsumierung getestet werden. Wird eine Sumbsumierung in beide Richtungen festgestellt, sind die beiden Konzepte äquivalent. Daher denke ich, dass dieses Verfahren allein für eine automatisierte Verifikation nicht geeignet ist.

4 Zusammenfassung

4.1 Resultate

Im Rahmen dieser Studienarbeit stellte ich Überlegungen an, wie eine automatisierte Verifikation von Software realisiert werden könnte. Mit Hilfe zwei verschiedener Verfahren überprüfte ich diese Überlegungen. Dafür wandelte ich jeweils fünf verschiedene UML-Diagramme in die Sprache TLA$^+$ und in die Syntax von OTTER um und ließ sie mit den Programmen TLC und OTTER überprüfen. Dabei stellte ich einige Probleme fest. UML-Diagramme mit OCL-Constraints lassen sich zwar relativ einfach in die Sprache TLA$^+$ umwandeln, aber TLC benötigt im Verhältnis viel Zeit und kann zum Beispiel keine Äquivalenzen oder Subsumierungen feststellen. Andererseits lieferte TLC eindeutige Ergebnisse und, falls ein Fehler auftrat, auch eine Beschreibung, an welcher Stelle der Fehler entstand.
Für die Umwandlung von UML-Diagrammen nach TLA$^+$ stellte ich noch einige Überlegungen an, wie eine Implementierung aussehen könnte.

Das Programm OTTER hingegen kann eine Subsumierung feststellen, jedoch ist es kompliziert, die zu beweisenden Formeln für komplexe Diagramme zu finden, denn dazu muss genau bekannt sein, auf welche Zusammenhänge getestet werden soll. Auch lassen sich lange nicht alle Elemente eines UML-Diagramms in die Syntax von OTTER umwandeln. Ein Vorteil dieses Programmes war jedoch die geringe Laufzeit. Innerhalb weniger als einer Sekunde erfolgte bei allen Dateien eine Ausgabe. Falls ein Fehler gefunden wurde, wurde auch hier eine Beschreibung mit ausgegeben.

Aus den gemachten Beobachtungen erstelle ich folgendes Fazit: Mit Hilfe der Sprache TLA$^+$ und dem Model-Checker TLC sollte es möglich sein, kleinere Softwarespezifikationen automatisch verifizieren zu können. Es kann aber nicht garantiert werden, dass alle Fehler oder Fehlkonstruktionen gefunden werden. Das Programm OTTER eignet sich jedoch nicht für eine automatische Verifizierung.

4.2 Ausblick

Als weitere Tests sollten TLA$^+$-Spezifikationen komplexerer UML-Diagramme erstellt und überprüft werden, um festzustellen, in wie fern das Programm TLC fähig ist, solche Beschreibungen zu verifizieren. Bereits bei recht einfachen UML-Diagrammen ist die Laufzeit, die der Model-Checker zur Überprüfung benötigt, relativ lang. Für ein Diagramm mit sieben Klassen und bei einer Belegung von maximal zehn Elementen pro Klasse beträgt die Laufzeit bereits über elf Minuten. Da die in der Praxis verwendeten UML-Diagramme für Software jedoch um einiges umfangreicher sind, werden sich höchstwahrscheinlich die Laufzeiten wesentlich erhöhen. An dieser Stelle sollte nach möglichen Erweiterungen oder Änderungen gesucht werden, die es ermöglichen würden die Verarbeitung zu beschleunigen. In wie weit dies realisierbar wäre, ist jedoch fraglich.
Würde dies getestet und stiegen die Laufzeiten nicht zu sehr an, so könnte versucht werden, die Umwandlung der UML- und OCL-Spezifikationen in die Sprache TLA$^+$ zu implementieren. Gelingt dies, so wäre als eine weitere denkbare Erweiterung, die Umformung auch auf andere Modelliersprachen auszudehnen.

Zur Zeit können mit dem Model-Checker TLC zwar Systemspezifikationen überprüft werden, doch dabei werden keine Äquivalenzen oder Subsumierungen erkannt. Mit dem Theorembeweiser OTTER hingegen können Systemspezifikationen auf Äquivalenzen und Sumsumierungen getestet werden, dies ist aber nicht für eine Automatisierung geeignet. Für eine komplette Überprüfung, bei der alle denkbaren Fehler gefunden werden, müssten das Verfahren des Model-Checkings und das der Deduktion kombiniert oder aber parallel verwendet werden. So könnten auch Designfehler in dem Ausgangsmodell erkannt werden.

Literatur

[KlWaBa04] Anneke Kleppe, Jos Warmer, Wim Bast *MDA Explained - The model driven Architecture: Practice and Promise*, Addison-Wesley, 3rd Printing, 2004

[StVö05] Thomas Stahl, Markus Völter *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management*, dpunkt.verlag, 1. Auflage, 2005

[Wiki1] http://de.wikipedia.org/wiki/Model-Checking, Stand vom 28.08.2006, 9:34 Uhr

[Schö05] Uwe Schöning *Logik für Informatiker*, Spektrum Akademischer Verlag, 5. Auflage, korrigierter Nachdruck 2005

[Wiki2] http://de.wikipedia.org/wiki/Deduktion, Stand vom 15.09.2006, 19:38 Uhr

[STS] Ralf Möller *Prädikatenlogik: Algorithmus für Erfüllbarkeitsproblem* http://www.sts.tu-harburg.de/%7Er.f.moeller/lectures/lgris-ws-05-06/LGRIS-06.pdf, 2005

[Lam02] Leslie Lamport *Specifying Systems - The TLA*$^+$ *Language and Tools for Hardware and Software Engineers*, Addison-Wesley, first printing, 2002, http://research.microsoft.com/users/lamport/tla/book-02-08-08.pdf

[McCu] William McCune *OTTER 3.3 Reference Manual*, August 2003, http://www-unix.mcs.anl.gov/AR/otter/otter33.pdf

[RuHaQu05] Chris Rupp, Jürgen Hahn, Stefan Queins, Mario Jeckle, Barbara Zengler *UML 2 glasklar - Praxiswissen für die UML-Modellierung und -Zertifizierung*, Hanser, 2. Auflage, 2005

[Fr05] Enrico Franconi *Introduction to Semantic Web Ontology Languages Formalising Ontologies*, 2005 http://trinity.dit.unitn.it/vikef/swap2005/formalising-ontologies.pdf

[Sch06UML] P.H. Schmitt *Formal Specification and Verification*, 2006, http://i12www.ira.uka.de/ pschmitt/FormSpez/Folien-2006/05UML-printversion.pdf

[WaKl03] Jos Warmer, Anneke Kleppe *The Object Constraint Language - Get Your Models Ready For MDA*, Addison-Wesley, Second Edition, 2003

[Ca03] Alan O'Callaghan *The Object Constraint Language - CSCI3007 Component Based Development*, 2.1.2003, http://www.cse.dmu.ac.uk/ aoc/teaching-notes/Contents/CSCI3007/OCL.ppt

[Sch06OCL] P.H. Schmitt *Formal Specification and Verification*, 2006, http://i12www.ira.uka.de/ pschmitt/FormSpez/Folien-2006/06OCL-printversion.pdf

[WaKl] Jos Warmer, Anneke Kleppe *The professional site of Jos Warmer and Anneke Kleppe*, http://www.klasse.nl/octopus/index.html

[Ul05] Christian Ullenboom *Java ist auch eine Insel - Programmieren für die Java 2 Plattform in der Version 5*, Galileo Computing, 4. Auflage, 2005

[Ko02] Helmut Kopka *LaTeX - Band 1: Einführung*, Pearson Studium, 3., überarbeitete Auflage, 2002

A TLA$^+$ Tools

A.1 Installation der TLA$^+$ Tools

Die Tools, die es zu TLA$^+$ gibt, können sehr einfach installiert werden. Voraussetzung für die Benutzung ist, dass Java auf dem PC installiert ist.
Zuerst muss die Datei *tla.zip* von Leslie Lamport Seite

http://research.microsoft.com/users/lamport/tla/tools.html

heruntergeladen und gespeichert werden. Dann muss ein neuer Ordner erstellt werden, in den die Datei entpackt wird. Angenommen, dieser Ordner hat den Pfad `D:\LeslieLamport`. Beim Entpacken wird ein neuer Unterordner `tla` erstellt, der wiederum einige Unterordner enthält. In jedem dieser Unterordner ist ein Tool beinhaltet. So sind auch ein Unterordner mit dem Namen `tlc` enthalten, der den Model-Checker enthält, sowie ein Unterordner mit dem Namen `tlatex`, der den Typesetter TLATEX enthält.
Schließlich muss noch der Pfad zu dem Ordner `tla` (hier `D:\LeslieLamport\tla`) zur Umgebungsvariable PATH hinzugefügt werden. Dies ist abhängig vom benutzten Betriebssystem. Bei neueren Windows-Systemen kann diese Einstellung unter *Start* / *Systemsteuerung* / *System* auf der Registerkarte *Erweitert* vorgenommen werden.

A.2 Benutzung von TLC

Wurde TLC wie oben beschrieben installiert, kann das Programm über die Konsole mit dem Befehl

```
java tlc.TLC Dateiname.tla
```

gestartet werden. Die Dateiendung *.tla* kann dabei weggelassen werden. Mögliche gewünschte Optionen müssen vor dem Dateinamen mit einem vorangehenden '-' angegeben werden. Eine Option ist zum Beispiel *-simulate*. Ist diese Option angegeben, läuft TLC im Simulationsmodus. In diesem Modus generiert TLC ein zufälliges Verhalten anstelle von allen möglichen Zuständen. Der Defaultwert für die Anzahl der berechneten Zustände liegt bei 100. Mit der zusätzlichen Option *-depth num* kann sonst ein beliebiger Integerwert *num* angegeben werden. Eine andere Option ist *-nowarning*. Ist sie angegeben, werden Warnungen bei erlaubten Ausdrücken, die aber zu Fehlern führen könnten, unterdrückt. Mit der Option *-cleanup* werden die von TLC erstellten Dateien nach Beendung der Verifikation wieder gelöscht.
Eine Auflistung und Beschreibung aller Optionen ist in [Lam02] zu finden.

A.3 Benutzung von TLATEX

TLATEX wird auch über die Konsole gestartet. Der Befehl dazu lautet

```
java tlatex.TLA Dateiname.tla
```

Auch hier kann die Dateiendung *.tla* weggelassen werden, falls die Dateiendung *.tla* lautet. Optionen können wie bei TLC vor dem Dateinamen angegeben werden. Mit dem Befehl *-ptsize 12* wird zum Beispiel die Schriftgröße 12 pt ausgewählt. Mit Angabe von *-shade* werden die Kommentare in der Ausgabedatei auf einem grauen Hintergrund ausgegeben. Es ist auch möglich, die Graustufe selber auszuwählen. Dies kann über den Befehl *-grayLevel num* geschehen, wobei für *num* ein Wert zwischen 0 und 1 gewählt werden kann. Der Wert 0 steht für schwarz, der Wert 1 für weiß. Der Defaultwert beträgt 0.85.
Weitere Befehle und Erklärungen dazu sind in [Lam02] aufgelistet.

B Otter

B.1 Installation von Otter

Zu OTTER existiert das einfache GUI *OtterFace8*, welches OTTER bereits beinhaltet. Es ist unter

http://www-unix.mcs.anl.gov/AR/otter/otterface8/

frei erhältlich. Gleichzeitig ist es auch ein GUI für Mace2.
Nach dem Entpacken kann durch Klicken auf die Datei *OtterFace8.jar* das GUI gestartet werden. Voraussetzung ist auch hier ein Java Runtime Environment.

Achtung: Unter Windows darf die Datei *OtterFace8.jar* nicht aus ihrem Ordner entfernt werden, da sonst die *.exe*-Dateien nicht gefunden werden können und so OTTER nicht gestartet werden kann. Dieses Problem kann einfach umgangen werden, indem eine Verknüpfung zu *OtterFace8.jar* erstellt wird, die an einen beliebigen Platz verschoben werden kann.

OTTER kann auch ohne GUI über die Kommandozeile gestartet werden. Die Installationsdatei hierfür kann unter

http://www-unix.mcs.anl.gov/AR/otter/dist33/

heruntergeladen werden. Nach dem Entpacken ist die Datei *otter.exe* im Unterordner `bin` zu finden.

B.2 Benutzung von Otter

Wurde das GUI OtterFace8 installiert, muss zum Starten nur auf die Datei *OtterFace8.jar* geklickt werden. Es öffnet sich dann das GUI. Oben links in der Menüleiste kann unter *File / Open (Select Input File)* die Eingabedatei ausgewählt werden. Über die beiden Radiobuttons kann gewählt werden, ob die Ausgabedatei auf dem Bildschirm ausgegeben oder direkt in eine Datei geschrieben werden soll. Mit Klick auf den Button *Start Otter* wird OTTER mit der ausgewählten Eingabedatei gestartet. Über Klick auf den Button *Cancel Otter Job* gibt es die Möglichkeit, die Überprüfung abzubrechen.

Soll OTTER ohne GUI benutzt werden, so kann das Programm über die Kommandozeile gestartet werden. Dazu muss zum Ordner `bin` navigiert werden. Dann kann mit dem Befehl

```
otter < Dateiname.in
```

OTTER mit der Eingabedatei *Dateiname.in* gestartet werden. Es sind zwei hohe Töne zu hören und die Ausgabe erscheint auf der Konsole. Um die Ausgabe in eine Datei mit Namen *Dateiname.out* schreiben zu lassen, muss folgender Befehl benutzt werden

```
otter < Dateiname.in > Dateiname.out
```

Die Dateinamen der Ein- und Ausgabedatei müssen dabei nicht identisch sein. Es ist auch möglich, dass die Dateiendungen der Ein- oder Ausgabedatei nicht *.in* bzw. *.out* lauten, sondern zum Beispiel *.txt*.

C TLA$^+$-Eingabedateien

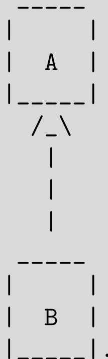
Die Datei Generalisierung.tla

MODULE *Generalisierung*

EXTENDS *TLC*, *Integers*, *Sequences*, *FiniteSets*

This TLA+ spec is the semantic mapping of a simple *UML* class model.

The class model consists of two classes A and *B* in package packname connected by an inheritance.



The following *Generalisierung.cfg* file can be used in model-checking this spec:

SPECIFICATION *Spec*

CONSTANT $Null = Null$

CONSTANT $A = \{a1, a2, a3, a4\}$
CONSTANT $B = \{b1, b2, b3, b4, b5, b6, b7, b8, b9, b10\}$

INVARIANT *Invariants*

The available populations of identifiers for instances of *UML* classes A and *B* is maintained in sets A, *B*. Making one such element member of *allInstances_packname_A* represents allocating it on the heap. The fact that it is allocated means that the same identifier cannot be allocated again, yet does not imply that it is reachable from some yet to be modelled root set of references.

The prefix *allInstances* hints at the fact that this formalization is amenable for extension to include *OCL*.

CONSTANT A, B
CONSTANT $Null$

VARIABLE $allInstances_packname_A,\ allInstances_packname_B$

$$InvariantAllInstances \triangleq \land\ allInstances_packname_A \subseteq (A \cup B)$$
$$\land\ allInstances_packname_B \subseteq B$$

$$InvariantInheritance \triangleq allInstances_packname_B \subseteq allInstances_packname_A$$

Conjunction of all invariants given by pieces

$$Invariants \triangleq \land InvariantAllInstances \\ \land InvariantInheritance$$

all *vars*, as needed for stuttering steps in the definition of *Spec* below

$vars \triangleq \langle allInstances_packname_A,\ allInstances_packname_B \rangle$

Operations

operations to add instances of A and *B*

$$\begin{aligned}
&\text{LOCAL } allocate_A(anA) \triangleq \\
&\quad \land anA \in A \\
&\quad \land anA \notin allInstances_packname_A \\
&\quad \land allInstances_packname_A' = allInstances_packname_A \cup \{anA\}
\end{aligned}$$

$$\begin{aligned}
&\text{LOCAL } allocate_B(aB) \triangleq \\
&\quad \land aB \in B \\
&\quad \land aB \notin allInstances_packname_A \\
&\quad \land aB \notin allInstances_packname_B \\
&\quad \land allInstances_packname_B' = allInstances_packname_B \cup \{aB\} \\
&\quad \land allInstances_packname_A' = allInstances_packname_A \cup \{aB\}
\end{aligned}$$

$$\begin{aligned}
&new_packname_B \triangleq \\
&\quad \land allInstances_packname_B \neq B \quad \text{\textit{i.e.} there are still non-allocated } IDs \\
&\quad \land \text{LET } aB \triangleq \text{CHOOSE } x \in B : x \notin allInstances_packname_B \\
&\qquad \text{IN } \ allocate_B(aB)
\end{aligned}$$

$$\begin{aligned}
&new_packname_A \triangleq \\
&\quad \land \neg(A \subseteq allInstances_packname_A) \quad \text{\textit{i.e.} there are still non-allocated } IDs \\
&\quad \land \text{LET } anA \triangleq \text{CHOOSE } x \in A : x \notin allInstances_packname_A \\
&\qquad \text{IN } \ \land allocate_A(anA) \\
&\qquad\quad\ \ \land \text{UNCHANGED } \langle allInstances_packname_B \rangle
\end{aligned}$$

operations to remove instances of A and *B*

$$\begin{aligned}
&remove_anOccurrenceOf_A(a) \triangleq \\
&\quad \text{IF } \land a \in A \\
&\qquad \land a \in allInstances_packname_A \\
&\qquad \land a \notin allInstances_packname_B \\
&\quad \text{THEN } \land allInstances_packname_A' = allInstances_packname_A \setminus \{a\} \\
&\qquad\quad \land \text{UNCHANGED } \langle allInstances_packname_B \rangle \\
&\quad \text{ELSE IF } \land a \in B \\
&\qquad\qquad \land a \in allInstances_packname_A \\
&\qquad\qquad \land a \in allInstances_packname_B \\
&\qquad\quad \text{THEN } \land allInstances_packname_A' = allInstances_packname_A \setminus \{a\} \\
&\qquad\qquad\quad \land allInstances_packname_B' = allInstances_packname_B \setminus \{a\} \\
&\qquad\quad \text{ELSE } \land \text{UNCHANGED } \langle allInstances_packname_A \rangle \\
&\qquad\qquad\quad \land \text{UNCHANGED } \langle allInstances_packname_B \rangle
\end{aligned}$$

$remove_anOccurrenceOf_B(b) \triangleq$
$\land\ b \in B$
$\land\ b \in allInstances_packname_A$
$\land\ b \in allInstances_packname_B$
$\land\ allInstances_packname_A' = allInstances_packname_A \setminus \{b\}$
$\land\ allInstances_packname_B' = allInstances_packname_B \setminus \{b\}$

What we know about initial states

Syntax and semantic analysis by TLA+ tools is very practical, *e.g.* it reports whether the initial state has not been completely specified

$Init \triangleq \land\ allInstances_packname_A = \{\}$
$\qquad \land\ allInstances_packname_B = \{\}$

$Next \triangleq \lor\ new_packname_A$
$\qquad \lor\ new_packname_B$
$\qquad \lor\ \exists\, a \in allInstances_packname_A :$
$\qquad\quad remove_anOccurrenceOf_A(a)$
$\qquad \lor\ \exists\, b \in allInstances_packname_B :$
$\qquad\quad remove_anOccurrenceOf_B(b)$

$Postcondition \triangleq \land\ allInstances_packname_A = (A \cup B)$
$\qquad \land\ allInstances_packname_B = B$

$Spec \triangleq Init \land \Box[Next]_{vars} \land \Diamond Postcondition$

THEOREM $Spec \Rightarrow \Box(Invariants)$

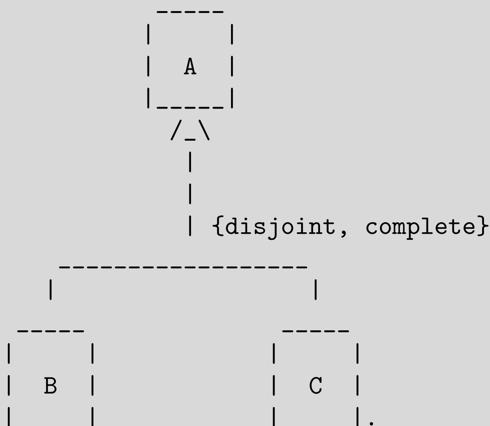
Die Datei GenDisCom.tla

MODULE *GenDisCom*

EXTENDS *TLC*, *Integers*, *Sequences*, *FiniteSets*

This TLA+ spec is the semantic mapping of a simple *UML* class model.

The class model consists of three classes A, *B* and *C* in package packname connected by an inheritance.



The following *GenDisCom.cfg* file can be used in model-checking this spec:

SPECIFICATION *Spec*

CONSTANT $Null = Null$

CONSTANT $A = \{\}$
CONSTANT $B = \{b1, b2, b3, b4, b5, b6, b7, b8, b9, b10\}$
CONSTANT $C = \{c1, c2, c3, c4, c5\}$

INVARIANT *Invariants*

The available populations of identifiers for instances of *UML* classes A ,*B* and *C* is maintained in sets A, *B* and *C*. Making one such element member of *allInstances_packname_A* represents allocating it on the heap. The fact that it is allocated means that the same identifier cannot be allocated again, yet does not imply that it is reachable from some yet to be modelled root set of references.

The prefix *allInstances* hints at the fact that this formalization is amenable for extension to include *OCL*.

CONSTANT A, B, C
CONSTANT *Null*

VARIABLE *allInstances_packname_A*, *allInstances_packname_B*, *allInstances_packname_C*

$$InvariantAllInstances \triangleq \begin{array}{l} \land\ allInstances_packname_A \subseteq (B \cup C) \\ \land\ allInstances_packname_B \subseteq B \\ \land\ allInstances_packname_C \subseteq C \end{array}$$

$InvariantInheritance \triangleq \land allInstances_packname_B \subseteq allInstances_packname_A$
$\qquad \land allInstances_packname_C \subseteq allInstances_packname_A$

$InvariantCompleteness \triangleq A = \{\}$

$InvariantDisjointness \triangleq (allInstances_packname_B \cap allInstances_packname_C) = \{\}$

Conjunction of all invariants given by pieces

$Invariants \triangleq \land InvariantAllInstances$
$\qquad \land InvariantInheritance$
$\qquad \land InvariantCompleteness$
$\qquad \land InvariantDisjointness$

all *vars*, as needed for stuttering steps in the definition of *Spec* below
$vars \triangleq \langle allInstances_packname_A, allInstances_packname_B, allInstances_packname_C \rangle$

Operations

operations to add instances of A and B

LOCAL $allocate_packname_B(aB) \triangleq$
$\land aB \in B$
$\land aB \notin allInstances_packname_A$
$\land aB \notin allInstances_packname_B$
$\land allInstances_packname_A' = allInstances_packname_A \cup \{aB\}$
$\land allInstances_packname_B' = allInstances_packname_B \cup \{aB\}$

LOCAL $allocate_packname_C(aC) \triangleq$
$\land aC \in C$
$\land aC \notin allInstances_packname_A$
$\land aC \notin allInstances_packname_C$
$\land allInstances_packname_A' = allInstances_packname_A \cup \{aC\}$
$\land allInstances_packname_C' = allInstances_packname_C \cup \{aC\}$

$new_packname_B \triangleq$
$\land B \neq allInstances_packname_B$ *i.e.* there are still non-allocated *IDs*
$\land$ LET $aB \triangleq$ CHOOSE $x \in B : x \notin allInstances_packname_B$
IN $\land allocate_packname_B(aB)$
$\quad \land$ UNCHANGED $\langle allInstances_packname_C \rangle$

$new_packname_C \triangleq$
$\land C \neq allInstances_packname_C$ *i.e.* there are still non-allocated *IDs*
$\land$ LET $aC \triangleq$ CHOOSE $x \in C : x \notin allInstances_packname_C$
IN $\land allocate_packname_C(aC)$
$\quad \land$ UNCHANGED $\langle allInstances_packname_B \rangle$

operations to remove instances of A and B

$remove_anOccurrenceOf_B(b) \triangleq$
$\land\ b \in B$
$\land\ b \in allInstances_packname_A$
$\land\ b \in allInstances_packname_B$
$\land\ allInstances_packname_A' = allInstances_packname_A \setminus \{b\}$
$\land\ allInstances_packname_B' = allInstances_packname_B \setminus \{b\}$
$\land$ UNCHANGED $\langle allInstances_packname_C \rangle$

$remove_anOccurrenceOf_C(c) \triangleq$
$\land\ c \in C$
$\land\ c \in allInstances_packname_A$
$\land\ c \in allInstances_packname_C$
$\land\ allInstances_packname_A' = allInstances_packname_A \setminus \{c\}$
$\land$ UNCHANGED $\langle allInstances_packname_B \rangle$
$\land\ allInstances_packname_C' = allInstances_packname_C \setminus \{c\}$

What we know about initial states

Syntax and semantic analysis by TLA+ tools is very practical, *e.g.* it reports whether the initial state has not been completely specified

$Init \triangleq \land\ allInstances_packname_A = \{\}$
$\qquad\quad \land\ allInstances_packname_B = \{\}$
$\qquad\quad \land\ allInstances_packname_C = \{\}$

$Next \triangleq \lor\ new_packname_B$
$\qquad\quad \lor\ new_packname_C$
$\qquad\quad \lor\ \exists\, b \in allInstances_packname_B :$
$\qquad\qquad remove_anOccurrenceOf_B(b)$
$\qquad\quad \lor\ \exists\, c \in allInstances_packname_C :$
$\qquad\qquad remove_anOccurrenceOf_C(c)$

$Postcondition \triangleq \land\ allInstances_packname_A = (B \cup C)$
$\qquad\qquad\qquad \land\ allInstances_packname_B = B$
$\qquad\qquad\qquad \land\ allInstances_packname_C = C$

$Spec \triangleq Init \land \Box[Next]_{vars} \land \Diamond Postcondition$

THEOREM $Spec \Rightarrow \Box(Invariants)$

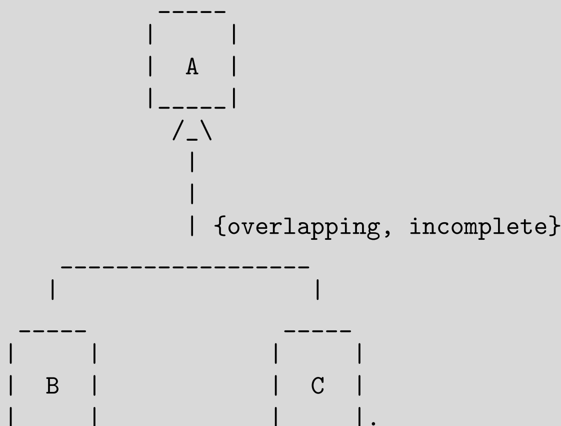
Die Datei GenOverIncom.tla

MODULE *GenOverIncom*

EXTENDS *TLC*, *Integers*, *Sequences*, *FiniteSets*

This TLA+ spec is the semantic mapping of a simple *UML* class model.

The class model consists of three classes A, *B* and *C* in package packname connected by an inheritance.



The following *GenOverIncom.cfg* file can be used in model-checking this spec:

SPECIFICATION *Spec*

CONSTANT $Null = Null$

CONSTANT $A = \{a1, a2, a3, a4\}$
CONSTANT $B = \{b1, b2, b3, b4, b5, b6, b7, b8, b9, b10\}$
CONSTANT $C = \{c1, c2, c3, c4, c5\}$

INVARIANT *Invariants*

The available populations of identifiers for instances of *UML* classes A ,*B* and *C* is maintained in sets A, *B* and *C*. Making one such element member of *allInstances_packname_A* represents allocating it on the heap. The fact that it is allocated means that the same identifier cannot be allocated again, yet does not imply that it is reachable from some yet to be modelled root set of references.

The prefix *allInstances* hints at the fact that this formalization is amenable for extension to include *OCL*.

CONSTANTS A, B, C
CONSTANT *Null*

VARIABLES *allInstances_packname_A*, *allInstances_packname_B*,
allInstances_packname_C

$InvariantAllInstances \triangleq \land allInstances_packname_A \subseteq (A \cup B \cup C)$
$\qquad \land allInstances_packname_B \subseteq B$
$\qquad \land allInstances_packname_C \subseteq C$

$InvariantInheritance \triangleq \land allInstances_packname_B \subseteq allInstances_packname_A$
$\qquad \land allInstances_packname_C \subseteq allInstances_packname_A$

Conjunction of all invariants given by pieces

$Invariants \triangleq \land InvariantAllInstances$
$\qquad \land InvariantInheritance$

all *vars*, as needed for stuttering steps in the definition of *Spec* below
$vars \triangleq \langle allInstances_packname_A, allInstances_packname_B,$
$\qquad allInstances_packname_C \rangle$

Operations

operations to add instances of A, B and C

LOCAL $allocate_packname_A(anA) \triangleq$
$\land anA \in A$
$\land anA \notin allInstances_packname_A$
$\land allInstances_packname_A' = allInstances_packname_A \cup \{anA\}$

LOCAL $allocate_packname_B(aB) \triangleq$
$\land aB \in B$
$\land aB \notin allInstances_packname_A$
$\land aB \notin allInstances_packname_B$
$\land allInstances_packname_A' = allInstances_packname_A \cup \{aB\}$
$\land allInstances_packname_B' = allInstances_packname_B \cup \{aB\}$

LOCAL $allocate_packname_C(aC) \triangleq$
$\land aC \in C$
$\land aC \notin allInstances_packname_A$
$\land aC \notin allInstances_packname_C$
$\land allInstances_packname_A' = allInstances_packname_A \cup \{aC\}$
$\land allInstances_packname_C' = allInstances_packname_C \cup \{aC\}$

$new_packname_A \triangleq$
$\land \neg(A \subseteq allInstances_packname_A)$ *i.e.* there are still non-allocated *IDs*
$\land$ LET $anA \triangleq$ CHOOSE $x \in A : x \notin allInstances_packname_A$
IN $\land allocate_packname_A(anA)$
$\quad \land$ UNCHANGED $\langle allInstances_packname_B \rangle$
$\quad \land$ UNCHANGED $\langle allInstances_packname_C \rangle$

$new_packname_B \triangleq$
$\land B \neq allInstances_packname_B$ *i.e.* there are still non-allocated *IDs*
$\land$ LET $aB \triangleq$ CHOOSE $x \in B : x \notin allInstances_packname_B$
IN $\land allocate_packname_B(aB)$
$\quad \land$ UNCHANGED $\langle allInstances_packname_C \rangle$

$new_packname_C \triangleq$

$\land\ C \neq allInstances_packname_C$ *i.e.* there are still non-allocated *IDs*

$\land$ LET $aC \triangleq$ CHOOSE $x \in C : x \notin allInstances_packname_C$

IN $\land\ allocate_packname_C(aC)$

$\land$ UNCHANGED $\langle allInstances_packname_B \rangle$

operations to remove instances of A, B and C

$remove_anOccurrenceOf_A(a) \triangleq$

IF $\land\ a \in A$

$\land\ a \in allInstances_packname_A$

THEN $\land\ allInstances_packname_A' = allInstances_packname_A \setminus \{a\}$

$\land$ UNCHANGED $\langle allInstances_packname_B \rangle$

$\land$ UNCHANGED $\langle allInstances_packname_C \rangle$

ELSE IF $\land\ a \in B$

$\land\ a \in allInstances_packname_A$

$\land\ a \in allInstances_packname_B$

THEN $\land\ allInstances_packname_A' = allInstances_packname_A \setminus \{a\}$

$\land\ allInstances_packname_B' = allInstances_packname_B \setminus \{a\}$

$\land$ UNCHANGED $\langle allInstances_packname_C \rangle$

ELSE IF $\land\ a \in C$

$\land\ a \in allInstances_packname_A$

$\land\ a \in allInstances_packname_C$

THEN $\land\ allInstances_packname_A' = allInstances_packname_A \setminus \{a\}$

$\land$ UNCHANGED $\langle allInstances_packname_B \rangle$

$\land\ allInstances_packname_C' = allInstances_packname_C \setminus \{a\}$

ELSE $\land$ UNCHANGED $\langle allInstances_packname_A \rangle$

$\land$ UNCHANGED $\langle allInstances_packname_B \rangle$

$\land$ UNCHANGED $\langle allInstances_packname_C \rangle$

$remove_anOccurrenceOf_B(b) \triangleq$

$\land\ b \in B$

$\land\ b \in allInstances_packname_A$

$\land\ b \in allInstances_packname_B$

$\land\ allInstances_packname_A' = allInstances_packname_A \setminus \{b\}$

$\land\ allInstances_packname_B' = allInstances_packname_B \setminus \{b\}$

$\land$ UNCHANGED $\langle allInstances_packname_C \rangle$

$remove_anOccurrenceOf_C(c) \triangleq$

$\land\ c \in C$

$\land\ c \in allInstances_packname_A$

$\land\ c \in allInstances_packname_C$

$\land\ allInstances_packname_A' = allInstances_packname_A \setminus \{c\}$

$\land$ UNCHANGED $\langle allInstances_packname_B \rangle$

$\land\ allInstances_packname_C' = allInstances_packname_C \setminus \{c\}$

What we know about initial states

Syntax and semantic analysis by TLA+ tools is very practical, *e.g.* it reports whether the initial state has not been completely specified

$$
\begin{aligned}
Init \triangleq\ & \land\ allInstances_packname_A = \{\} \\
& \land\ allInstances_packname_B = \{\} \\
& \land\ allInstances_packname_C = \{\}
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \lor\ new_packname_A \\
& \lor\ new_packname_B \\
& \lor\ new_packname_C \\
& \lor\ \exists\, a \in allInstances_packname_A : \\
& \quad remove_anOccurrenceOf_A(a) \\
& \lor\ \exists\, b \in allInstances_packname_B : \\
& \quad remove_anOccurrenceOf_B(b) \\
& \lor\ \exists\, c \in allInstances_packname_C : \\
& \quad remove_anOccurrenceOf_C(c)
\end{aligned}
$$

$$
\begin{aligned}
Postcondition \triangleq\ & \land\ allInstances_packname_A = (A \cup B \cup C) \\
& \land\ allInstances_packname_B = B \\
& \land\ allInstances_packname_C = C
\end{aligned}
$$

$$Spec \triangleq Init \land \Box[Next]_{vars} \land \Diamond Postcondition$$

THEOREM $Spec \Rightarrow \Box(Invariants)$

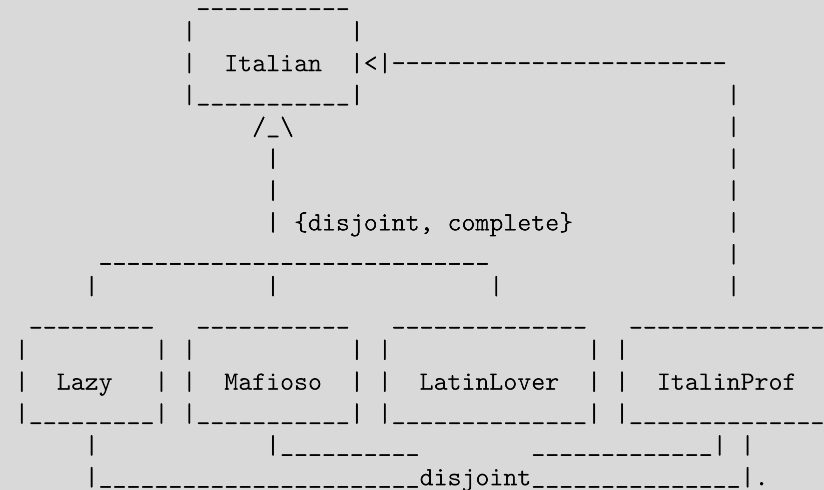
Die Datei Italians.tla

MODULE *Italians*

EXTENDS *TLC*, *Integers*, *Sequences*, *FiniteSets*

This TLA+ spec is the semantic mapping of a simple *UML* class model.

The class model consists of five classes.



The following *Italians.cfg* file can be used in model-checking this spec:

SPECIFICATION *Spec*

CONSTANT $Null = Null$

CONSTANT $Italian = \{\}$
CONSTANT $Lazy = \{L1, L2, L3, L4, L5, L6, L7, L8, L9, L10\}$
CONSTANT $Mafioso = \{M1, M2, M3, M4\}$
CONSTANT $LatinLover = \{LL1, LL2, LL3\}$
CONSTANT $ItalianProf = \{I1, I2\}$

INVARIANT *Invariants*

The available populations of identifiers for instances of the *UML* classes *Italian*, *Lazy*, *Mafioso*, *LatinLover* and *ItalianProf* is maintained in the sets *Italian*, *Lazy*, *Mafioso*, *LatinLover* and *ItalianProf*. Making one such element member of *allInstances_Italian*, ... represents allocating it on the heap. The fact that it is allocated means that the same identifier cannot be allocated again, yet does not imply that it is reachable from some yet to be modelled root set of references.

The prefix *allInstances* hints at the fact that this formalization is amenable for extension to include *OCL*.

CONSTANTS *Italian*, *Lazy*, *Mafioso*, *LatinLover*, *ItalianProf*
CONSTANT *Null*

VARIABLES *allInstances_Italian*, *allInstances_Lazy*, *allInstances_Mafioso*, *allInstances_LatinLover*, *allInstances_ItalianProf*

$$InvariantAllInstances \triangleq \land allInstances_Italian \subseteq (Lazy \cup Mafioso \cup LatinLover \cup ItalianProf) \land allInstances_Lazy \subseteq Lazy \land allInstances_Mafioso \subseteq Mafioso \land allInstances_LatinLover \subseteq LatinLover \land allInstances_ItalianProf \subseteq ItalianProf$$

$$InvariantInheritance \triangleq \land allInstances_Lazy \subseteq allInstances_Italian \land allInstances_Mafioso \subseteq allInstances_Italian \land allInstances_LatinLover \subseteq allInstances_Italian \land allInstances_ItalianProf \subseteq allInstances_Italian$$

$$InvariantCompleteness \triangleq Italian = \{\}$$

$$InvariantDisjointness \triangleq (allInstances_Lazy \cap allInstances_Mafioso \cap allInstances_LatinLover) = \{\}$$

$$InvariantAdditionalDisjointness \triangleq \land (allInstances_Lazy \cap allInstances_ItalianProf) = \{\} \land (allInstances_Mafioso \cap allInstances_ItalianProf) = \{\}$$

Conjunction of all invariants given by pieces

$$Invariants \triangleq \land InvariantAllInstances \land InvariantInheritance \land InvariantCompleteness \land InvariantDisjointness \land InvariantAdditionalDisjointness$$

all *vars*, as needed for stuttering steps in the definition of *Spec* below

$$vars \triangleq \langle allInstances_Italian, allInstances_Lazy, allInstances_Mafioso, allInstances_LatinLover, allInstances_ItalianProf \rangle$$

Operations

operations to add instances

$$\text{LOCAL } allocate_aLazy(aLazy) \triangleq \land aLazy \in Lazy \land aLazy \notin allInstances_Lazy \land aLazy \notin allInstances_Italian \land allInstances_Italian' = allInstances_Italian \cup \{aLazy\} \land allInstances_Lazy' = allInstances_Lazy \cup \{aLazy\}$$

$$\text{LOCAL } allocate_aMafioso(aMafioso) \triangleq \land aMafioso \in Mafioso \land aMafioso \notin allInstances_Mafioso \land aMafioso \notin allInstances_Italian \land allInstances_Italian' = allInstances_Italian \cup \{aMafioso\} \land allInstances_Mafioso' = allInstances_Mafioso \cup \{aMafioso\}$$

$$\text{LOCAL } allocate_aLatinLover(aLatinLover) \triangleq \land aLatinLover \in LatinLover \land aLatinLover \notin allInstances_LatinLover \land aLatinLover \notin allInstances_Italian \land allInstances_Italian' = allInstances_Italian \cup \{aLatinLover\} \land allInstances_LatinLover' = allInstances_LatinLover \cup \{aLatinLover\}$$

```
LOCAL allocate_anItalianProf(anItalianProf) ≜
  ∧ anItalianProf ∈ ItalianProf
  ∧ anItalianProf ∉ allInstances_ItalianProf
  ∧ anItalianProf ∉ allInstances_Italian
  ∧ allInstances_Italian′ = allInstances_Italian ∪ {anItalianProf}
  ∧ allInstances_ItalianProf′ = allInstances_ItalianProf ∪ {anItalianProf}

new_Lazy ≜
  ∧ Lazy ≠ allInstances_Lazy   i.e. there are still non-allocated IDs
  ∧ LET aLazy ≜ CHOOSE x ∈ Lazy : x ∉ allInstances_Lazy
     IN  ∧ allocate_aLazy(aLazy)
         ∧ UNCHANGED ⟨allInstances_Mafioso⟩
         ∧ UNCHANGED ⟨allInstances_LatinLover⟩
         ∧ UNCHANGED ⟨allInstances_ItalianProf⟩

new_Mafioso ≜
  ∧ Mafioso ≠ allInstances_Mafioso   i.e. there are still non-allocated IDs
  ∧ LET aMafioso ≜ CHOOSE x ∈ Mafioso : x ∉ allInstances_Mafioso
     IN  ∧ allocate_aMafioso(aMafioso)
         ∧ UNCHANGED ⟨allInstances_Lazy⟩
         ∧ UNCHANGED ⟨allInstances_LatinLover⟩
         ∧ UNCHANGED ⟨allInstances_ItalianProf⟩

new_LatinLover ≜
  ∧ LatinLover ≠ allInstances_LatinLover   i.e. there are still non-allocated IDs
  ∧ LET aLatinLover ≜ CHOOSE x ∈ LatinLover : x ∉ allInstances_LatinLover
     IN  ∧ allocate_aLatinLover(aLatinLover)
         ∧ UNCHANGED ⟨allInstances_Lazy⟩
         ∧ UNCHANGED ⟨allInstances_Mafioso⟩
         ∧ UNCHANGED ⟨allInstances_ItalianProf⟩

new_ItalianProf ≜
  ∧ ItalianProf ≠ allInstances_ItalianProf   i.e. there are still non-allocated IDs
  ∧ LET anItalianProf ≜ CHOOSE x ∈ ItalianProf : x ∉ allInstances_ItalianProf
     IN  ∧ allocate_anItalianProf(anItalianProf)
         ∧ UNCHANGED ⟨allInstances_Lazy⟩
         ∧ UNCHANGED ⟨allInstances_LatinLover⟩
         ∧ UNCHANGED ⟨allInstances_Mafioso⟩

operations to remove instances

remove_aLazy(l) ≜
  ∧ l ∈ allInstances_Italian
  ∧ l ∈ allInstances_Lazy
  ∧ allInstances_Italian′ = allInstances_Italian \ {l}
  ∧ allInstances_Lazy′ = allInstances_Lazy \ {l}
  ∧ UNCHANGED ⟨allInstances_Mafioso⟩
  ∧ UNCHANGED ⟨allInstances_LatinLover⟩
  ∧ UNCHANGED ⟨allInstances_ItalianProf⟩
```

$remove_aMafioso(m) \triangleq$
$\land\ m \in allInstances_Italian$
$\land\ m \in allInstances_Mafioso$
$\land\ allInstances_Italian' = allInstances_Italian \setminus \{m\}$
$\land$ UNCHANGED $\langle allInstances_Lazy \rangle$
$\land\ allInstances_Mafioso' = allInstances_Mafioso \setminus \{m\}$
$\land$ UNCHANGED $\langle allInstances_LatinLover \rangle$
$\land$ UNCHANGED $\langle allInstances_ItalianProf \rangle$

$remove_aLatinLover(ll) \triangleq$
$\land\ ll \in allInstances_Italian$
$\land\ ll \in allInstances_LatinLover$
$\land\ allInstances_Italian' = allInstances_Italian \setminus \{ll\}$
$\land$ UNCHANGED $\langle allInstances_Lazy \rangle$
$\land$ UNCHANGED $\langle allInstances_Mafioso \rangle$
$\land\ allInstances_LatinLover' = allInstances_LatinLover \setminus \{ll\}$
$\land$ UNCHANGED $\langle allInstances_ItalianProf \rangle$

$remove_anItalianProf(ip) \triangleq$
$\land\ ip \in allInstances_Italian$
$\land\ ip \in allInstances_ItalianProf$
$\land\ allInstances_Italian' = allInstances_Italian \setminus \{ip\}$
$\land$ UNCHANGED $\langle allInstances_Lazy \rangle$
$\land$ UNCHANGED $\langle allInstances_Mafioso \rangle$
$\land$ UNCHANGED $\langle allInstances_LatinLover \rangle$
$\land\ allInstances_ItalianProf' = allInstances_ItalianProf \setminus \{ip\}$

What we know about initial states

Syntax and semantic analysis by TLA+ tools is very practical, *e.g.* it reports whether the initial state has not been completely specified

$Init \triangleq$
$\land\ allInstances_Italian = \{\}$
$\land\ allInstances_Lazy = \{\}$
$\land\ allInstances_Mafioso = \{\}$
$\land\ allInstances_LatinLover = \{\}$
$\land\ allInstances_ItalianProf = \{\}$

$Next \triangleq$
$\lor\ new_Lazy$
$\lor\ new_Mafioso$
$\lor\ new_LatinLover$
$\lor\ new_ItalianProf$
$\lor\ \exists\, l \in allInstances_Lazy :$
$remove_aLazy(l)$
$\lor\ \exists\, m \in allInstances_Mafioso :$
$remove_aMafioso(m)$
$\lor\ \exists\, ll \in allInstances_LatinLover :$
$remove_aLatinLover(ll)$
$\lor\ \exists\, ip \in allInstances_ItalianProf :$
$remove_anItalianProf(ip)$

$$
\begin{aligned}
Postcondition \triangleq\ & \land\ allInstances_Italian = (Lazy \cup Mafioso \cup LatinLover \cup ItalianProf) \\
& \land\ allInstances_Lazy = Lazy \\
& \land\ allInstances_Mafioso = Mafioso \\
& \land\ allInstances_LatinLover = LatinLover \\
& \land\ allInstances_ItalianProf = ItalianProf
\end{aligned}
$$

$$Spec \triangleq Init \land \Box[Next]_{vars} \land \Diamond Postcondition$$

THEOREM $Spec \Rightarrow \Box(Invariants)$

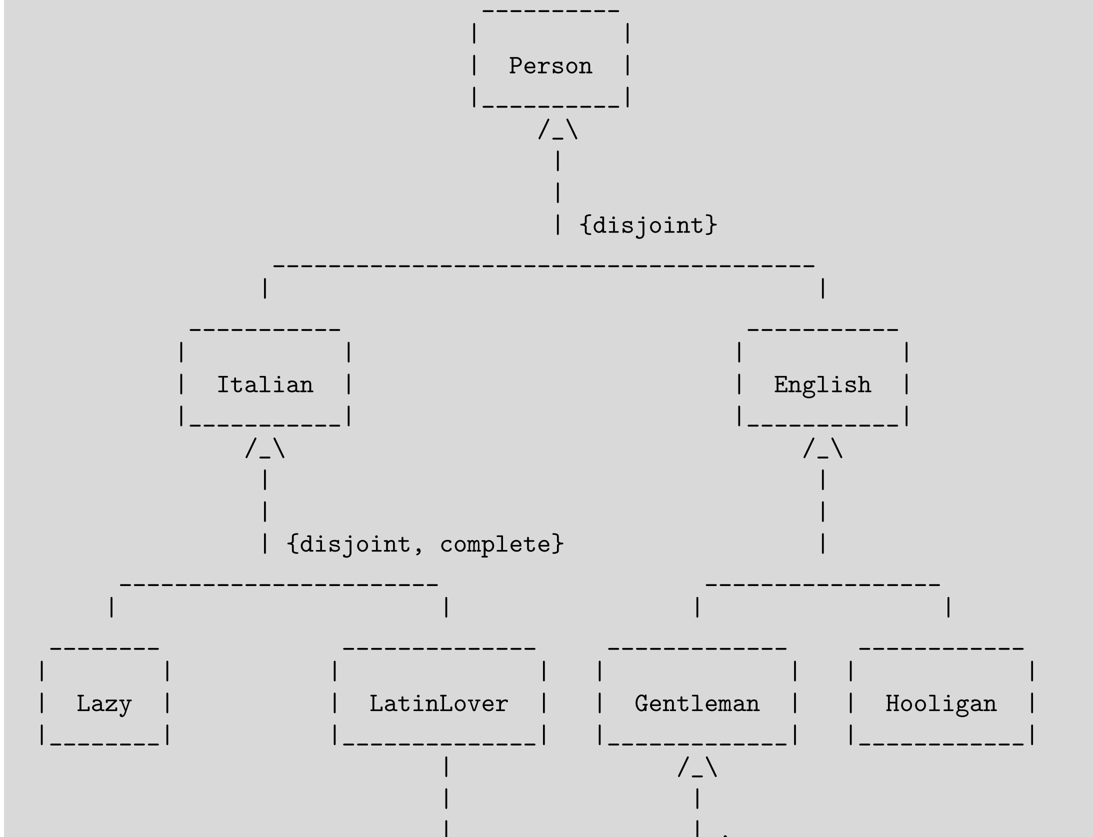
Die Datei Persons.tla

MODULE *Persons*

EXTENDS *TLC*, *Integers*, *Sequences*, *FiniteSets*

This TLA+ spec is the semantic mapping of a simple *UML* class model.

The class model consists of seven classes in package packname connected by some inheritance.



The following *Persons.cfg* file can be used in model-checking this spec:

SPECIFICATION *Spec*

CONSTANT $Null = Null$

CONSTANT $Person = \{p1, p2, p3\}$
CONSTANT $Italian = \{\}$
CONSTANT $Lazy = \{l1, l2, l3, l4\}$
CONSTANT $LatinLover = \{ll1, ll2, ll3\}$
CONSTANT $English = \{e1, e2, e3, e4, e5\}$
CONSTANT $Gentleman = \{g1, g2, g3\}$
CONSTANT $Hooligan = \{h1, h2, h3, h4, h5, h6\}$

INVARIANT *Invariants*

The available populations of identifiers for instances of *UML* classes is maintained in sets. Making one such element member of *allInstances_class* represents allocating it on the heap. The fact that it is allocated means that the same identifier cannot be allocated again, yet does not imply that it is reachable from some yet to be modelled root set of references.

The prefix *allInstances* hints at the fact that this formalization is amenable for extension to include *OCL*.

CONSTANT $Person$, $Italian$, $Lazy$, $LatinLover$, $English$, $Gentleman$, $Hooligan$
CONSTANT $Null$

VARIABLE $allInstances_Person$, $allInstances_Italian$, $allInstances_Lazy$, $allInstances_LatinLover$
VARIABLE $allInstances_English$, $allInstances_Gentleman$, $allInstances_Hooligan$

$$
\begin{aligned}
& InvariantAllInstances \triangleq \\
& \quad \land\ allInstances_Person \subseteq \\
& \qquad (Person \cup Lazy \cup LatinLover \cup English \cup Gentleman \cup Hooligan) \\
& \quad \land\ allInstances_Italian \subseteq (Lazy \cup LatinLover) \\
& \quad \land\ allInstances_English \subseteq (English \cup Gentleman \cup Hooligan \cup LatinLover) \\
& \quad \land\ allInstances_Lazy \subseteq Lazy \\
& \quad \land\ allInstances_LatinLover \subseteq LatinLover \\
& \quad \land\ allInstances_Gentleman \subseteq (Gentleman \cup LatinLover) \\
& \quad \land\ allInstances_Hooligan \subseteq Hooligan
\end{aligned}
$$

Invariants for inheritance of class *Person*

$$
\begin{aligned}
InvariantInheritancePerson \triangleq\ & \land\ allInstances_Italian \subseteq allInstances_Person \\
& \land\ allInstances_English \subseteq allInstances_Person
\end{aligned}
$$

$$InvariantDisjointnessPerson \triangleq (allInstances_Italian \cap allInstances_English) = \{\}$$

$$
\begin{aligned}
InvariantPerson \triangleq\ & \land\ InvariantInheritancePerson \\
& \land\ InvariantDisjointnessPerson
\end{aligned}
$$

Invariants for inheritance of class *Italian*

$$
\begin{aligned}
InvariantInheritanceItalian \triangleq\ & \land\ allInstances_Lazy \subseteq allInstances_Italian \\
& \land\ allInstances_LatinLover \subseteq allInstances_Italian
\end{aligned}
$$

$$InvariantCompletenessItalian \triangleq Italian = \{\}$$

$$InvariantDisjointnessItalian \triangleq (allInstances_Lazy \cap allInstances_LatinLover) = \{\}$$

$$
\begin{aligned}
InvariantItalian \triangleq\ & \land\ InvariantInheritanceItalian \\
& \land\ InvariantCompletenessItalian \\
& \land\ InvariantDisjointnessItalian
\end{aligned}
$$

Invariant for inheritance of class *English*

$$
\begin{aligned}
InvariantInheritanceEnglish \triangleq\ & \land\ allInstances_Gentleman \subseteq allInstances_English \\
& \land\ allInstances_Hooligan \subseteq allInstances_English
\end{aligned}
$$

$$InvariantEnglish \triangleq InvariantInheritanceEnglish$$

Invariant for inheritance of class *Gentleman*

$$InvariantInheritanceGentleman \triangleq allInstances_LatinLover \subseteq allInstances_Gentleman$$

$$InvariantGentleman \triangleq InvariantInheritanceGentleman$$

Conjunction of all invariants given by pieces

$$
\begin{aligned}
Invariants \triangleq\ & \land\ InvariantAllInstances \\
& \land\ InvariantPerson \\
& \land\ InvariantItalian \\
& \land\ InvariantEnglish \\
& \land\ InvariantGentleman
\end{aligned}
$$

all *vars*, as needed for stuttering steps in the definition of *Spec* below

$$
\begin{aligned}
vars \triangleq\ & \langle allInstances_Person,\ allInstances_Italian,\ allInstances_English,\ allInstances_Lazy, \\
& allInstances_LatinLover,\ allInstances_Gentleman,\ allInstances_Hooligan \rangle
\end{aligned}
$$

Operations

operations to add instances

$$
\begin{aligned}
& \text{LOCAL } allocate_aPerson(aPerson) \triangleq \\
& \quad \land\ allInstances_Person' = allInstances_Person \cup \{aPerson\}
\end{aligned}
$$

$$
\begin{aligned}
& \text{LOCAL } allocate_aLazy(aLazy) \triangleq \\
& \quad \land\ allInstances_Person' = allInstances_Person \cup \{aLazy\} \\
& \quad \land\ allInstances_Italian' = allInstances_Italian \cup \{aLazy\} \\
& \quad \land\ allInstances_Lazy' = allInstances_Lazy \cup \{aLazy\}
\end{aligned}
$$

$$
\begin{aligned}
& \text{LOCAL } allocate_aLatinLover(aLatinLover) \triangleq \\
& \quad \land\ allInstances_Person' = allInstances_Person \cup \{aLatinLover\} \\
& \quad \land\ allInstances_Italian' = allInstances_Italian \cup \{aLatinLover\} \\
& \quad \land\ allInstances_LatinLover' = allInstances_LatinLover \cup \{aLatinLover\} \\
& \quad \land\ allInstances_Gentleman' = allInstances_Gentleman \cup \{aLatinLover\} \\
& \quad \land\ allInstances_English' = allInstances_English \cup \{aLatinLover\}
\end{aligned}
$$

$$
\begin{aligned}
& \text{LOCAL } allocate_anEnglish(anEnglish) \triangleq \\
& \quad \land\ allInstances_Person' = allInstances_Person \cup \{anEnglish\} \\
& \quad \land\ allInstances_English' = allInstances_English \cup \{anEnglish\}
\end{aligned}
$$

$$
\begin{aligned}
& \text{LOCAL } allocate_aGentleman(aGentleman) \triangleq \\
& \quad \land\ allInstances_Person' = allInstances_Person \cup \{aGentleman\} \\
& \quad \land\ allInstances_English' = allInstances_English \cup \{aGentleman\} \\
& \quad \land\ allInstances_Gentleman' = allInstances_Gentleman \cup \{aGentleman\}
\end{aligned}
$$

$$
\begin{aligned}
& \text{LOCAL } allocate_aHooligan(aHooligan) \triangleq \\
& \quad \land\ allInstances_Person' = allInstances_Person \cup \{aHooligan\} \\
& \quad \land\ allInstances_English' = allInstances_English \cup \{aHooligan\} \\
& \quad \land\ allInstances_Hooligan' = allInstances_Hooligan \cup \{aHooligan\}
\end{aligned}
$$

$$
\begin{aligned}
& new_Lazy \triangleq \\
& \quad \land\ Lazy \neq allInstances_Lazy \quad \textit{i.e.} \text{ there are still non-allocated } IDs \\
& \quad \land\ \text{LET } aLazy \triangleq \text{CHOOSE } x \in Lazy : x \notin allInstances_Lazy \\
& \qquad \text{IN} \quad \land\ allocate_aLazy(aLazy) \\
& \qquad\qquad \land\ \text{UNCHANGED } \langle allInstances_LatinLover \rangle \\
& \qquad\qquad \land\ \text{UNCHANGED } \langle allInstances_English \rangle \\
& \qquad\qquad \land\ \text{UNCHANGED } \langle allInstances_Gentleman \rangle \\
& \qquad\qquad \land\ \text{UNCHANGED } \langle allInstances_Hooligan \rangle
\end{aligned}
$$

$$
\begin{array}{l}
new_LatinLover \triangleq \\
\quad \land\ allInstances_LatinLover \neq LatinLover \quad \text{\textit{i.e.} there are still non-allocated \textit{IDs}} \\
\quad \land\ \textsc{let}\ aLatinLover \triangleq \textsc{choose}\ x \in LatinLover : x \notin allInstances_LatinLover \\
\quad\quad \textsc{in}\ \ \land\ allocate_aLatinLover(aLatinLover) \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Lazy \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Hooligan \rangle
\end{array}
$$

$$
\begin{array}{l}
new_Gentleman \triangleq \\
\quad \land\ \neg(Gentleman \subseteq allInstances_Gentleman) \quad \text{\textit{i.e.} there are still non-allocated \textit{IDs}} \\
\quad \land\ \lor\ \textsc{let}\ aGentleman \triangleq \textsc{choose}\ x \in Gentleman : x \notin allInstances_Gentleman \\
\quad\quad\quad \textsc{in}\ \ \land\ allocate_aGentleman(aGentleman) \\
\quad\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Italian \rangle \\
\quad\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Lazy \rangle \\
\quad\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_LatinLover \rangle \\
\quad\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Hooligan \rangle
\end{array}
$$

$$
\begin{array}{l}
new_Hooligan \triangleq \\
\quad \land\ allInstances_Hooligan \neq Hooligan \quad \text{\textit{i.e.} there are still non-allocated \textit{IDs}} \\
\quad \land\ \textsc{let}\ aHooligan \triangleq \textsc{choose}\ x \in Hooligan : x \notin allInstances_Hooligan \\
\quad\quad \textsc{in}\ \ \land\ allocate_aHooligan(aHooligan) \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Italian \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Lazy \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_LatinLover \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Gentleman \rangle
\end{array}
$$

$$
\begin{array}{l}
new_English \triangleq \\
\quad \land\ \neg(English \subseteq allInstances_English) \quad \text{\textit{i.e.} there are still non-allocated \textit{IDs}} \\
\quad \land\ \textsc{let}\ anEnglish \triangleq \textsc{choose}\ x \in English : x \notin allInstances_English \\
\quad\quad \textsc{in}\ \ \land\ allocate_anEnglish(anEnglish) \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Italian \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Lazy \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_LatinLover \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Gentleman \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Hooligan \rangle
\end{array}
$$

$$
\begin{array}{l}
new_Person \triangleq \\
\quad \land\ \neg(Person \subseteq allInstances_Person) \quad \text{\textit{i.e.} there are still non-allocated \textit{IDs}} \\
\quad \land\ \textsc{let}\ aPerson \triangleq \textsc{choose}\ x \in Person : x \notin allInstances_Person \\
\quad\quad \textsc{in}\ \ \land\ allocate_aPerson(aPerson) \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Italian \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Lazy \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_LatinLover \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_English \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Gentleman \rangle \\
\quad\quad\quad\ \ \land\ \textsc{unchanged}\ \langle allInstances_Hooligan \rangle
\end{array}
$$

operations to remove instances

$remove_aPerson(p) \triangleq$
$\land\ p \in Person$
$\land\ p \in allInstances_Person$
$\land\ allInstances_Person' = allInstances_Person \setminus \{p\}$
$\land$ UNCHANGED $\langle allInstances_Italian \rangle$
$\land$ UNCHANGED $\langle allInstances_LatinLover \rangle$
$\land$ UNCHANGED $\langle allInstances_Lazy \rangle$
$\land$ UNCHANGED $\langle allInstances_English \rangle$
$\land$ UNCHANGED $\langle allInstances_Gentleman \rangle$
$\land$ UNCHANGED $\langle allInstances_Hooligan \rangle$

$remove_aLazy(l) \triangleq$
$\land\ l \in Lazy$
$\land\ l \in allInstances_Lazy$
$\land\ allInstances_Lazy' = allInstances_Lazy \setminus \{l\}$
$\land\ allInstances_Italian' = allInstances_Italian \setminus \{l\}$
$\land\ allInstances_Person' = allInstances_Person \setminus \{l\}$
$\land$ UNCHANGED $\langle allInstances_LatinLover \rangle$
$\land$ UNCHANGED $\langle allInstances_English \rangle$
$\land$ UNCHANGED $\langle allInstances_Gentleman \rangle$
$\land$ UNCHANGED $\langle allInstances_Hooligan \rangle$

$remove_aLatinLover(ll) \triangleq$
$\land\ ll \in LatinLover$
$\land\ ll \in allInstances_LatinLover$
$\land\ allInstances_LatinLover' = allInstances_LatinLover \setminus \{ll\}$
$\land\ allInstances_Italian' = allInstances_Italian \setminus \{ll\}$
$\land\ allInstances_Person' = allInstances_Person \setminus \{ll\}$
$\land\ allInstances_Gentleman' = allInstances_Gentleman \setminus \{ll\}$
$\land\ allInstances_English' = allInstances_English \setminus \{ll\}$
$\land$ UNCHANGED $\langle allInstances_Lazy \rangle$
$\land$ UNCHANGED $\langle allInstances_Hooligan \rangle$

$remove_anEnglish(e) \triangleq$
$\land\ e \in English$
$\land\ e \in allInstances_English$
$\land\ allInstances_English' = allInstances_English \setminus \{e\}$
$\land\ allInstances_Person' = allInstances_Person \setminus \{e\}$
$\land$ UNCHANGED $\langle allInstances_Italian \rangle$
$\land$ UNCHANGED $\langle allInstances_Lazy \rangle$
$\land$ UNCHANGED $\langle allInstances_LatinLover \rangle$
$\land$ UNCHANGED $\langle allInstances_Gentleman \rangle$
$\land$ UNCHANGED $\langle allInstances_Hooligan \rangle$

$remove_aGentleman(g) \triangleq$

$\land g \in Gentleman$
$\land g \in allInstances_Gentleman$
$\land allInstances_Gentleman' = allInstances_Gentleman \setminus \{g\}$
$\land allInstances_English' = allInstances_English \setminus \{g\}$
$\land allInstances_Person' = allInstances_Person \setminus \{g\}$
$\land$ UNCHANGED $\langle allInstances_Italian \rangle$
$\land$ UNCHANGED $\langle allInstances_Lazy \rangle$
$\land$ UNCHANGED $\langle allInstances_LatinLover \rangle$
$\land$ UNCHANGED $\langle allInstances_Hooligan \rangle$

$remove_aHooligan(p) \triangleq$

$\land p \in Hooligan$
$\land p \in allInstances_Hooligan$
$\land allInstances_Hooligan' = allInstances_Hooligan \setminus \{p\}$
$\land allInstances_English' = allInstances_English \setminus \{p\}$
$\land allInstances_Person' = allInstances_Person \setminus \{p\}$
$\land$ UNCHANGED $\langle allInstances_Italian \rangle$
$\land$ UNCHANGED $\langle allInstances_Lazy \rangle$
$\land$ UNCHANGED $\langle allInstances_LatinLover \rangle$
$\land$ UNCHANGED $\langle allInstances_Gentleman \rangle$

What we know about initial states

Syntax and semantic analysis by TLA+ tools is very practical, *e.g.* it reports whether the initial state has not been completely specified

$Init \triangleq$
$\land allInstances_Person = \{\}$
$\land allInstances_Italian = \{\}$
$\land allInstances_English = \{\}$
$\land allInstances_Lazy = \{\}$
$\land allInstances_LatinLover = \{\}$
$\land allInstances_Gentleman = \{\}$
$\land allInstances_Hooligan = \{\}$

$$
\begin{aligned}
Next \triangleq\ & \lor\ new_Person \\
& \lor\ new_Lazy \\
& \lor\ new_LatinLover \\
& \lor\ new_English \\
& \lor\ new_Gentleman \\
& \lor\ new_Hooligan \\
& \lor\ \exists\, p \in allInstances_Person : \\
& \quad remove_aPerson(p) \\
& \lor\ \exists\, l \in allInstances_Lazy : \\
& \quad remove_aLazy(l) \\
& \lor\ \exists\, ll \in allInstances_LatinLover : \\
& \quad remove_aLatinLover(ll) \\
& \lor\ \exists\, e \in allInstances_English : \\
& \quad remove_anEnglish(e) \\
& \lor\ \exists\, g \in allInstances_Gentleman : \\
& \quad remove_aGentleman(g) \\
& \lor\ \exists\, h \in allInstances_Hooligan : \\
& \quad remove_aHooligan(h)
\end{aligned}
$$

$$
\begin{aligned}
Postcondition \triangleq\ & \land\ allInstances_Person = Person \cup allInstances_Italian \cup allInstances_English \\
& \land\ allInstances_Italian = Lazy \cup LatinLover \\
& \land\ allInstances_Lazy = Lazy \\
& \land\ allInstances_LatinLover = LatinLover \\
& \land\ allInstances_English = English \cup allInstances_Gentleman \cup Hooligan \\
& \land\ allInstances_Gentleman = Gentleman \cup LatinLover \\
& \land\ allInstances_Hooligan = Hooligan
\end{aligned}
$$

$$Spec \triangleq Init \land \Box[Next]_{vars} \land \Diamond Postcondition$$

THEOREM $Spec \Rightarrow \Box(Invariants)$

Die Datei AB.tla, geschrieben von Miguel Garcia

MODULE *AB*

EXTENDS *TLC*, *Integers*, *Sequences*, *FiniteSets*

This TLA+ spec is the semantic mapping of a simple *UML* class model.

The class model consists of two classes A and B in package packname connected by an association D which is bidirectional. On the A side, it is $\{ordered\}$ and allows duplicates (as per $\{notUnique\}$) on the A side, with multiplicity boundaries 'amin' and 'amax'.

As to the B side, no explicit tags are given, thus declaring multiplicity $0 \mathrel{..}$ unlimited , no ordering and disallowing duplicates.

Operations are defined to allocate objects and manipulate links. The spec as a whole guarantees that for any sequence of operations, the invariants specified in the class model (about multiplicities, ordering, and bidirectionality) are maintained. This guarantee can be model checked for arbitrary finite populations of instances of A and B.

The following *AB.cfg* file can be used in model-checking this spec:

SPECIFICATION *Spec*

CONSTANT $Null = Null$

CONSTANT $A = \{a1,\ a2,\ a3,\ a4\}$
CONSTANT $B = \{b1,\ b2,\ b3,\ b4,\ b5,\ b6,\ b7,\ b8,\ b9,\ b10\}$

CONSTANT $amin = 2$
CONSTANT $amax = 6$

INVARIANT *Invariants*

The available populations of identifiers for instances of *UML* classes A and B is maintained in sets A, B. Making one such element member of *allInstances_packname_A* represents allocating it on the heap. The fact that it is allocated means that the same identifier cannot be allocated again, yet does not imply that it is reachable from some yet to be modelled root set of references.

The prefix *allInstances* hints at the fact that this formalization is amenable for extension to include *OCL*.

CONSTANT A, B
CONSTANT *amin*, *amax*
CONSTANT *Null*

VARIABLE *allInstances_packname_A*, *allInstances_packname_B*

Two total functions allow retrieving the instances reachable over the association in each direction. Invariants are added below to capture all relevant *UML* constraints (*e.g.* bi-directionality)

Two functions are used instead of a single global relation in order to show the kind of pre, post and frame conditions that have to be specified so that the invariants are maintained. Doing so makes the spec closer to program verification.

VARIABLE *AsForBoverD*, *BsForAoverD*

Helper predicates

sequence to set

$SeqToSet(seq) \triangleq \{seq[x] : x \in 1 \,..\, Len(seq)\}$

whether *elem* shows up in non-empty sequence *seq*

$ElemIsInSeq(elem, seq) \triangleq$
$\quad elem \in SeqToSet(seq)$

Invariants on structural aspects

$InvariantAllInstances \triangleq \land allInstances_packname_A \subseteq A$
$\qquad\qquad\qquad\qquad\quad \land allInstances_packname_B \subseteq B$

$InvariantAssociation_AB \triangleq \land AsForBoverD \in [B \rightarrow Seq(A)]$
$\qquad\qquad\qquad\qquad\quad \land BsForAoverD \in [A \rightarrow \text{SUBSET } B]$

$InvariantsAssociations \triangleq InvariantAssociation_AB$

Notice that the formulation of bidirectionality depends on the tags $\{ordered\}$ and $\{notUnique\}$ for association ends. Specifying this manually is error prone.

for each 'b', those As reachable over D from it must in turn have b among their reachable instances over D

$InvariantBidirectionality_AtoB \triangleq$
$\quad \lor AsForBoverD = \langle\rangle$
$\quad \lor \forall b \in \text{DOMAIN } AsForBoverD :$
$\qquad\quad \text{LET } AsReachable \triangleq AsForBoverD[b] \text{ IN}$ this is a TLA+ sequence
$\qquad\qquad \forall i \in \text{DOMAIN } AsReachable :$
$\qquad\qquad\quad \text{LET } anA \triangleq AsReachable[i] \text{ IN}$
$\qquad\qquad\qquad b \in BsForAoverD[anA]$

counterpart of the above, this time for each a

$InvariantBidirectionality_BtoA \triangleq$
$\quad \lor BsForAoverD = \langle\rangle$
$\quad \lor \forall a \in \text{DOMAIN } BsForAoverD :$
$\qquad\quad \text{LET } BsReachable \triangleq BsForAoverD[a] \text{ IN}$ this is a TLA+ set
$\qquad\qquad \forall aB \in BsReachable :$
$\qquad\qquad\quad ElemIsInSeq(a, AsForBoverD[aB])$

$InvariantsDirectionality \triangleq \land InvariantBidirectionality_AtoB$
$\qquad\qquad\qquad\qquad\quad \land InvariantBidirectionality_BtoA$

Invariants on multiplicities

$InvariantMultiplicity_BtoA \triangleq$
$\forall\, b \in \text{DOMAIN } AsForBoverD :$
$\text{LET } c \triangleq Len(AsForBoverD[b]) \text{IN}$
$\wedge\, c \geq amin$
$\wedge\, amin \geq 0$
$\wedge\, amax \geq c$
$\wedge\, amax \geq amin$

$InvariantsMultiplicity \triangleq InvariantMultiplicity_BtoA$

Invariants about allocated objects only referring to allocated objects. This is partial specification of heap consistency, limited to what has been modeled of an object store. Modeling garbage collection is still missing, but experience in that area was gained by model-checking the Schorr-Waite graph-marking algorithm.

$InvariantConsistentHeap_AsForBoverD \triangleq$
$\forall\, b \in \text{DOMAIN } AsForBoverD :$
$\text{LET } seq \triangleq AsForBoverD[b] \text{IN}$ this is a sequence
$SeqToSet(seq) \subseteq allInstances_packname_A$

$InvariantConsistentHeap_BsForAoverD \triangleq$
$\forall\, a \in \text{DOMAIN } BsForAoverD :$
$\text{LET } s \triangleq BsForAoverD[a] \text{IN}$ this is a set
$s \subseteq allInstances_packname_B$

$InvariantsConsistentHeap \triangleq \wedge\, InvariantConsistentHeap_AsForBoverD$
$\wedge\, InvariantConsistentHeap_BsForAoverD$

Conjunction of all invariants given by pieces

$Invariants \triangleq$
$\wedge\, InvariantAllInstances$
$\wedge\, InvariantsAssociations$
$\wedge\, InvariantsDirectionality$
$\wedge\, InvariantsMultiplicity$
$\wedge\, InvariantsConsistentHeap$

all *vars*, as needed for stuttering steps in the definition of *Spec* below
$vars \triangleq \langle allInstances_packname_A, allInstances_packname_B,$
$AsForBoverD, BsForAoverD$
$\rangle$

Operations

$\text{LOCAL } allocate_packname_A(anA) \triangleq$
$\wedge\, anA \in A$
$\wedge\, anA \notin allInstances_packname_A$
$\wedge\, allInstances_packname_A' = allInstances_packname_A \cup \{anA\}$

$\text{LOCAL } allocate_packname_B(aB) \triangleq$
$\quad \land aB \in B$
$\quad \land aB \notin allInstances_packname_B$
$\quad \land allInstances_packname_B' = allInstances_packname_B \cup \{aB\}$

$new_packname_A \triangleq$
$\quad \land A \neq allInstances_packname_A$ *i.e.* there are still non-allocated *IDs*
$\quad \land \text{LET } anA \triangleq \text{CHOOSE } x \in A : x \notin allInstances_packname_A$
$\quad\quad \text{IN} \quad \land allocate_packname_A(anA)$
$\quad\quad\quad\quad \land \text{UNCHANGED } \langle allInstances_packname_B, AsForBoverD, BsForAoverD \rangle$

$new_packname_B \triangleq$
$\quad \land B \neq allInstances_packname_B$ *i.e.* there are still non-allocated *IDs*
$\quad \land \text{LET } aB \triangleq \text{CHOOSE } x \in B : x \notin allInstances_packname_B$
$\quad\quad \text{IN} \quad \land allocate_packname_B(aB)$
$\quad\quad\quad\quad \land \text{UNCHANGED } \langle allInstances_packname_A, AsForBoverD, BsForAoverD \rangle$

Make a reachable over D for b, by adding it to a sequence. To maintain bidirectionality b should be reached from a.

$append_A_overD_B(a, b) \triangleq$
preconditions, allocated objects
$\quad \land a \in allInstances_packname_A$
$\quad \land b \in allInstances_packname_B$
preconditions on multiplicities
$\quad \land Len(AsForBoverD[b]) + 1 \leq amax$
$\quad \land Len(AsForBoverD[b]) + 1 \geq amin$
postconditions
$\quad \land AsForBoverD' = [AsForBoverD \text{ EXCEPT } ![b] = @ \circ \langle a \rangle]$
$\quad \land BsForAoverD' = [BsForAoverD \text{ EXCEPT } ![a] = @ \cup \{b\}]$
frame spec
$\quad \land \text{UNCHANGED } \langle allInstances_packname_A, allInstances_packname_B \rangle$

auxiliary function, set of *Bs* reachable over D, collected over all As in the sequence *seqOfAs*
$BsForAs(seqOfAs) \triangleq \text{UNION } \{BsForAoverD[seqOfAs[x]] : x \in \text{DOMAIN } seqOfAs\}$

auxiliary function, sequence resulting from remove at index index
$RemoveAt(seq, index) \triangleq SubSeq(seq, 1, index - 1) \circ SubSeq(seq, index + 1, Len(seq))$

$remove_anOccurrenceOfA_overD_B(a, b) \triangleq$

preconditions

$\land\ a \in allInstances_packname_A$
$\land\ b \in allInstances_packname_B$
$\land\ ElemIsInSeq(a, AsForBoverD[b])$

preconditions on multiplicities

$\land\ Len(AsForBoverD[b]) - 1 \leq amax$
$\land\ Len(AsForBoverD[b]) - 1 \geq amin$

postconditions

$\land\ \text{LET } toRemove \triangleq \text{CHOOSE } index \in 1 \mathrel{..} Len(AsForBoverD[b]) : \text{TRUE}$
$\quad \text{IN } \text{LET } newAs \triangleq RemoveAt(AsForBoverD[b], toRemove)$
$\quad\quad \text{IN } \land\ AsForBoverD' = [AsForBoverD \text{ EXCEPT } ![b] = newAs]$
$\quad\quad\quad \land\ BsForAoverD' = [BsForAoverD \text{ EXCEPT } ![a] = BsForAs(newAs)]$

frame spec

$\land\ \text{UNCHANGED } \langle allInstances_packname_A, allInstances_packname_B \rangle$

What we know about initial states

Syntax and semantic analysis by TLA+ tools is very practical, *e.g.* it reports whether the initial state has not been completely specified

$$Init \triangleq \land\ allInstances_packname_A = \{\}$$
$$\land\ allInstances_packname_B = \{\}$$
$$\land\ AsForBoverD = [b \in B \mapsto \langle\rangle]$$
$$\land\ BsForAoverD = [a \in A \mapsto \{\}]$$

$$Next \triangleq \lor\ new_packname_A$$
$$\lor\ new_packname_B$$
$$\lor\ \exists\, a \in allInstances_packname_A : \exists\, b \in allInstances_packname_B : append_A_overD_B(a, b)$$
$$\lor\ \exists\, a \in allInstances_packname_A : \exists\, b \in allInstances_packname_B : remove_anOccurrenceOfA_overD_B(a, b)$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

$\text{THEOREM } Spec \Rightarrow \Box(Invariants)$

D Otter-Eingabedateien

Die Datei Generalisierung-in.txt

```
% This file represents the following diagram
%      -----
%     |     |
%     |  A  |
%     |_____|
%       /_\
%        |
%        |
%        |
%      -----
%     |     |
%     |  B  |
%     |_____|

set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all x (A(x)). all y (B(y) -> A(y)).

end_of_list.

formula_list(sos).

exists z (-A(z)).       % -> Generalisierung-out

end_of_list.
```

Die Datei Generalisierung-in2.txt

```
% This file represents the following diagram
%       _____
%      |     |
%      |  A  |
%      |_____|
%        /_\
%         |
%         |
%         |
%       _____
%      |     |
%      |  B  |
%      |_____|

set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all x (A(x)). all y (B(y) -> A(y)).

end_of_list.

formula_list(sos).

exists z (B(z) & -A(z)).    % -> Generalisierung-out2

end_of_list.
```

Die Datei GenOverIncom-in.txt

```
% This file represents the following diagram
%             -----
%            |     |
%            |  A  |
%            |_____|
%              /_\
%               |
%               |
%               | {overlapping, incomplete}
%      ------------------
%     |                  |
%   -----              -----
%  |     |            |     |
%  |  B  |            |  C  |
%  |_____|            |_____|
%

set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all x A(x).             % an element can just be an element of A
all y (B(y) -> A(y)).   % inheritance
all z (C(z) -> A(z)).   % inheritance

end_of_list.

formula_list(sos).

exists p (B(p) & C(p)).

end_of_list.
```

Die Datei GenOverIncom-in2.txt

```
% This file represents the following diagram
%             _____
%            |     |
%            |  A  |
%            |_____|
%              /_\
%               |
%               |
%               | {overlapping, incomplete}
%       __________________
%      |                  |
%    _____              _____
%   |     |            |     |
%   |  B  |            |  C  |
%   |_____|            |_____|
%
set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all x A(x).             % an element can just be an element of A
all y (B(y) -> A(y)).   % inheritance
all z (C(z) -> A(z)).   % inheritance

end_of_list.

formula_list(sos).

exists q (A(q) & -B(q) & -C(q)).

end_of_list.
```

Die Datei GenDisCom-in.txt

```
% This file represents the following diagram
%              -----
%             |     |
%             |  A  |
%             |_____|
%               /_\
%                |
%                |
%                | {disjoint, complete}
%       ------------------
%      |                  |
%    -----              -----
%   |     |            |     |
%   |  B  |            |  C  |
%   |_____|            |_____|
%
set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all ( (A(x) -> B(x)) | (A(x) -> C(x)) ).    % completeness
all y (B(y) -> (A(y) & -C(y))).             % disjointness
all z (C(z) -> A(z)).                       % disjointness

end_of_list.

formula_list(sos).

exists t (A(t) & -B(t) & -C(t)).
      % test if there can exist an element which is
      %    just an element of A but not of B or C
end_of_list.
```

Die Datei GenDisCom-in2.txt

```
% This file represents the following diagram
%              -----
%             |     |
%             |  A  |
%             |_____|
%               /_\
%                |
%                |
%                | {disjoint, complete}
%       ------------------
%      |                  |
%    -----              -----
%   |     |            |     |
%   |  B  |            |  C  |
%   |_____|            |_____|
%
set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all y (B(y) -> A(y)).                   % inheritance
all z (C(z) -> A(z)).                   % inheritance
all p (B(p) -> -C(p)).                  % disjointness
all q (C(q) -> -B(q)).                  % disjointness
all r ((A(r) & B(r)) | (A(r) & C(r))).  % completeness

end_of_list.

formula_list(sos).

exists t (A(t) & -B(t) & -C(t)).
       % test if there can exist en element which can be
       %   just in set A but not B or C
end_of_list.
```

Die Datei Italians-in.txt

```
% This file represents the following diagram
%              ___________
%             |           |
%             |  Italian  |<|------------------------
%             |___________|                          |
%                  /_\                               |
%                   |                                |
%                   |                                |
%                   | {disjoint, complete}           |
%       ____________________________                 |
%      |            |               |                |
%  _________   ___________   ______________   ______________
% |         | |           | |              | |              |
% |  Lazy   | |  Mafioso  | |  LatinLover  | |  ItalianProf |
% |_________| |___________| |______________| |______________|
%      |            |__________        _____________| |
%      |_______________________disjoint_______________|
%

set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all x (ItalianProf(x) -> Italian(x)).
all x (Lazy(x) -> -ItalianProf(x)).
          % ItalianProf and Lazy are disjoint
all x (Mafioso(x) -> -ItalianProf(x)).
          % ItalianProf and Mafioso are disjoint
all x (Italian(x) -> (Lazy(x) | Mafioso(x) | LatinLover(x))).
          % completeness of Lazy, Mafioso and LatinLover
all x (Lazy(x) -> (Italian(x) & -Mafioso(x) & -LatinLover(x))).
all x (Mafioso(x) -> (Italian(x) & -Lazy(x) & -LatinLover(x))).
all x (LatinLover(x) -> (Italian(x) & -Lazy(x) & -Mafioso(x))).
          % disjointness of Lazy, Mafioso and LatinLover
end_of_list.

formula_list(sos).

all x (-LatinLover(x) & ItalianProf(x)).    % test if concept LatinLover
                                            %    subsumes concept ItalianProf
end_of_list.
```

Die Datei Italians-in2.txt

```
% This file represents the following diagram
%              -----------
%             |           |
%             |  Italian  |<|------------------------
%             |___________|                          |
%                  /_\                               |
%                   |                                |
%                   |                                |
%                   | {disjoint, complete}           |
%       ----------------------------                 |
%      |            |              |                 |
%  ---------   -----------   --------------   --------------
% |         | |           | |              | |              |
% |  Lazy   | |  Mafioso  | |  LatinLover  | |  ItalianProf |
% |_________| |___________| |______________| |______________|
%      |            |__________        _____________| |
%      |_______________________disjoint_______________|
%

set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all x (ItalianProf(x) -> Italian(x)).
all x (Lazy(x) -> -ItalianProf(x)).
          % ItalianProf and Lazy are disjoint
all x (Mafioso(x) -> -ItalianProf(x)).
          % ItalianProf and Mafioso are disjoint
all x (Italian(x) -> (Lazy(x) | Mafioso(x) | LatinLover(x))).
          % completeness of Lazy, Mafioso and LatinLover
all x (Lazy(x) -> (Italian(x) & -Mafioso(x) & -LatinLover(x))).
all x (Mafioso(x) -> (Italian(x) & -Lazy(x) & -LatinLover(x))).
all x (LatinLover(x) -> (Italian(x) & -Lazy(x) & -Mafioso(x))).
          % disjointness of Lazy, Mafioso and LatinLover
end_of_list.

formula_list(sos).

all x (-ItalianProf(x) & LatinLover(x)).     % test if concept ItalianProf
                                             %   subsumes concept LatinLover

end_of_list.
```

Die Datei Persons-in.txt

```
% This file represents the following diagram
%                                    __________
%                                   |          |
%                                   |  Person  |
%                                   |__________|
%                                       /_\
%                                        |
%                                        |
%                                        | {disjoint}
%                     ---------------------------------------
%                    |                                       |
%               -----------                             -----------
%              |           |                           |           |
%              |  Italian  |                           |  English  |
%              |___________|                           |___________|
%                   /_\                                     /_\
%                    |                                       |
%                    |                                       |
%                    | {disjoint, complete}                  |
%         -----------------------                   -----------------
%        |                       |                 |                 |
%   --------              --------------     -------------     ------------
%  |        |            |              |   |             |   |            |
%  |  Lazy  |            |  LatinLover  |   |  Gentleman  |   |  Hooligan  |
%  |________|            |______________|   |_____________|   |____________|
%                                |                 /_\
%                                |                  |
%                                |__________________|
%

set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all x (Italian(x) -> Person(x) & -English(x)).
           % disjointness and inherited class
all x (English(x) -> Person(x)).
           % English is subclass of Person

all X (Lazy(x) -> (Italian(x) & -LatinLover(x))).
           % disjointness of Lazy and LatinLover
all x (LatinLover(x) -> Italian(x)).
           % LatinLover is subclass of Italian
all x (Italian(x) -> (Lazy(x) | LatinLover(x))).
           % completeness of Lazy and LatinLover

all x (Gentleman(x) -> English(x)).
           % Gentleman is subclass of English
all x (Hooligan(x) -> English(x)).
           % Hooligan is subclass of English
```

```
all x (LatinLover(x) -> Gentleman(x)).
          % LatinLover is subclass of Gentleman
end_of_list.

formula_list(sos).

exists x (LatinLover(x)).
          % test if LatinLover can exist -> Persons-out.txt
          %     => LatinLovers cannot exist
end_of_list.
```

Die Datei Persons-in2.txt

```
% This file represents the following diagram
%                                  ----------
%                                 |          |
%                                 |  Person  |
%                                 |__________|
%                                      /_\
%                                       |
%                                       |
%                                       | {disjoint}
%                   ---------------------------------------
%                  |                                       |
%             -----------                             -----------
%            |           |                           |           |
%            |  Italian  |                           |  English  |
%            |___________|                           |___________|
%                 /_\                                     /_\
%                  |                                       |
%                  |                                       |
%                  | {disjoint, complete}                  |
%        -----------------------                   -----------------
%       |                       |                 |                 |
%   --------             --------------     -------------     ------------
%  |        |           |              |   |             |   |            |
%  |  Lazy  |           |  LatinLover  |   |  Gentleman  |   |  Hooligan  |
%  |________|           |______________|   |_____________|   |____________|
%                               |                /_\
%                               |                 |
%                               |_________________|
%

set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all x (Italian(x) -> Person(x) & -English(x)).
           % disjointness and inherited class
all x (English(x) -> Person(x)).
           % English is subclass of Person

all X (Lazy(x) -> (Italian(x) & -LatinLover(x))).
           % disjointness of Lazy and LatinLover
all x (LatinLover(x) -> Italian(x)).
           % LatinLover is subclass of Italian
all x (Italian(x) -> (Lazy(x) | LatinLover(x))).
           % completeness of Lazy and LatinLover

all x (Gentleman(x) -> English(x)).
           % Gentleman is subclass of English
all x (Hooligan(x) -> English(x)).
           % Hooligan is subclass of English
```

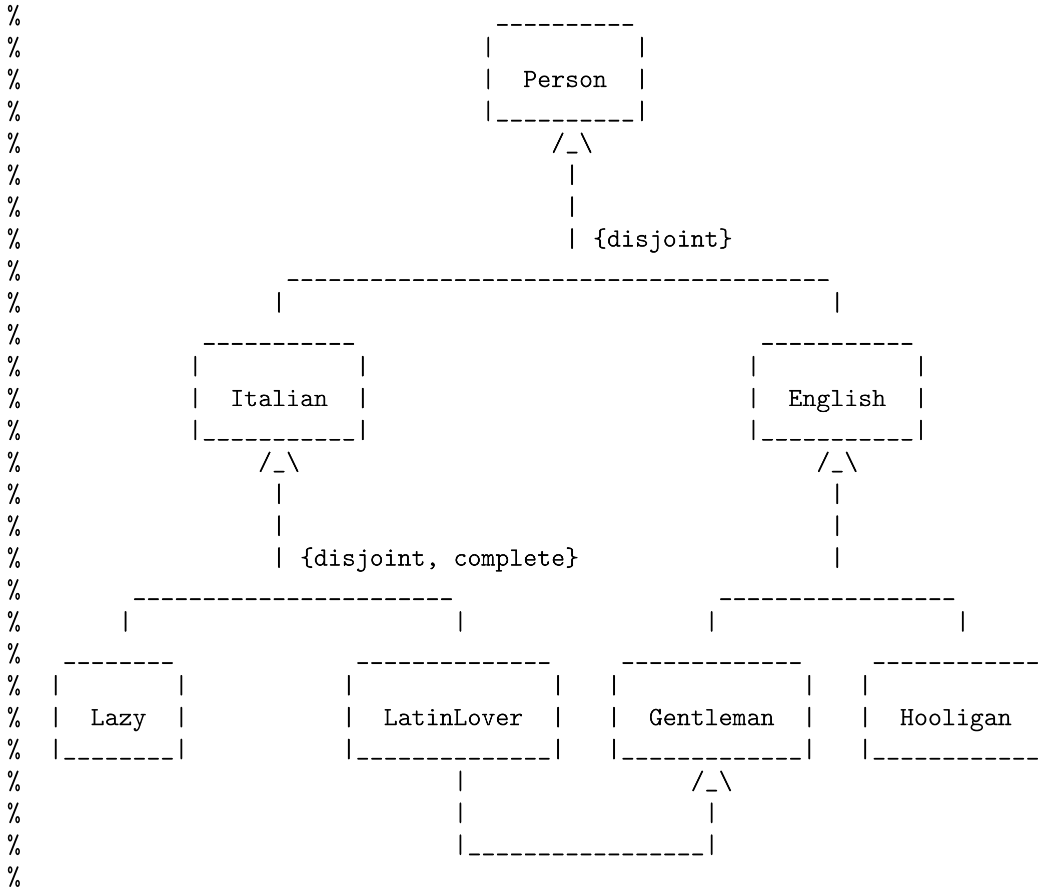
```
all x (LatinLover(x) -> Gentleman(x)).
           % LatinLover is subclass of Gentleman
end_of_list.

formula_list(sos).

all x (-Italian(x) & Lazy(x)).
          % test if concept Italian subsumes concept Lazy -> Persons-out2.txt

 end_of_list.
```

Die Datei Persons-in3.txt



```
% This file represents the following diagram
%                                  ----------
%                                 |          |
%                                 |  Person  |
%                                 |__________|
%                                      /_\
%                                       |
%                                       |
%                                       | {disjoint}
%                   ---------------------------------------
%                  |                                       |
%             -----------                             -----------
%            |           |                           |           |
%            |  Italian  |                           |  English  |
%            |___________|                           |___________|
%                 /_\                                     /_\
%                  |                                       |
%                  |                                       |
%                  | {disjoint, complete}                  |
%        -----------------------                   -----------------
%       |                       |                 |                 |
%   --------             --------------     -------------     ------------
%  |        |           |              |   |             |   |            |
%  |  Lazy  |           |  LatinLover  |   |  Gentleman  |   |  Hooligan  |
%  |________|           |______________|   |_____________|   |____________|
%                               |                /_\
%                               |                 |
%                               |_________________|
%

set(binary_res).    % Befehlsart: binaere Reolution

formula_list(usable).

all x (Italian(x) -> Person(x) & -English(x)).
           % disjointness and inherited class
all x (English(x) -> Person(x)).
           % English is subclass of Person

all X (Lazy(x) -> (Italian(x) & -LatinLover(x))).
           % disjointness of Lazy and LatinLover
all x (LatinLover(x) -> Italian(x)).
           % LatinLover is subclass of Italian
all x (Italian(x) -> (Lazy(x) | LatinLover(x))).
           % completeness of Lazy and LatinLover

all x (Gentleman(x) -> English(x)).
           % Gentleman is subclass of English
all x (Hooligan(x) -> English(x)).
           % Hooligan is subclass of English
```

```
all x (LatinLover(x) -> Gentleman(x)).
           % LatinLover is subclass of Gentleman
end_of_list.

formula_list(sos).

all x (-Lazy(x) & Italian(x)).
          % test if concept Lazy subsumes concept Italian -> Persons-out3.txt
          %  together with Persons-out2 => Italian and Lazy are identical

end_of_list.
```

E Verzeichnisstruktur der CD

Sämtliche im Rahmen dieser Bachelorarbeit entstandenen Dateien sind auf der zugehörigen CD zu finden. Die Eingabedateien für den Model-Checker TLC in der Sprache TLA$^+$ sind im Ordner `TLA\Eingabedateien` zu finden. Die Eingabedateien für den Beweiser OTTER befinden sich im Ordner `Otter\Eingabedateien`. Die dazugehörigen Ausgabedateien sind in den Ordnern `TLA\Ausgabedateien` und `Otter\Ausgabedateien` enthalten. Die Installationsdateien der benutzen Programme befinden sich jeweils als *.zip*-Datei im Ordner `TLA\Installationsdateien` bzw. `Otter\Installationsdateien`. Eine digitale Version dieses Berichtes ist im Ordner `Bericht` abgespeichert.

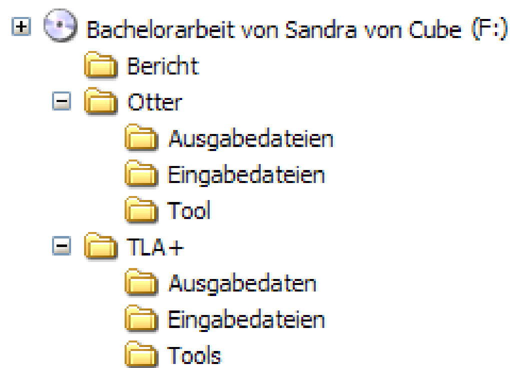


Abbildung 12: Verzeichnisstruktur der CD