Dr. Sebastian Küpper

Moderne Methoden der Software-Entwicklung

Modul 65010

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Moderne Methoden der Software-Entwicklung

Sebastian Küpper

8. Dezember 2025

Vorbemerkungen zum Modul Moderne Techniken der Software-Entwicklung und dem vorliegenden Kurstext

Das Modul „Moderne Techniken der Software-Entwicklung“ baut auf dem alten Modul „Moderne Programmiertechniken und -methoden“ von Friedrich Steimann auf und befindet sich derzeit in einer Umbauphase, in der schrittweise die Inhalte überarbeitet und teilweise ausgetauscht werden. Hierbei entsteht ein neues Skript, da das alte Skript nur sehr eingeschränkt wartbar ist. Im Moodle befinden sich das vollständige alte und das – dem aktuellen Stand nach – vollständige neue Skript zum Download. In der Druckfassung habe ich für einen einfacheren Überblick über die prüfungsrelevanten Inhalte eine Zusammenstellung dieser beiden Kurse erstellt, die Ihnen hiermit vorliegt. Bitte beachten Sie, dass der neue und der alte Kurstext keine gemeinsame Nummerierung haben und stellenweise möglicherweise Referenzen auf Inhalte zu finden sind, die nicht länger prüfungsrelevant und Teil dieses Dokuments sind. Die Inhalte, die Sie in diesem Dokument finden, sind die folgenden sieben Lektionen:

- Neuer Kurstext, Lektion 1: Design by Contract
- Neuer Kurstext, Lektion 2: Model Checking und Modellierung
- Neuer Kurstext, Lektion 3: Computation Tree Logic und ihre Anwendung zum Model Checking
- Neuer Kurstext, Lektion 4: Videospielentwicklung mit einer Spieleengine (Unity)
- Neuer Kurstext, Lektion 5: Praktische Aspekte der Software-Entwicklung in Unity
- Alter Kurstext, Kurseinheit 5: Refactoring
- Alter Kurstext, Kurseinheit 6: Metaprogrammierung

Die Einsendearbeiten orientieren sich an dieser Reihenfolge der Lektionen, so dass in den ersten beiden Wochen Heft 1 (passend zu Lektion 1) zu bearbeiten ist, anschließend Heft 2 (passend zu Lektion 2) usw.

Inhaltsverzeichnis

Lektion 1: Design by Contract **4**
1.1 Wozu Design by Contract . . . 4
1.2 Verhaltensspezifikation durch Zusicherungen: Vor- und Nachbedingungen, Invarianten . . . 5
1.3 Ein paar einfache Beispiele . . . 8
1.4 Design by Contract in der Analysephase . . . 11
1.5 Design by Contract in der Programmierung . . . 11
1.5.1 Zeitpunkt der Überprüfung von Zusicherungen . . . 12
1.5.2 Beeinflussung der Programmierung . . . 14
1.6 Schlussregeln für Hoare-Tripel . . . 15
1.7 Zusicherungen und Vererbung . . . 29
1.8 Spezifikationssprachen für das Design by Contract . . . 32
1.8.1 Eiffel . . . 33
1.8.2 Assertions in C# . . . 36
1.8.3 Java Modelling Language . . . 37
1.8.4 Code contracts in .NET . . . 40
1.8.5 Grenzen der Ausdrucksstärke . . . 41
1.9 Design by Contract-Tools als strukturiertes Testen . . . 41
1.10 Vor- und Nachteile des Design by Contract . . . 42
1.11 Zusammenfassung und Ausblick . . . 43
1.12 Zusammenfassung der wesentlichen Lernziele der Kurseinheit . . . 44

Lektion 2: Model Checking und Modellierung **45**
2.1 Einleitung und Problemstellung . . . 45
2.1.1 Hinweise zur Literatur . . . 45
2.1.2 Verifikation und Testen . . . 45
2.1.3 Transitionssysteme: Definition und Beispiele . . . 46
2.2 Verhaltensäquivalenzen für Transitionssysteme . . . 50
2.3 Bisimulation . . . 58
2.4 Das Bisimulationsspiel . . . 69
2.5 Zusammenfassung der wesentlichen Lernziele der Kurseinheit . . . 75

Lektion 3: Computation Tree Logic und ihre Anwendung zum Model Checking **76**
3.1 Das Model Checking-Problem . . . 77

3.2 Die Hennessy-Milner Logik und ihr Zusammenhang zur Bisimulation 78
3.3 Die Computation Tree Logic (CTL) und ihre Erweiterung 83
3.4 Der CTL Model Checking Algorithmus 90
3.5 Anwendung der CTL zum Model Checking 102
3.6 Zusammenfassung der wesentlichen Lernziele der Kurseinheit 102

Lektion 4: Videospielentwicklung mit einer Spieleengine (Unity) 104
4.1 Orientierung und Rotation in Unity 113
4.2 Algorithmen zur Kollisionserkennung 124
4.2.1 Kollision zwischen Achsenparallelen Quadern 126
4.2.2 Kollision zwischen zwei Kugeln 129
4.2.3 Der Trennungssatz von Eidelheit 129
4.2.4 Kollision zwischen Kugel und Kapsel 131
4.2.5 Kollisionserkennung zwischen zwei Kapseln 133
4.2.6 Kollisionen mit Nicht-Achsenparallelen Quadern (OBBs) 139
4.3 Zusammenfassung der wesentlichen Lernziele der Kurseinheit 148

Lektion 5: Praktische Aspekte der Software-Entwicklung in Unity 150
5.1 Kollisionserkennung in Unity und Hoppz' Sprungmechanik 150
5.2 Komplexe Interaktionen mit dem Kollisionssystem 160
5.3 Trigger und das Observer Pattern 167
5.3.1 Komponentenbasierte Entwicklung 167
5.3.2 Observer Pattern und Sammelgegenstand auflesen 169
5.4 Weitere Entwurfsmuster, die in Unity Verwendung finden 173
5.4.1 Facade Pattern 174
5.4.2 Template Method Pattern 175
5.5 Ausgewählte Unity-Funktionen 176
5.5.1 Prefabs 176
5.5.2 Szenenwechsel (Level) 177
5.5.3 UI und Canvas 178
5.6 Zusammenfassung der wesentlichen Lernziele der Kurseinheit 180

Lektion 2: Model Checking und Modellierung

2.1 Einleitung und Problemstellung

2.1.1 Hinweise zur Literatur

Die Inhalte dieses Kursmaterials basieren auf verschiedenen Quellen, einzelne Kapitel folgen allerdings jeweils einer Hauptquelle. Kapitel 2 und 4 folgen in weiten Teilen den Darstellungen in den Kursmaterialien „Modellierung nebenläufiger Systeme“ (?, ?), Kapitel 5 den Kursmaterialien „Modellierung, Analyse, Verifikation“ (?, ?), Zusammenfassungen von Ergebnissen der theoretischen Informatik (z.B. die WHILE-Sprache, Unentscheidbarkeit) dem Buch Einführung in die Theoretische Informatik (?, ?).

Besonderer Dank gilt Barbara König für die Genehmigung der stellenweise engen Orientierung an ihren Kursmaterialien.

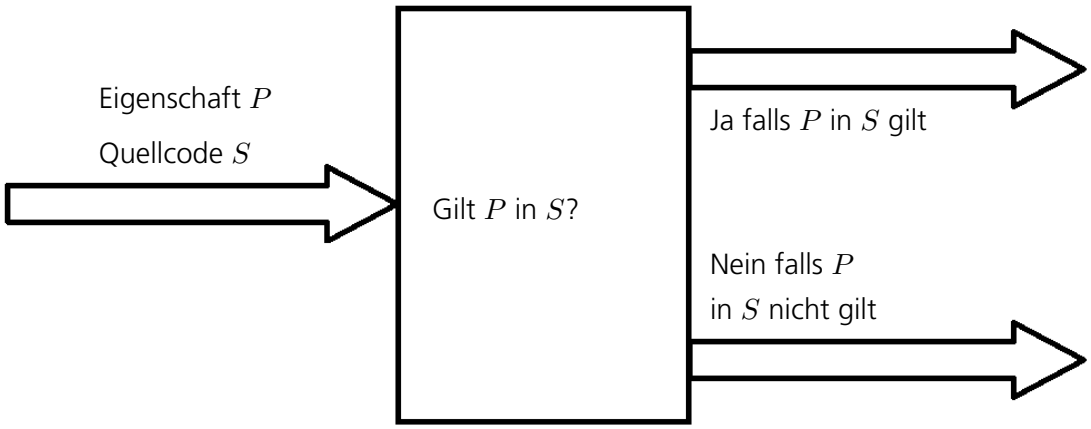
2.1.2 Verifikation und Testen

Eine zentrale Fragestellung in der Entwicklung von Software-Systemen ist im Regelfall die Korrektheit des Systems. Bei sicherheitskritischen Systemen, beispielsweise der Steuersoftware einer Rakete oder eines Atomkraftwerks ist es unbedingt notwendig, dass die Software genau das gewünschte Verhalten zeigt. Anderenfalls können Menschenleben gefährdet oder große wirtschaftliche Schäden hervorgerufen werden. Aber selbst bei Software, die im Fall einer Fehlfunktion nicht unmittelbar zu fatalen Konsequenzen führt, ist die Frage der Korrektheit spätestens bei der Vergütung der Entwicklungstätigkeit nicht unerheblich.

Um die Korrektheit eines Software-Systems zu überprüfen, gibt es zwei grundsätzliche Ansätze: Software kann entweder verifiziert oder getestet werden. Ziel der Verifikation ist es, einen absoluten Beweis für die Korrektheit zu erbringen, wohingegen beim Testen vor allem das Identifizieren von Fehlern im Vordergrund steht. In diesem Kurs soll das Testen als Technik zur Überprüfung der Korrektheit eines Software-Systems allerdings nicht in den Fokus gerückt werden. Testverfahren werden ausführlich in den Veranstaltungen des Lehrgebiets Softwaretechnik diskutiert. Stattdessen konzentrieren wir uns im vorliegenden Kurs ausschließlich auf Verifikationstechniken.

Idealerweise stellt man sich als Software-Entwickler Verifikation von Software wahrscheinlich ungefähr wie folgt vor: Gegeben sei ein Software-System, im einfachsten Fall unmittelbar in Form des Quelltextes, sowie eine formale Beschreibung der gewünschten Eigenschaft des Systems, beispielsweise in Form von Vor- und Nachbedingungen oder in Form eines logischen Ausdrucks. Der Entwickler gibt beide Eingaben in eine bereits vorweg

implementierte Blackbox, drückt einen Knopf mit der Aufschrift „verifizieren“, wartet eine Weile und erhält am Ende entweder die Ausgabe „ja“ oder „nein“ – je nachdem ob die Eigenschaft erfüllt ist oder nicht. Exemplarisch sei diese Idealvorstellung in der nachfolgenden Abbildung dargestellt.



Dass diese Idealvorstellung in der Praxis leider nicht uneingeschränkt erreicht werden kann, folgt aus der Unentscheidbarkeit bereits einfacher Eigenschaften wie Terminierung. In den Kursen der Theoretischen Informatik werden zahlreiche Probleme vorgestellt, die ebenfalls unentscheidbar sind.

2.1.3 Transitionssysteme: Definition und Beispiele

In der Theoretischen Informatik haben Sie bereits nichtdeterministische endliche Automaten kennengelernt, die zur Beschreibung von regulären Sprachen dienen. Ganz ähnlich zu nichtdeterministischen Automaten sind Transitionssysteme definiert:

Definition 2.1.1 : Transitionssystem

Ein Transitionssystem ist ein Dreitupel $(Z, A, \rightarrow)$, wobei

- Z eine Menge von Zuständen ist
- A eine Menge von Aktionen ist und
- $\rightarrow \subseteq Z \times A \times Z$ die Menge der Transitionen ist.

Wir schreiben $z \xrightarrow{a} z'$, falls $(z, a, z') \in \rightarrow$ gilt und schreiben $z_1 \xrightarrow{a_1 a_2 \ldots a_n} z_{n+1}$ falls es Zustände $z_2, z_3, ..., z_n$ gibt, so dass $z_1 \xrightarrow{a_1} z_2 \xrightarrow{a_2} z_3 ... \xrightarrow{a_n} z_{n+1}$ gilt. Es gilt mit dieser Konvention $z \xrightarrow{\epsilon} z$ für alle Zustände $z \in Z$. Wenn uns der erreichte Zustand nicht

interessiert, schreiben wir zudem für alle $w \in A^*$ kurz $z \xrightarrow{w}$ falls es einen Zustand z' gibt, so dass $z \xrightarrow{w} z'$ und anderenfalls, wenn es einen solchen Zustand z' nicht gibt, $z \not\xrightarrow{w}$.

Transitionssysteme unterscheiden sich von nichtdeterministischen Automaten in zweierlei Hinsicht: Es gibt keine Start- oder Endzustände und die Menge der Zustände und Alphabetsymbole ist nicht zwingend endlich. Dementsprechend werden wir sehen, dass der Begriff der Sprache eines Transitionssystems sich ein wenig von der eines nichtdeterministischen Automaten unterscheidet. Allerdings halten wir an dieser Stelle bereits fest, dass endliche Transitionssysteme, d.h. solche, bei denen die Zustandsmenge und das Alphabet endlich sind, als Modellierungswerkzeug die Ausdrucksmächtigkeit im Vergleich zu turingmächtigen Systemen einschränken. Für endliche Transitionssysteme umgehen wir also die Problematik der Unentscheidbarkeit bereits auf Modellierungsebene, können im Gegenzug aber nicht alle Systeme mit (endlichen) Transitionssystemen modellieren.

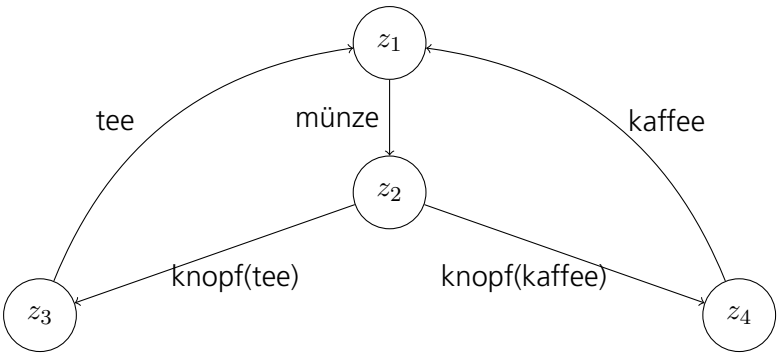
Ein klassisches Beispiel, mit dem Transitionssysteme illustriert werden, ist eine Maschine für Heißgetränke, die dem Kunden die Wahl aus den Getränken Tee und Kaffee lässt:

Beispiel 2.1.2

Wir können einen Automaten für Heißgetränke wie folgt als Transitionssystem modellieren:

$$(\{z_1, z_2, z_3, z_4\}, \{kaffee, münze, tee, knopf(kaffee), knopf(tee)\}, \rightarrow),$$

wobei wir die Überführungsfunktion $\rightarrow$ wie folgt grafisch repräsentieren.



Diese grafische Notation für Transitionssysteme werden wir in Beispielen häufig verwenden, da sie, jedenfalls bei hinreichend kleinen Transitionssystemen, wesentlich suggestiver ist, als eine Darstellung als Formel.

Ähnlich wie bei Automaten gibt es auch im Falle von Transitionssystemen eine deterministische Variante.

Definition 2.1.3 : Deterministisches Transitionssystem

Ein Transitionssystem $(Z, A, \rightarrow)$ ist deterministisch, falls für alle $z \in Z$, $a \in A$ gilt, dass $z \xrightarrow{a} z_1$ und $z \xrightarrow{a} z_2$ impliziert, dass $z_1 = z_2$.

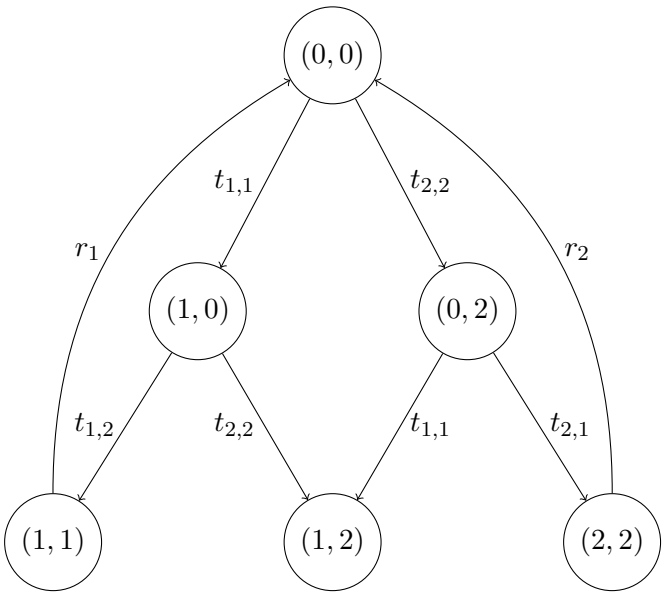
Dementsprechend ist unser zuvor besprochenes Beispiel eines Kaffee- und Tee-Automaten deterministisch. Beachten Sie, dass, anders als bei deterministischen Automaten, nicht gefordert wird, dass für jedes Paar von Zustand und Aktion ein Nachfolgezustand existiert.

Wir wollen nun einen ersten Blick auf ein Beispiel werfen, in dem ein Transitionssystem zur Modellierung eines Protokolls und zur Identifikation eines Fehlers in diesem Protokoll dient. Das Problem der Verklemmung (engl. Deadlock) ist ein typischer Softwarefehler in einem verteilten System. Wenn mehrere Prozesse unabhängig voneinander auf eine geteilte Ressource zugreifen können sollen, kann dieser Fehler auftreten, wie anhand des Bildes der dinierenden Philosophen (engl. dining philosophers) illustriert wird. Was viele Menschen über Philosophen nämlich nicht wissen[1], ist der Umstand, dass Philosophen zum Essen immerzu zwei Gabeln benötigen – das motorische Geschick wird durch das angestrengte Denken offenbar in Mitleidenschaft gezogen. Nun kommt es aber gelegentlich vor, dass Philosophen sich zu einem gemeinsamen Abendessen verabreden, um über gewichtige Probleme zu debattieren. Zu diesem Zwecke bietet sich natürlich ein runder Tisch in einem Restaurant an, der den maximalen Abstand zwischen Paaren von Philosophen minimiert. Leider wissen aber insbesondere viele Gastronomen nichts von den Essgewohnheiten von Philosophen, so dass es vorkommen kann, dass für jeden Philosophen nur eine einzige Gabel vorgesehen ist, die sich dann auf der linken Seite ihres Tellers befindet. Zum Glück für die Philosophen bedeutet das an einem runden Tisch allerdings, dass jeder Philosoph zwei Gabeln neben seinem Teller liegen hat. In der beschriebenen Situation kann ein Philosoph also durchaus essen, indem er zunächst nach der Gabel links neben seinem Teller greift, anschließend die Gabel rechts von seinem Teller aufnimmt, mit seinen beiden Esswerkzeugen eine Weile isst und anschließend seine Gabeln wieder auf dem Tisch platziert.

Im Laufe des Abends können die Philosophen sich nun in zwei Zuständen befinden, sie können hungrig sein oder in Diskussionslaune sein. Wenn ein Philosoph in Diskussionslaune ist, diskutiert er und verwendet hierzu keine Gabeln. Wenn er aber hungrig ist, versucht er wie oben beschrieben zwei Gabeln aufzunehmen und zu essen. Gelingt dem Philosophen nicht, nach der Aufnahme der linken Gabel auf die rechte Gabel vom Tisch aufzunehmen, weil sie gerade in Verwendung ist, wartet der Philosoph geduldig, bis die zweite Gabel wieder verfügbar ist und nimmt sie dann auf, um seinen Magen zu füllen. Die Philosophen können unabhängig voneinander hungrig werden und Gabeln aufnehmen. Wir wollen nun diese Situation als Transitionssystem modellieren, um zu untersuchen, ob das beschriebene System eine Verklemmung besitzt, ob also passieren kann, dass Philosophen wechselseitig aufeinander warten. Angenommen wir betrachten die Situation im einfachsten Fall mit nur zwei Philosophen, dann können wir die Situation wie folgt modellieren:

[1] Dem Vernehmen nach hängt das damit zusammen, dass diese Aussage in der Realität gar nicht der Wahrheit entspräche.

Zustände sind Paare $(g_1, g_2) \in \{0,1,2\} \times \{0,1,2\}$ mit der folgenden Bedeutung: g_1 zeigt an, ob die Gabel des ersten Philosophen gerade auf dem Tisch liegt (dann ist $g_1 = 0$), vom ersten Philosophen gehalten wird (dann ist $g_1 = 1$) oder vom zweiten Philosophen gehalten wird (dann ist $g_1 = 2$). Analog zeigt g_2 den Status der Gabel des zweiten Philosophen an. Es gibt also insgesamt $3^2 = 9$ Zustände. Weiterhin gibt es die Aktionen $t_{i,j}$ und r_i mit $i, j \in \{1,2\}$. Die Aktion $t_{i,j}$ bedeutet, dass der i-te Philosoph die j-te Gabel aufnimmt (t für engl. take) und r_i bedeutet, dass der i-te Philosoph beide Gabeln zurücklegt (r für engl. return). Betrachten wir nun den zugehörigen Graphen, der die Transitionsfunktion anzeigt, so sehen wir, dass von der Startsituation $(0,0)$ aus die Situation $(1,2)$ erreichbar ist, von der aus keine weitere Aktion durchgeführt werden kann, es gibt also einen Deadlock im Transitionssystem.



Diese Situation entspricht dem ungünstigen Szenario, dass der erste Philosoph hungrig geworden ist und seine Gabel aufnimmt. Noch bevor er aber nach rechts zur zweiten Gabel greifen kann, wird auch der zweite Philosoph hungrig und greift nach seiner Gabel. Nun warten beiden Philosphen wechselseitig aufeinander und leiden ohne Eingreifen von außen beliebig lange Hunger.

Analog lässt sich das Szenario auch für eine größere Zahl an Philosophen modellieren, in jedem Fall können aber Verklemmungen auftreten. Im betrachteten Beispiel gibt es verschiedene Lösungsmöglichkeiten für das Verklemmungsproblem:

- Einer der Philosophen nimmt immer zuerst die rechte Gabel auf, mindestens ein anderer erst die linke.

- Allgemeiner: Man definiert eine Prioritätenfolge auf den Gabeln, die für alle Philosophen gilt und verlangt, dass die Gabeln immer nur in dieser Reihenfolge aufgenommen werden können.

- Ein Philosoph muss stets beide Gabeln auf einmal aufnehmen.
- Philosophen können nicht beliebig lange mit einer Gabel in der Hand warten, sondern müssen nach einer Weile die Gabel wieder ablegen.

Wie eingangs erwähnt, ist das Bild der dinierenden Philosophen ein abstraktes Modell für Schwierigkeiten, die in einem Netzwerk mit geteilten Ressourcen auftreten können und die Lösungsstrategien entsprechen tatsächlichen Lösungsstrategien für Verklemmungen in nebenläufigen Systemen. Die ersten beiden Punkte entsprechen der Definition einer Prioritätenfolge auf geteilten Ressourcen, der dritte Punkt der Einführung größerer atomarer Aktionen und der vierte Lösungsvorschlag entspricht einem Timeout. Unser Ziel soll es sein, derartige Probleme bereits am Modell zu erkennen und somit in der Implementation des Systems von vorneherein zu vermeiden.

2.2 Verhaltensäquivalenzen für Transitionssysteme

Zur Analyse von Transitionssystemen müssen wir uns zunächst darüber Gedanken machen, was das Verhalten eines Transitionssystems charakterisieren soll. Wir beginnen damit, verschiedene Ansätze, das Verhalten eines Zustandes zu definieren und zu vergleichen. Sie kennen aus der theoretischen Informatik bereits den Begriff der Sprachäquivalenz für nichtdeterministische Automaten, der sich nahezu unmittelbar auf Transitionssysteme übertragen lässt. Allerdings werden wir uns mit verschiedenen Varianten der Sprachäquivalenz beschäftigen, denn wenn wir uns einmal unsere Eingangsbeispiele zu Transitionssystemen in Erinnerung rufen, werden wir Transitionssysteme vor allem dazu verwenden, beliebig lang laufende Systeme zu modellieren. Daher kann es sinnvoll sein, nicht nur endliche Abläufe zu betrachten.

Definition 2.2.1 : Sprachäquivalenz

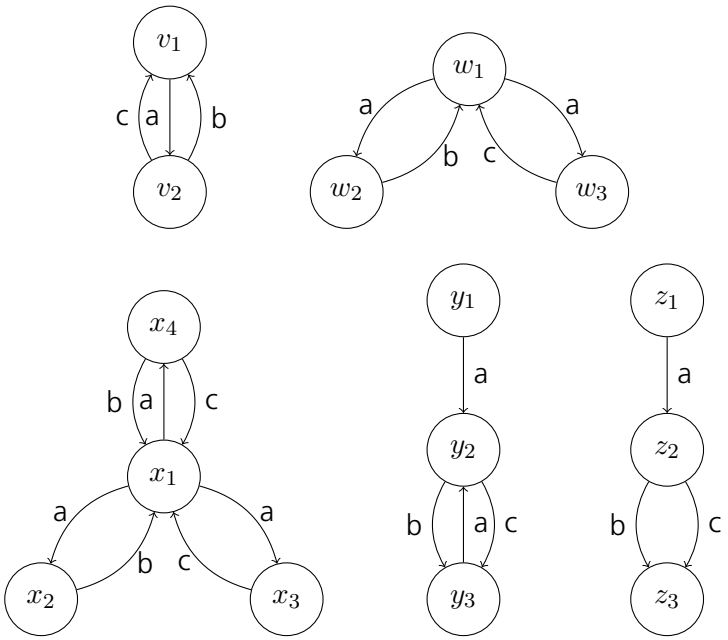
Es sei $(Z, A, \rightarrow)$ ein Transitionssystem und $z \in Z$ ein Zustand des Transitionssystems. Wir definieren nun drei verschiedene Sprachen des Zustandes z und verwenden die Schreibweisen A^* für alle endlichen Sequenzen über A, A^ω für alle unendlichen Sequenzen über A und A^∞ für alle (unendlichen oder endlichen) Sequenzen über A:

- $S(z) = \{w \in A^* \mid \exists z' \in Z : z \xrightarrow{w} z'\}$
- $S^\omega(z) = \{a_1a_2a_3... \in A^\omega \mid \exists z_1, z_2, z_3, ... : z \xrightarrow{a_1} z_1 \xrightarrow{a_2} z_2 \xrightarrow{a_3} z_3...\}$
- $S^\infty(z) = S(z) \cup S^\omega(z)$

Zwei Zustände $z, z' \in Z$ sind S-sprachäquivalent (S^ω-sprachäquivalent, S^∞-sprachäquivalent), wenn die jeweiligen S-Sprachen (S^ω-Sprachen, S^∞-Sprachen) gleich sind, in Zeichen: $S(z) = S(z')$ ($S^\omega(z) = S^\omega(z')$, $S^\infty(z) = S^\infty(z')$).

Selbsttestaufgabe 2.2.1

Für diese und zwei weitere Selbsttestaufgaben verwenden wir die folgenden fünf Transitionssysteme:



1. Sind die Zustände v_1 und w_1 sprachäquivalent? Geben Sie hierzu die jeweils generierte Sprache $S(v_1)$ und $S(w_1)$ an.
2. Sind die Zustände w_1 und x_1 sprachäquivalent? Geben Sie hierzu die jeweils generierte Sprache $S(w_1)$ und $S(x_1)$ an.
3. Sind die Zustände v_1 und z_1 sprachäquivalent? Geben Sie hierzu die jeweils generierte Sprache $S(v_1)$ und $S(z_1)$ an.
4. Sind die Zustände v_1 und y_1 sprachäquivalent? Geben Sie hierzu die jeweils generierte Sprache $S(v_1)$ und $S(y_1)$ an.

Musterlösung zu Selbsttestaufgabe 2.2.1

Wir geben zunächst die Sprachen der gefragten Zustände an:

$$\begin{aligned} &S(v_1) = S(w_1) = S(x_1) = S(y_1) \\ =&\{w \in \{a,b,c\}^* \mid \forall 1 \leq i \leq |w| : i \mod 2 = 1 \Leftrightarrow w[i] = a\} \end{aligned}$$

Alternativ kann man diese Sprache auch als regulären Ausdruck aufschreiben: $L((a(b|c))^*)$. Weniger formell ist es die Menge aller Wörter, die an ungeraden Stellen ein a und an geraden Stellen ein b oder ein c besitzen. Die Sprache $S(z_1) = \{\epsilon, a, ab, ac\}$ ist hingegen endlich. Damit kann man unmittelbar die Lösung der Aufgabenteile einsehen:

1. Ja.
2. Ja.
3. Nein.
4. Ja.

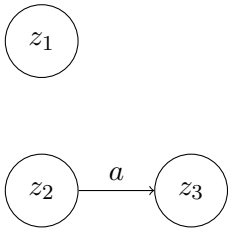
Es ist offensichtlich, dass zwei Zustände S^∞-sprachäqivalent sind, genau dann wenn sie sowohl S-sprachäquivalent, als auch S^ω-sprachäquivalent sind: Ist $S(x) = S(x')$ und $S^\omega(x) = S^\omega(x')$, so ist auch $S(x) \cup S^\omega(x) = S(x') \cup S^\omega(x')$. Umgekehrt beachten wir, dass $S(x)$ und $S^\omega(x)$ disjunkt sind für alle x, da $S(x)$ nur endliche Sequenzen enthält und S^ω nur unendliche Sequenzen enthält. Also gilt, wenn $S^\infty(x) = S^\infty(x')$ ist:

$$\begin{aligned} S(x) &= (S^\omega(x) \cup S(x)) \cap A^* = S^\infty(x) \cap A^* \\ &= S^\infty(x') \cap A^* = (S^\omega(x') \cup S(x')) \cap A^* = S(x') \end{aligned}$$

und analog

$$\begin{aligned} S^\omega(x) &= (S^\omega(x) \cup S(x)) \cap A^\omega = S^\infty(x) \cap A^\omega \\ &= S^\infty(x') \cap A^\omega = (S^\omega(x') \cup S(x')) \cap A^\omega = S^\omega(x'). \end{aligned}$$

Ein wenig schwieriger einzusehen ist, dass S-Sprachäquivalenz nicht S^ω-Sprachäquivalenz impliziert und umgekehrt, dass S^ω-Sprachäquivalenz nicht S-Sprachäquivalenz impliziert. Ein Beispiel zu finden für zwei Zustände, die S^ω-sprachäquivalent sind aber nicht S-sprachäquivalent sind, ist allerdings unproblematisch, wenn man sich vor Augen führt, dass alle Paare von Zuständen, die keine unendlichen Wörter akzeptieren, unmittelbar als S^ω-sprachäquivalent eingesehen werden können. Das einfachste Beispiel für diesen Umstand wären also die beiden folgenden Zustände z_1, z_2:



Von z_1 aus sind keine Aktionen möglich, aber von z_2 aus kann man eine a-Aktion durchführen, also gilt $S(z_1) = \{\epsilon\} \neq \{\epsilon, a\} = S(z_2)$. Allerdings kann weder von z_1 aus noch von z_2 aus eine unendlich lange Folge von Transitionen durchgeführt werden, also gilt trivial $S^\omega(z_1) = \emptyset = S^\omega(z_2)$.

Für die umgekehrte Richtung, zwei Zustände, die S-sprachäquivalent, aber nicht S^ω-sprachäquivalent sind, benötigen wir unendlich viele Zustände.

Satz 2.2.2

In einem Transitionssystem mit endlich vielen Zuständen gilt: Wenn zwei Zustände z und z' S-sprachäquivalent sind, dann sind sie auch S^ω-sprachäquivalent.

Beweis. Der Grund hierfür ist einfach ersichtlich, wenn man sich vor Augen führt, dass jedes endliche Präfix eines unendlichen Wortes ein endliches Wort ist. Damit zwei Zustände $z, z' \in Z$ existieren können, die nicht S^ω-sprachäquivalent sind, muss einer der beiden Zustände, o.B.d.A. z einen unendlichen Pfad besitzen, dessen Wort w vom anderen Zustand z' aus nicht eingelesen werden kann. Wenn die beiden Zustände zusätzlich S-sprachäquivalent sind, müssen aber alle endlichen Präfixe des Wortes w von z' aus akzeptiert werden. Wir konstruieren nun einen unendlichen w-Pfad $z_0, z_1, z_2 ...$ von z' aus.

Es gilt $z_0 = z$. Die (einelementige) Sequenz $S_0 = z_0$ hat die Eigenschaft, dass es für jedes endliche Präfix w' von w einen w'-Pfad gibt, der mit $S_0 = z_0$ beginnt.

Sei nun eine Sequenz S_i der Länge $i \in \mathbb{N}$ gegeben, die die Eigenschaft hat, dass jedes endliche Präfix w' von w entweder von einem Präfix von S_i akzeptiert wird, oder (wenn w' länger als i ist) S_i Präfix eines Pfades ist, der w' akzeptiert. Wir wollen nun einen Zustand z_{i+1} wählen, so dass die Sequenz $S_i z_{i+1}$ diese Eigenschaft behält.

Hierzu verwenden wir das unendliche Taubenschlag- oder Schubfachprinzip, das besagt: Wenn man eine unendliche Zahl an Elementen (hier: Präfixe des Wortes w) auf endlich viele Äquivalenzklassen verteilen möchte (hier: Zustände), dann muss es mindestens eine unendlich große Äquivalenzklasse geben.

Wir betrachten nun also die Menge aller Präfixe w' von w, die mehr als i Zeichen enthalten, sowie deren zugehörigen Sequenzen $S_{w'}$, so dass S_i ein Präfix von $S_{w'}$ ist und $S_{w'}$ das Wort w' akzeptiert. In jeder dieser Sequenzen gibt es einen Zustand, der durch das Einlesen des $i+1$-ten Zeichens von w' erreicht wird. Mindestens einer dieser Zustände, nennen wir ihn $\hat{z}$, wird nach obigem unendlichen Taubenschlagprinzip unendlich oft erreicht. Wir setzen $z_{i+1} = \hat{z}$.

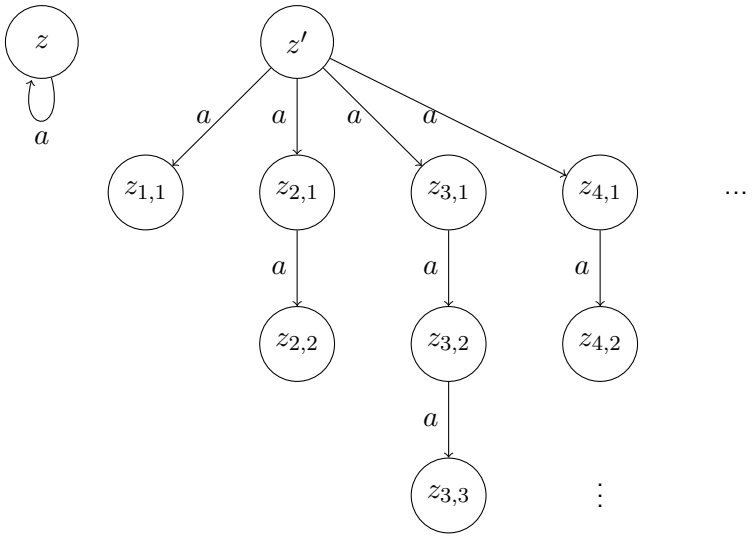
Wir müssen nun zeigen, dass die so erhaltene Sequenz $S_i z_{i+1}$ die in der Induktionsvoraussetzung gegebene Bedingung auch tatsächlich erfüllt. Sei hierzu ein beliebiges Präfix $\overline{w}$ von w gegeben. Wenn $\overline{w}$ maximal die Länge $i+1$ hat, ist nichts zu zeigen. Ist $\overline{w}$ hingegen länger als $i+1$, so ist zu zeigen, dass es eine Fortsetzung von $S_i z_{i+1}$ gibt, die $\overline{w}$ akzeptiert. Da $S_i z_{i+1}$ Präfix unendlich vieler Sequenzen $S_{w'}$ ist, die ein Präfix w' von w mit mehr als i Zeichen akzeptieren, gibt es ein Präfix $\tilde{w}$, das länger ist als $\overline{w}$, so dass $S_i z_{i+1}$ ein Präfix von $S_{\tilde{w}}$ ist und $S_{\tilde{w}}$ das Wort $\tilde{w}$ akzeptiert. Da $\overline{w}$ ein Präfix von $\tilde{w}$ ist, muss das Präfix der Länge $|w'| + 1 > i + 1$ von $S_{\tilde{w}}$ das Wort w' akzeptieren.

Wir haben also eine unendliche Sequenz von Zuständen konstruiert, die jedes Präfix von w akzeptiert, also akzeptiert die Sequenz insgesamt auch w. □

Dieser Beweis gibt uns auch bereits einen Anhaltspunkt, wie ein Beispiel aussehen kann, bei dem S^ω-Sprachäquivalenz nicht S-Sprachäquivalenz impliziert.

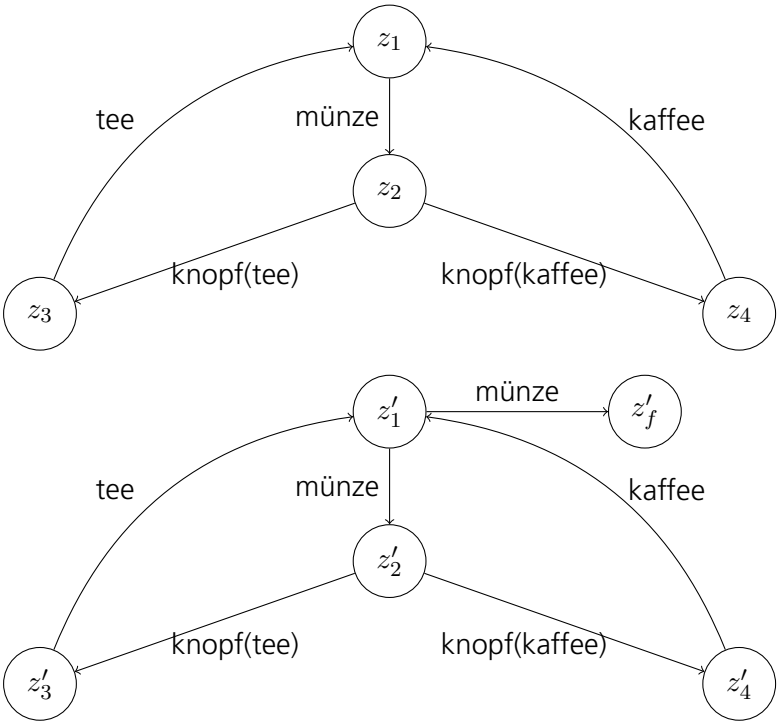
Beispiel 2.2.3

Wir benötigen einen Zustand z, der einen unendlichen Pfad für das unendliche Wort w besitzt und einen weiteren Zustand z', der auf Pfaden, die ab einem bestimmten Index disjunkt sind, für jede endliche Wortlänge n das Präfix der Länge n von w akzeptieren. Im einfachsten Fall wird das unendliche Wort unmittelbar durch eine Schleife gebildet und das Transitionssystem kann beispielsweise wie folgt aussehen:



Es gibt also für jedes $n \in \mathbb{N}$ eine eigene Sequenz von Zuständen $z_{n,1}z_{n,2}...z_{n,n}$, so dass $z'z_{n,1}z_{n,2}...z_{n,n}$ genau das Wort a^n akzeptiert. $S(z) = \{a^n \mid n \in \mathbb{N}_0\} = S(z')$, aber $S^\omega(z) = \{a^\omega\} \neq \emptyset = S^\omega(z')$.

Wir haben bislang Adaptionen des Begriffs Sprachäquivalenz in Anlehnung an die Automatentheorie kennen gelernt. Allerdings reicht es uns bei der Analyse von Software-Systemen in aller Regel nicht, nur zu überprüfen, welche Aktionen *möglich* sind, manchmal ist es, im Falle von nichtdeterministischen Systemen, gleichsam interessant, zu untersuchen, welche Aktionen *unmöglich* sind. Zu diesem Zweck betrachten wir zunächst einmal das folgende motivierende Beispiel, das unser einführendes Beispiel eines Heißgetränkeautomaten wieder aufgreift. Neben dem ursprünglichen Beispiel (oben) betrachten wir noch eine alternative Modellierung (unten), bei der die Möglichkeit modelliert wird, dass der Getränkeautomat in einen Fehlerzustand übergeht.



Die Sprachen der Zustände z_1 und z'_1 sind gleich: Beide Graphen sind isomorph mit Ausnahme des zusätzlichen Zustandes z'_f und des zugehörigen münze-Übergangs im rechten Automaten. Allerdings ermöglicht dieser keine zusätzlichen Transitionsfolgen, da vom Zustand z'_1 aus bereits ein münze-Übergang existiert. Allerdings verhalten sich beide Automaten aus Sicht eines Benutzers durchaus unterschiedlich. Während im ursprünglichen Transitionssystem sichergestellt ist, dass auf jeden Münzeinwurf irgendwann die Auswahl aus Kaffee und Tee erfolgen kann, kann das rechte Transitionssystem nach Münzeinwurf blockieren. Selbst ungeachtet des Umstandes, dass man auf diese Weise keine Heißgetränke mehr erhalten kann eine unangenehme Situation.

Nach dieser Überlegung wollen wir nun eine weitere Verhaltensäquivalenz für Transitionssysteme kennen lernen, die Verweigerungsäquivalenz (engl. failures equivalence).

Definition 2.2.4 : Verweigerungsäquivalenz

Es sei $(Z, A, \rightarrow)$ ein Transitionssystem und $z \in Z$ ein beliebiger Zustand, dann bezeichnen wir die folgende Menge als die Menge von Verweigerungspaaren:

$$V(z) = \{(w, A') \in A^* \times \mathcal{P}(A) \mid \exists z' \in Z : z \xrightarrow{w} z' \wedge \forall a \in A' : z' \not\xrightarrow{a}\}$$

Wir nennen zwei Zustände z, z' verweigerungsäquivalent oder V-äquivalent, falls $V(z) = V(z')$ gilt.

Die Verweigerungspaare eines Zustandes z sind also die Paare aller Wörter w und Aktionen $A' \subseteq A$, so dass man von z aus das Wort w einlesen kann und in einem Zustand z' landen

kann, so dass man anschließend keine Transition mit einem Label a ausführen kann. Die Menge $V(z)$ ist oft unendlich, weil A^* unendlich ist. Es ist wichtig zu beachten, dass im Falle eines nichtdeterministischen Transitionssystems die Menge $V(z)$ maximale Paare (w, A') und (w, A'') enthalten kann – maximal bedeutet hier, dass es kein $\overline{A}' \supseteq A'$ gibt mit $(w, \overline{A}') \in V(z)$ – die verschieden sind. Der Grund hierfür ist, dass mit dem gleichen Wort w verschiedene Zustände erreicht werden können, die jeweils potentiell verschiedene Eingaben verweigern.

Betrachten wir unser obiges Beispiel, so können wir beobachten, dass die Zustände z_1 und z_1' nicht verweigerungsäquivalent sind, denn es gilt $(\text{münze}, A) \in V(z_1')$ aber $(\text{münze}, A) \notin V(z_1)$

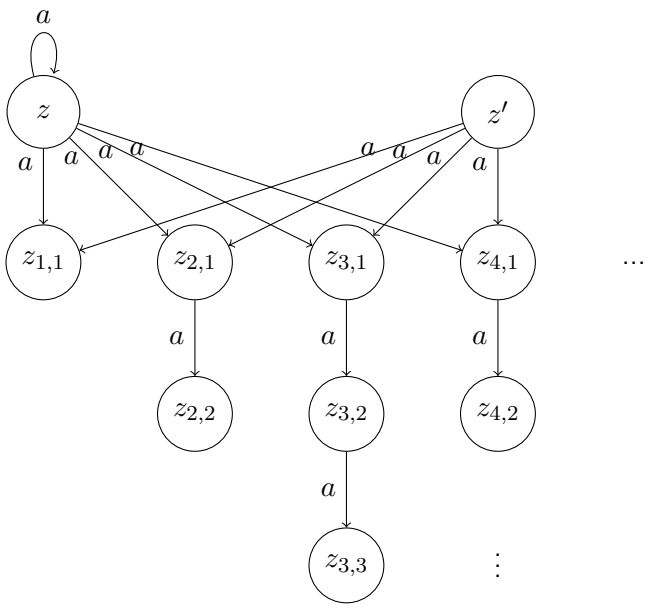
Wir haben damit auch bereits ein Beispiel kennen gelernt, das zeigt, dass S^∞-Äquivalenz (und damit auch S^ω- und S-Äquivalenz) nicht Verweigerungsäquivalenz implizieren. Umgekehrt gilt allerdings, dass V-Äquivalenz S-Äquivalenz impliziert.

Satz 2.2.5

Es sei $(Z, A, \to)$ ein Transitionssystem und z, z' V-äquivalent. Dann sind z, z' auch S-äquivalent.

Beweis. Wir zeigen dies per Widerspruchsbeweis und nehmen an $V(z) = V(z')$, aber $S(z) \neq S(z')$. O.B.d.A nehmen wir an es existiert ein Wort $w \in S(z) \setminus S(z')$, dann gilt $(w, \emptyset) \in V(z) \setminus V(z')$, da es einen Zustand gibt, der von z aus mit w erreicht wird. Ganz sicher ist es möglich, von dort aus gar keine Eingabe zu verweigern, es gilt also $(w, \emptyset) \in V(z)$. Allerdings gibt es keinen Zustand, der mit w von z' aus erreicht wird, also ist $V(z') \cap \{(w, A') \mid A' \subseteq A\} = \emptyset$ nach Definition von $V(z')$. Das ist aber ein Widerspruch zu der Annahme $V(z) = V(z')$. □

Verweigerungsäquivalenz impliziert allerdings nicht S^ω-Äquivalenz (und demnach auch nicht S^∞-Äquivalenz), wie man anhand einer leichten Modifikation eines vorherigen Beispiels erkennen kann:



Die Zustände z und z' sind verweigerungsäquivalent, da sie für jedes $n \geq 1$ nach Eingabe des Wortes a^n das Zeichen a verweigern können, aber nichts für das Wort ϵ verweigern können. Sie sind aber nicht S^ω-äquivalent, weil z ein unendliches Wort akzeptiert, z' aber nicht.

Selbsttestaufgabe 2.2.2

Wir setzen das Beispiel 2.2.1 fort und vergleichen die gleichen Paare wie in Beispiel 2.2.1 auf Verweigerungsäquivalenz.

1. Sind die Zustände v_1 und w_1 verweigerungsäquivalent? Geben Sie bei nicht verweigerungsäquivalenten Paaren ein Paar (w, A') an, das ein Verweigerungspaar des einen aber nicht des anderen Zustandes ist.
2. Sind die Zustände w_1 und x_1 verweigerungsäquivalent? Geben Sie bei nicht verweigerungsäquivalenten Paaren ein Paar (w, A') an, das ein Verweigerungspaar des einen aber nicht des anderen Zustandes ist.
3. Sind die Zustände v_1 und z_1 verweigerungsäquivalent? Geben Sie bei nicht verweigerungsäquivalenten Paaren ein Paar (w, A') an, das ein Verweigerungspaar des einen aber nicht des anderen Zustandes ist.
4. Sind die Zustände v_1 und y_1 verweigerungsäquivalent? Geben Sie bei nicht verweigerungsäquivalenten Paaren ein Paar (w, A') an, das ein Verweigerungspaar des einen aber nicht des anderen Zustandes ist.

Musterlösung zu Selbsttestaufgabe 2.2.2

1. Nein, $(a, \{b\})$ kann von w_1 aus verweigert werden (man macht einen Schritt von w_1 nach w_3), aber nicht von v_1 aus.
2. Ja.
3. Nein. Das folgt bereits daraus, dass die Zustände nicht sprachäquivalent sind. Von Zustand z_1 aus kann das Paar $(ab, \{a\})$ verweigert werden, nicht jedoch von v_1 aus.
4. Ja.

2.3 Bisimulation

Wir wollen nun eine Verhaltensäquivalenz kennen lernen, die alle bisher vorgestellten Verhaltensäquivalenzen subsummiert und damit ein vollständigeres Bild der Nutzererfahrung eines Transitionssystems ergibt: Bisimulation. Wir werden zudem sehen, dass Bisimulation sich auch algorithmisch als noch bedeutend angenehmer erweist als S-Sprachäquivalenz. Im Fall von nichtdeterministischen endlichen Automaten – und damit ganz analog auch bei Transitionssystemen – ist die S-Sprachäquivalenz zu entscheiden PSPACE-vollständig, wohingegen wir ein Polynomzeitverfahren kennen lernen werden, um zu entscheiden, ob zwei Zustände bisimular sind.

Definition 2.3.1 : Bisimulation

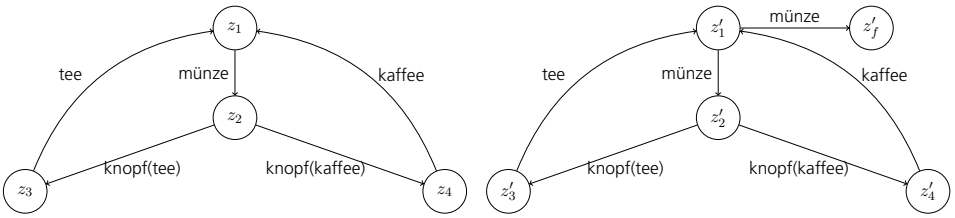
Es sei $(Z, A, \to)$ ein Transitionssystem und $R \subseteq Z \times Z$ eine Relation. Dann nennen wir R eine Bisimulationsrelation oder kurz eine Bisimulation, falls die folgenden Eigenschaften für alle $(z_1, z_2) \in R$ gelten:

1. Falls es ein $a \in A$ und ein $z_1' \in Z$ gibt, so dass $z_1 \xrightarrow{a} z_1'$, dann gibt es ein z_2' so dass $z_2 \xrightarrow{a} z_2'$ und $(z_1', z_2') \in R$
2. Falls es ein $a \in A$ und ein $z_2' \in Z$ gibt, so dass $z_2 \xrightarrow{a} z_2'$, dann gibt es ein z_1' so dass $z_1 \xrightarrow{a} z_1'$ und $(z_1', z_2') \in R$

Wenn es eine Bisimulationsrelation R gibt, die zwei Zustände z_1, z_2 in Relation setzt, also $(z_1, z_2) \in R$, dann nennen wir z_1 und z_2 bisimular, in Zeichen $z_1 \sim z_2$. Die Relation $\sim$ nennen wir die Bisimularität.

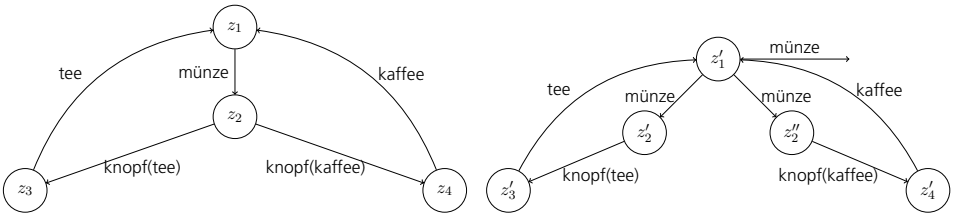
Hinweis: „Bisimulation“ heißt in der englischsprachigen Literatur ebenfalls „bisimulation“, allerdings verwendet man im Englischen statt des Begriffs „bisimular“ den Begriff „bisimilar“.

Alle Beispiele, die wir bisher betrachtet haben, die nicht sprachäquivalent oder verweigerungsäquivalent waren, sind ebenfalls nicht bisimular. Betrachten wir noch einmal unser Beispiel eines Heißgetränkeautomaten, so können wir feststellen, dass die Zustände z_1 und z_1' nicht bisimular sind:



Eine Bisimulationsrelation, die (z_1, z_1') enthält, muss wegen Bedingung 2 auch (z_2, z_f') enthalten, denn z_1' kann einen münze-Schritt zu z_f' ausführen und der einzige Zustand, der mit einer münze-Transition von z_1 aus erreichbar ist, ist der Zustand z_2. Allerdings kann (z_2, z_f') in keiner Bisimulationsrelation liegen, da z_2 (beispielsweise) eine knopf(tee)-Transition ermöglicht, z_f' aber nicht.

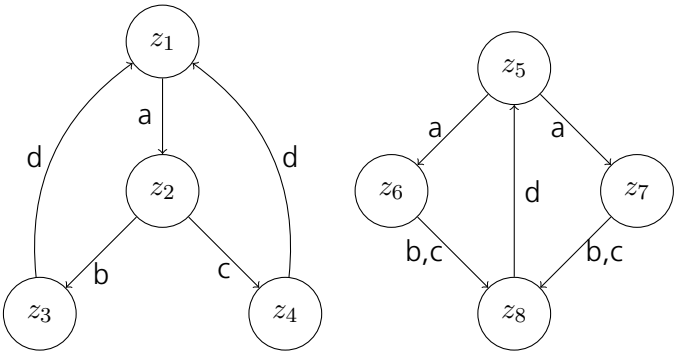
Das klassische Beispiel mit dem Bisimulation motiviert wird, ist aber das folgende:



Auch in diesem Fall sind die Zustände z_1 und z_1' nicht bisimular, denn damit eine Bisimulationsrelation z_1 und z_1' in Relation setzen kann, muss nach Bedingung 2 auch z_2 mit z_2' in Relation gesetzt werden, da von z_1' aus eine münze-Transition zu z_2' möglich ist und der einzige Zustand, der von z_1 aus mit einer münze-Transition erreicht werden kann z_2 ist. Nun führt Bedingung 1 zum Widerspruch, denn von z_2 aus ist es möglich, eine taste(kaffee)-Transition durchzuführen, es gibt aber keinen Zustand, der mit einer taste(kaffee)-Transition von z_2' aus erreichbar ist.

Dieses Beispiel zeigt eine weitere Hinsicht, in der Sprachäquivalenz nicht hinreichend ist, um die Nutzererfahrung eines Software-Systems abzubilden. Das linke Modell ermöglicht dem Nutzer nach Einwurf einer Münze die Wahl, die Kaffee-Taste zu drücken um einen Kaffe zu erhalten, oder die Tee-Taste zu drücken, um einen Tee zu erhalten. In dem rechten Modell hingegen wird die Wahl, ob der Nutzer die eine oder die andere Taste wird drücken können, bereits bei der Eingabe der Münze nichtdeterministisch getroffen. Je nach konkreter Umsetzung des Nichtdeterminismus bedeutet das, dass der Nutzer bereits zum Zeitpunkt des Einwurfs der Münze entscheiden muss, welches Getränk er bestellen möchte, oder gar, dass der Nutzer gar keinen Einfluss darauf hat, welches der beiden Getränke er erhält. In jedem Fall wird die Nutzererfahrung im rechten Modell signifikant anders ausfallen als im linken Modell, obwohl die Zustände z_1 und z_1' sprachäquivalent sind.

Wir betrachten nun ein nicht-triviales Beispiel für zwei Zustände, die bisimular sind:



Die Zustände z_1 und z_5 sind bisimular, wie man anhand der folgenden Bisimulationsrelation erkennen kann:

$$R = \{(z_1, z_5), (z_2, z_6), (z_2, z_7), (z_3, z_8), (z_4, z_8)\}$$

Zu Übungszwecken überprüfen wir diese Relation darauf, ob sie tatsächlich eine Bisimulationsrelation ist. Hierfür müssen wir die beiden Bedingungen, die wir an Bisimulationsrelationen gestellt haben, für jedes Paar in der Relation überprüfen.

(z_1, z_5) Von Zustand z_1 aus ist nur eine a-Transition möglich, mit der der Zustand z_2 erreicht wird. Von z_5 aus kann man beispielsweise den a-Nachfolger z_6 wählen und sieht, dass $(z_2, z_6) \in R$ gilt. Umgekehrt kann von z_6 aus nur eine a-Transaktion durchgeführt werden, es können allerdings zwei mögliche Nachfolgezustände gefunden werden: z_6 und z_7. Von z_1 aus können wir in beiden Fällen nur z_2 auswählen, das ist allerdings kein Problem, denn sowohl $(z_2, z_6) \in R$, als auch $(z_2, z_7) \in R$.

(z_2, z_6) Von z_2 aus ist eine b-Transition möglich, die zu z_3 führt. Von z_6 aus gibt es ebenfalls nur eine b-Transaktion, die zu z_8 führt und $(z_3, z_8) \in R$. Weiterhin ist von z_2 aus eine c-Transaktion möglich, die zu z_4 führt und wiederum ist z_8 der einzige c-Nachfolger von z_6. Es gilt $(z_4, z_8) \in R$. Umgekehrt führt eine b-Transaktion von z_6 aus zu z_8 und von z_2 aus gibt es nur eine mögliche b-Transaktion, die zu z_3 führt. Es gilt $(z_3, z_8) \in R$. Schließlich führt die c-Transaktion von z_6 aus zu z_8, wohingegen von z_2 aus nur eine c-Transaktion möglich ist, die zu z_4 führt. Es gilt $(z_4, z_8) \in R$ wie gefordert.

(z_2, z_7) Analog zum vorherigen Fall.

(z_3, z_8) Von Zustand z_3 aus gibt es nur eine Transition: Eine d-Transition zu z_1. Von z_8 aus gibt es ebenfalls eine d-Transition, die zu z_5 führt und $(z_1, z_5) \in R$. Umgekehrt gibt es von z_8 aus nur eine d-Transition. Diese führt zu z_5. Von z_3 aus gibt es ebenfalls eine d-Transition, die zu z_1 führt und $(z_1, z_5) \in R$.

(z_4, z_8) Analog zum vorherigen Fall.

Wir haben also für alle Paare in der Relation die Bedingungen an eine Bisimulation überprüft und sind daher sicher, dass R tatsächlich eine Bisimulation ist.

Selbsttestaufgabe 2.3.1

Wir setzen das Beispiel 2.2.1 fort und vergleichen die gleichen Paare wie in Beispiel 2.2.1 auf Bisimularität.

1. Sind die Zustände v_1 und w_1 bisimular? Geben Sie bei bisimularen Paaren eine Bisimulationsrelation an, die (v_1, w_1) enthält. Anderenfalls beweisen Sie, dass die beiden Zustände nicht bisimular sein können.
2. Sind die Zustände w_1 und x_1 bisimular? Geben Sie bei bisimularen Paaren eine Bisimulationsrelation an, die (w_1, wx_1) enthält. Anderenfalls beweisen Sie, dass die beiden Zustände nicht bisimular sein können.
3. Sind die Zustände v_1 und z_1 bisimular? Geben Sie bei bisimularen Paaren eine Bisimulationsrelation an, die (v_1, z_1) enthält. Anderenfalls beweisen Sie, dass die beiden Zustände nicht bisimular sein können.
4. Sind die Zustände v_1 und y_1 bisimular? Geben Sie bei bisimularen Paaren eine Bisimulationsrelation an, die (v_1, y_1) enthält. Anderenfalls beweisen Sie, dass die beiden Zustände nicht bisimular sein können.

Musterlösung zu Selbsttestaufgabe 2.3.1

1. Nein. Angenommen R sei eine Bisimulationsrelation, die (v_1, w_1) enthält. Dann gilt: Da $v_1 \xrightarrow{a} v_2$ muss es einen Zustand w geben, so dass $w_1 \xrightarrow{a} w$ und $(v_2, w) \in$. Wir können hierzu aus zwei Zuständen wählen: $w = w_2$ oder $w = w_3$. Falls $w = w_2$, dann kann aber kein Antwortschritt auf den Schritt $v_2 \xrightarrow{c} v_1$ gegeben werden. Umgekehrt, wenn $w = w_3$, dann kann kein Antwortschritt auf den Schritt $v_2 \xrightarrow{b} v_1$ gegeben werden. Da die beiden Zustände nicht verweigerungsäquivalent sind und Bisimulation Verweigerungsäquivalenz impliziert, wäre dies als alternative Argumentation ebenfalls hinreichend.
2. Nein. Angenommen R sei eine Bisimulationsrelation, die (w_1, x_1) enthält. Dann gilt: Da $x_1 \xrightarrow{a} x_4$ muss es einen Zustand w geben, so dass $w_1 \xrightarrow{a} w$ und $(w, x_4) \in$. Wir können hierzu aus zwei Zuständen wählen: $w = w_2$ oder $w = w_3$. Falls $w = w_2$, dann kann aber kein Antwortschritt auf den Schritt $x_4 \xrightarrow{c} x_1$ gegeben werden. Umgekehrt, wenn $w = w_3$, dann kann kein Antwortschritt auf den Schritt $x_4 \xrightarrow{b} x_1$ gegeben werden.

3. Nein, angenommen $(v_1, z_1) \in R$, dann muss wegen $v_1 \xrightarrow{a} v_2$ auch $(v_2, z_2) \in R$ (da z_2 der einzige mögliche Antwort-Zustand ist). Dann wiederum muss, beispielsweise wegen $v_2 \xrightarrow{b} v_1$, auch $(v_1, z_3) \in R$ gelten (wiederum ist z_3 der Zustand der beim einzigen möglichen Antwortschritt erreicht wird). Nun gibt es aber eine Transition $v_1 \xrightarrow{a} v_2$, auf die von z_3 aus nicht geantwortet werden kann. Da die beiden Zustände nicht sprachäquivalent sind und Bisimulation Sprachäquivalenz impliziert, wäre dies als alternative Argumentation ebenfalls hinreichend.

4. Ja. Das kann man anhand der Bisimulationsrelation

$$R = \{(v_1, y_1), (v_2, y_2), (v_1, y_3)\}$$

einsehen.

Aus der Definition der Bisimularität folgt nicht unmittelbar, dass es sich hierbei ebenfalls um eine Bisimulationsrelation handelt. Diesen Umstand wollen wir im Folgenden beweisen, unter zuhilfenahme der Aussage, dass die Vereinigung von Bisimulationen wiederum eine Bisimulation ist.

Satz 2.3.2

Sei $(Z, A, \to)$ ein Transitionssystem und $R_i, i \in I$ eine Familie (nicht notwendigerweise endlich oder abzählbar) von Bisimulationsrelationen.

1. Es gilt $R = \bigcup\{R_i \mid i \in I\}$ ist eine Bisimulationsrelation.
2. $\sim$ ist eine Bisimulationsrelation.

Beweis.

1. Wir weisen die erste Eigenschaft von Bisimulationsrelationen nach, die zweite Eigenschaft ist analog. Es sei $(z_1, z_2) \in R$ beliebig gewählt und $a \in A$, $z_1' \in Z$ derart, dass $z_1 \xrightarrow{a} z_1'$ gilt. Dann muss es eine Relation R_i, $i \in I$ geben, so dass $(z_1, z_2) \in R_i$. Nach Voraussetzung ist R_i eine Bisimulationsrelation, daher muss es ein z_2' geben, so dass $z_2 \xrightarrow{a} z_2'$ und $(z_1', z_2') \in R_i$. Da $R_i \subseteq R$ nach Definition von R, folgt also $(z_1', z_2') \in R$ und die erste Bisimulationseigenschaft ist somit erfüllt.

2. Nach Definition ist $\sim = \bigcup\{R \mid R \subseteq Z \times Z, R \text{ ist Bisimulationsrelation}\}$, zusammen mit Teil 1 folgt, dass $\sim$ eine Bisimulationsrelation ist.

□

Wir haben Bisimulation eine Verhaltensäquivalenz genannt, sind aber bislang den Beweis schuldig geblieben, dass die Bisimularität auch tatsächlich eine Äquivalenzrelation ist. Wie wir im oben diskutierten Beispiel gesehen habe, gilt im Allgemeinen nicht, dass jede Bisimulationsrelation eine Äquivalenzrelation ist, denn beispielsweise enthielt die Beispiels-Bisimulation kein reflexives Paar.

Satz 2.3.3

Es sei $(Z, A, \rightarrow)$ ein Transitionssystem, dann gilt:

1. Die Relation $\mathsf{Id}_Z = \{(z, z) \mid z \in Z\}$ ist eine Bisimulationsrelation (Reflexivität).
2. Wenn R eine Bisimulationsrelation ist, dann ist auch $R^{-1} = \{(z_2, z_1) \mid (z_1, z_2) \in R\}$ eine Bisimulationsrelation (Symmetrie).
3. Wenn $R_1, R_2 \subseteq Z \times Z$ Bisimulationsrelationen sind, dann ist auch $R_1 R_2 = \{(z_1, z_3) \mid \exists z_2 \in Z : (z_1, z_2) \in R_1 \wedge (z_2, z_3) \in R_2\}$ eine Bisimulationsrelation (Transitivität).

Beweis. Punkt 1 ist offensichtlich, Punkt 2 folgt daraus, dass die Bedingungen für Bisimulationsrelationen symmetrisch sind. Es bleibt also nur Punkt 3 zu beweisen. Seien also $z_1, z_3 \in Z$ gegeben so dass $(z_1, z_3) \in R_1 R_2$. Dann gibt es ein $z_2 \in Z$ so dass $(z_1, z_2) \in R_1$ und $(z_2, z_3) \in R_2$. Sei weiterhin ein $a \in A$ und ein $z_1' \in Z$ gegeben, so dass $z_1 \xrightarrow{a} z_1'$. Dann gibt es, weil R_1 eine Bisimulation ist, einen Zustand $z_2' \in Z$ so dass $z_2 \xrightarrow{a} z_2'$ und $(z_1', z_2') \in R_1$. Da R_2 eine Bisimulation ist, muss es dann aber einen Zustand z_3' geben, so dass $z_3 \xrightarrow{a} z_3'$ und $(z_2', z_3') \in R_2$. Nach Definition von $R_1 R_2$ ist dann $(z_1, z_3) \in R_1 R_2$. Analog lässt sich der Fall argumentieren, dass es einen Zustand $z_3' \in Z$ und $a \in A$ gibt, so dass $z_3 \xrightarrow{a} z_3'$. $R_1 R_2$ ist also Bisimulationsrelation. □

Wir erinnern uns an das Verfahren zur Minimierung von deterministischen endlichen Automaten, das in der Vorlesung „Theoretische Informatik“ unter dem Namen „Table-Filling Algorithmus“ und in der Vorlesung „Grundlagen der Informatik 2“ unter dem Namen „[Algorithmus zur Erstellung des] Minimalautomat[en]“ vorgestellt wurde:

Algorithmus 2.3.4 : DFA-Minimierung

Eingabe: DFA M (Zustände, die vom Startzustand aus nicht erreichbar sind, sind bereits entfernt)
Ausgabe: Mengen von erkennungsäquivalenten Zuständen

1. Stelle eine Tabelle aller Zustandspaare $\{z, z'\}$ mit $z \neq z'$ auf.
2. Markiere alle Paare $\{z, z'\}$ wobei z ein Endzustand ist und z' nicht (oder umgekehrt)

   (z, z' sind sicherlich nicht erkennungsäquivalent.)
3. Für jedes noch unmarkierte Paar $\{z_1, z_2\}$ und jede Aktion $a \in A$ betrachte den a-Nachfolger z_1' von z_1 und den a-Nachfolger z_2' von z_2. Teste, ob $\{z_1', z_2'\}$ bereits markiert ist. Wenn ja: markiere auch $\{z_1, z_2\}$.

(Von z, z' gibt es Übergänge zu nicht erkennungsäquivalenten Zuständen, sie können daher nicht erkennungsäquivalent sein.)

4. Wiederhole den vorherigen Schritt, bis sich keine Änderung in der Tabelle mehr ergibt.

5. Für alle jetzt noch unmarkierten Paare $\{z, z'\}$ gilt: z und z' sind erkennungsäquivalent.

Dieser Algorithmus ist ein sogenannter Partitionsverfeinerungsalgorithmus. Die Idee hierbei ist, dass eine Äquivalenzrelation schrittweise verfeinert wird, bis sie sich nicht mehr ändert. In diesem Fall sind die Partitionen anhand der Tabelle abzulesen: Zwei Zustände z, z' stehen in Relation R_i, falls im i-ten Schritt das Paar $\{z, z'\}$ noch nicht markiert wurde. Der grundsätzliche Aufbau eines solchen Algorithmus lässt sich gut anhand dieses Beispiels erläutern: Zunächst werden alle Zustände in eine Äquivalenzklasse gesetzt (Schritt 1), anschließend werden sukzessive Äquivalenzklassen aufgetrennt (Schritte 2-3), bis die Äquivalenzklassen sich nicht mehr ändern (Schritte 4-5).

Um die Bisimularität zu berechnen, verwenden wir nun ebenfalls einen Paritionsverfeinerungslagorithmus. Allerdings trennen wir in diesem Fall Zustandspaare anhand der definierenden Eigenschaft der Bisimulation: Immer wenn wir in der aktuellen Relation für ein Paar z, z' feststellen, dass die Bedingungen an eine Bisimulationsrelation nicht erfüllt sind, trennen wir die beiden Zustände in verschiedene Äquivalenzklassen auf. Konkret sieht der Algorithmus wie folgt aus:

Algorithmus 2.3.5

Eingabe: Transitionssystem $(Z, A, \to)$
Ausgabe: Bisimularität $\sim$ auf $(Z, A, \to)$

1. Wähle zunächst die gröbste Äquivalenzrelation $\sim_0 = Z \times Z$, die alle Paare von Zuständen in Relation setzt.

2. Gegeben die Relation $\sim_i$, berechne die Nachfolgerelation $\sim_{i+1}$ wie folgt: Für jedes Paar $(z_1, z_2) \in \sim_i$ ist $(z_1, z_2) \in \sim_{i+1}$ genau dann wenn für alle $a \in A$ gilt:
   a) Für jedes z_1' mit $z_1 \xrightarrow{a} z_1'$ gibt es ein z_2' mit $z_2 \xrightarrow{a} z_2'$ und $z_1' \sim_i z_2'$. Die erste Bedingung an Bisimulationsrelationen muss also für das Paar (z_1, z_2) in $\sim_i$ erfüllt sein.
   b) Für jedes z_2' mit $z_2 \xrightarrow{a} z_2'$ gibt es ein z_1' mit $z_1 \xrightarrow{a} z_1'$ und $z_1' \sim_i z_2'$. Die zweite Bedingung an Bisimulationsrelationen muss also für das Paar (z_1, z_2) in $\sim_i$ erfüllt sein.

3. Sobald $\sim_n = \sim_{n+1}$, d.h. die Relation wird in einem Schritt nicht weiter verfeinert, terminiert das Verfahren und gibt $\sim_n$ aus.

Ganz analog zum Algorithmus für den Minimalautomaten kann dieser Algorithmus natür-

lich auch mithilfe einer $Z \times Z$-Tabelle implementiert werden, in der immer dann, wenn in Schritt i ein Paar (z_1, z_2) getrennt wird, deren Eintrag markiert wird. In der weiteren Argumentation werden wir allerdings die Zwischenergebnisse $\sim_i$ häufiger referenzieren, so dass die abstraktere Formulierung hilfreich ist.

Wir beginnen die Analyse des Algorithmus damit, zu beweisen, dass er tatsächlich das korrekte Ergebnis liefert.

Satz 2.3.6

Es sei $(Z, A, \rightarrow)$ ein Transitionssystem. Wenn Algorithmus 2.3.5 terminiert und $\sim_n$ als Ergebnis zurückliefert, dann ist $\sim_n = \sim$ die Bisimularität auf $(Z, A, \rightarrow)$.

Beweis. Wir müssen zeigen, dass jede Bisimulationsrelation auf $(Z, A, \rightarrow)$ Teilemenge oder gleich $\sim_n$ ist. Sei hierzu eine beliebige Bisimulation $R \subseteq Z \times Z$ gegeben. Wir zeigen induktiv, dass $R \subseteq \sim_i$ für alle $0 \leq i \leq n$.

Der Induktionsanfang ist trivial, denn $\sim_0 = Z \times Z \supseteq R$. Angenommen also $R \subseteq \sim_i$ für ein $i \leq n$, dann ist zu zeigen, dass $R \subseteq \sim_{i+1}$. Sei $(z_1, z_2) \in R$ beliebig gewählt, dann gilt für alle $a \in A$ und $z_1' \in Z$: Falls $z_1 \xrightarrow{a} z_1'$, dann gibt es ein $z_2' \in Z$ so dass $z_2 \xrightarrow{a} z_2'$ und $(z_1', z_2') \in R$, da R eine Bisimulationsrelation ist. Da nach Induktionsvoraussetzung $R \subseteq \sim_i$ gilt, gilt auch $(z_1', z_2') \in \sim_i$. Analog: Falls $z_2 \xrightarrow{a} z_2'$, dann gibt es ein $z_1' \in Z$ so dass $z_1 \xrightarrow{a} z_1'$ und $(z_1', z_2') \in R$, da R eine Bisimulationsrelation ist. Da nach Induktionsvoraussetzung $R \subseteq \sim_i$ gilt, gilt auch $(z_1', z_2') \in \sim_i$. Also gilt nach Definition des zweiten Schritts in Algorithmus 2.3.5 auch $(z_1, z_2) \in \sim_{i+1}$. □

Algorithmus 2.3.5 terminiert nicht notwendigerweise, wenn das Transitionssystem unendlich viele Zustände hat, wie man relativ einfach an Beispiel 2.2.3 ersehen kann: In Relation $\sim_i$, $i \in \mathbb{N}$ können Zustände $z_{i,1}$ und $z_{i+1,1}$ noch nicht getrennt werden, schon aber in Relation $\sim_{i+1}$. Daher wird der Algorithmus auf diesem Beispiel niemals terminieren. Allerdings ist der Fall unendlich großer Zustandsmengen im Allgemeinen ohnehin nicht wirklich praktikabel, denn bereits ein einzelner Schritt bedarf des Vergleichs unendlich vieler Paare von Zustände und wird daher unendlich viele Operationen in Anspruch nehmen. Dass der Algorithmus dennoch praktische Relevanz hat, liegt an seinem günstigen Terminierungsverhalten im Falle endlich vieler Zustände. Wir können zeigen, dass der Algorithmus nur polynomiell in der Zahl der Zustände viele Schritte benötigt. Bereits ohne weitere Analyse des Algorithmus kann man dies durch eine grobe Abschätzung feststellen, denn es gilt stets $\sim_i \supseteq \sim_{i+1}$, also kann der Algorithmus maximal $| \sim_0 | = |Z| \cdot |Z|$ viele Iterationen benötigen. In jeder Iteration müssen maximal $|Z| \cdot |Z|$ viele Paare verglichen werden und jeder Vergleich bedarf maximal der Kontrolle $|A| \cdot |Z| \cdot 2$ Nachfolger. Im Falle dessen, dass man die Zwischenergebnisse in Matrixform dokumentiert, dauert ein solcher Überprüfungsschritt maximal $|Z|$ viele Schritte, also kann man direkt ablesen, dass die Zahl der Schritte in Abhängigkeit von der Zustandsmenge $O(|Z|^4)$ nicht überschreiten kann. Dass die Laufzeit ohne großen Aufwand noch etwas schärfer auf $O(|Z|^3)$ abgeschätzt werden kann, folgt aus dem Umstand, dass jedes $\sim_i$ eine Äquivalenzrelation ist.

Lemma 2.3.7

Für alle $i \in \mathbb{N}_0$ gilt, dass $\sim_i$ in Algorithmus 2.3.5 eine Äquivalenzrelation ist.

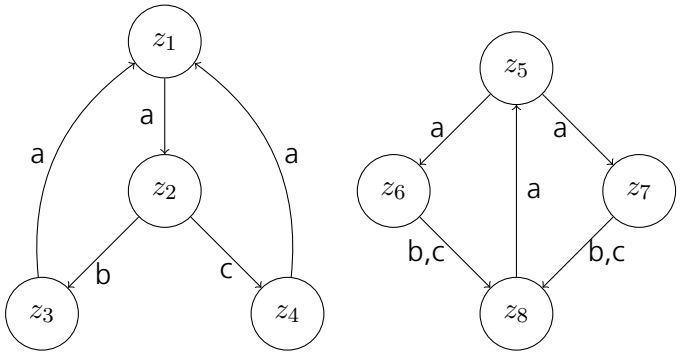
Beweis. $\sim_0$ ist trivialerweise eine Äquivalenzrelation. Die Iteration des Algorithmus erhält Reflexivität, da die Bedingungen für Bisimulation auf reflexiven Relationen für reflexive Paare automatisch erfüllt sind. Ebenso einfach kann man einsehen, dass die Iteration Symmetrie erhält, denn die Bedingungen für Bisimulationsrelationen sind symmetrisch. Es bleibt also nur zu zeigen, dass auch Transitivität erhalten wird. Sei also $\sim_i$ transitiv, $(z_1, z_2) \in \sim_{i+1}$ und $(z_2, z_3) \in \sim_{i+1}$. Zu zeigen ist, dass dann auch $(z_1, z_3) \in \sim_{i+1}$ gilt.

Da $\sim_{i+1} \subseteq \sim_i$ und $\sim_i$ transitiv ist, wissen wir, dass $(z_1, z_3) \in \sim_i$ gilt. Sei nun irgendein $a \in A$, $z_1' \in Z$ gegeben, so dass $z_1 \xrightarrow{a} z_1'$, dann gibt es, wegen $(z_1, z_2) \in \sim_{i+1}$ ein z_2', so dass $z_2 \xrightarrow{a} z_2'$ und $(z_1', z_2') \in \sim_i$ und, weil $(z_2, z_3) \in \sim_{i+1}$, ebenfalls ein $z_3' \in Z$, so dass $(z_2', z_3') \in \sim_i$. Da $\sim_i$ transitiv ist, ist auch $(z_1', z_3') \in \sim_i$ und damit ist die erste Bedingung erfüllt, damit $(z_1, z_3) \in \sim_{i+1}$ ist. Die zweite Bedingung kann analog argumentiert werden. □

Die Zahl der Iterationen des Algorithmus lässt sich mit dieser Information nun auf maximal $|Z|$ abschätzen, denn eine Äquivalenzelation auf $|Z|$ Elementen kann maximal $|Z|$ mal verfeinert werden. Damit ist die Laufzeit des Algorithmus insgesamt also auf $O(|Z|^3)$ Operationen beschränkt. Es sei hierzu noch angemerkt, dass die Abschätzung der Laufzeit im Allgemeinen relativ grob ist, denn Transitionssysteme sind oft eher dünn besetzt mit Transitionen, wodurch die Zahl der notwendigen Vergleiche in der Praxis oft deutlich geringer ausfällt. Außerdem sei an dieser Stelle noch einmal darauf hingewiesen, dass dieser Algorithmus kein deterministisches Transitionssystem voraussetzt, wohingegen im Minimierungalgorithmus ein deterministischer Automat gefordert werden muss. Das Sprachäquivalenzproblem für nichtdeterministische Automaten ist PSPACE-vollständig, so dass nach momentanem Stand Bisimularität bedeutend effizienter entschieden werden kann (vgl. Kurs zur Theoretischen Informatik, PSPACE $\supseteq$ NP $\supseteq$ P).

Beispiel 2.3.8

Wir betrachten das folgende Transitionssystem:



Wenden wir den Bisimulationsalgorithmus auf dieses Transitionssystem an, so beginnen wir mit einer Äquivalenzrelation $\sim_0$, die alle Zustände in die gleiche Äquivalenzklasse setzt. Wir schreiben die Äquivalenzrelation mithilfe ihrer Äquivalenzklassen und erhalten:

$$\sim_0 = \{z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8\}$$

z_1 und z_2 können im nächsten Schritt als nicht bisimular erkannt werden: z_1 kann eine a-Transition zu z_2 ausführen, aber z_2 kann keine a-Transition ausführen. Aus dem gleichen Grund kann z_1 von z_6 und z_7 separiert werden.

Betrachten wir als nächstes z_2, so können wir z_2 von z_3, z_4 und z_8 separieren, da z_2 einen b-Schritt durchführen kann, aber keiner der anderen Zustände kann einen b-Schritt ausführen.

Der Zustand z_3 bietet eine a-Transition an und ist daher bereits im ersten Schritt als nicht bisimular zu den Zuständen z_6 und z_7 zu erkennen, die keine a-Transition ermöglichen. Analog kann man argumentieren, dass z_4 nicht bisimular zu z_6 und z_7 ist.

Betrachten wir als nächstes z_5, so stellen wir fest, dass z_5 eine a-Transition anbietet, aber z_6 und z_7 nicht, also von z_6 und z_7 getrennt werden kann.

Schließlich sehen wir, dass z_6 und z_7 jeweils eine b-Transition ermöglichen, z_8 aber nicht, so dass wir z_6 und z_7 von z_8 separieren können.

Sie können beobachten, dass wir einen Zustand mit Index i immer nur mit den Zuständen mit größerem Index verglichen haben – auf Grund der Symmetrie der zu vergleichenden Eigenschaften müssen wir nämlich, wenn wir ein Paar (z_i, z_j) bereits untersucht haben, das Paar (z_j, z_i) nicht mehr betrachten.

Nach dem ersten Schritt des Algorithmus erhalten wir also die Äquivalenzklassen:

$$\sim_1 = \{\{z_1, z_3, z_4, z_5, z_8\}, \{z_2, z_6, z_7\}\}$$

Im zweiten Schritt stellen wir nun fest, dass z_1 von z_3 getrennt werden muss, denn z_1 kann einen a-Schritt zu z_2 machen, z_3 kann mit a aber nur den Zustand z_1 erreichen und $z_3 \not\sim_1 z_1$. Analog kann man einsehen, dass z_1 von z_4 und z_8 getrennt werden müssen. Weiterhin muss z_3 von z_5 getrennt werden, denn z_3 erlaubt einen a-Schritt

zu z_1, z_5 kann mit einem a-Schritt hingegen nur z_6 oder z_7 erreichen aber $z_1 \not\sim_1 z_6$ und $z_1 \not\sim_1 z_7$. Wiederum ist die Argumentation für z_4 analog.

Schließlich müssen auch z_5 und z_8 getrennt werden: z_5 kann eine a-Transition zu z_6 ausführen, wohingegen z_8 mit einer a-Transition ausschließlich z_5 erreichen kann und $z_5 \not\sim_1 z_6$. Insgesamt erhalten wir die folgenden Äquivalenzklassen:

$$\sim_2 = \{\{z_3, z_4, z_8\}, \{z_1, z_5\}, \{z_2, z_6, z_7\}\}$$

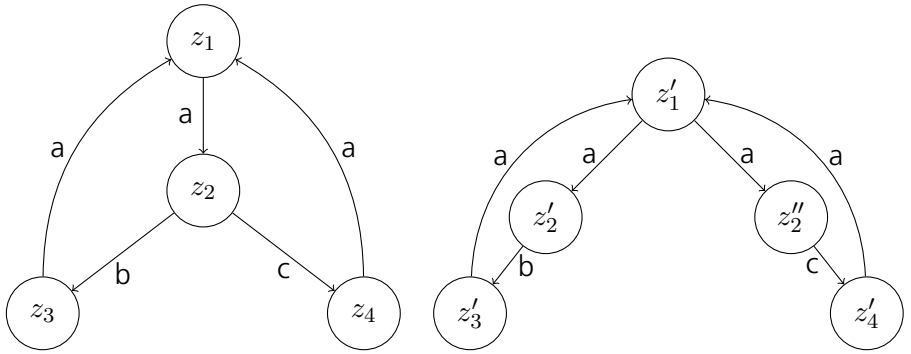
In der darauffolgenden Iteration des Algorithmus finden wir keine Paare von Zuständen mehr, die getrennt werden müssten, also gilt $\sim_3 = \sim_2$ und somit $\sim = \sim_2$ und der Algorithmus terminiert.

Bevor wir eine weitere Charakterisierung der Bisimularität kennen lernen, betrachten wir zunächst einige Beispiele für den Bisimulationsalgorithmus.

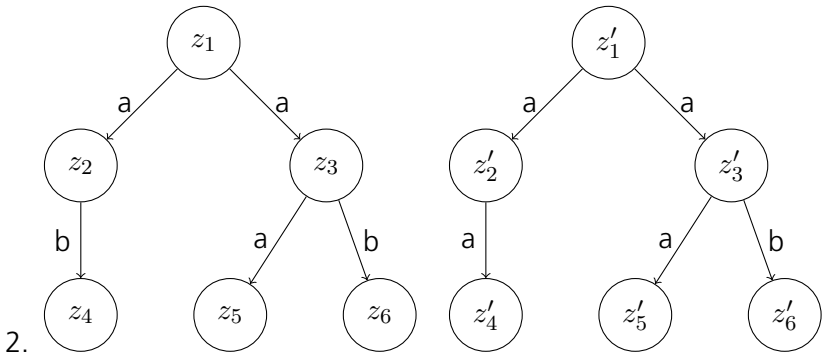
Selbsttestaufgabe 2.3.2

Führen Sie den Bisimulationsalgorithmus auf den folgenden beiden Beispielen durch:

1.



2.



Musterlösung zu Selbsttestaufgabe 2.3.2

1.

$$\sim_0 = \{\{z_1, z_2, z_3, z_4, z'_1, z'_2, z''_2, z'_3, z'_4\}\}$$

$$\sim_1 = \{\{z_1, z_3, z_4, z_1', z_3', z_4'\}, \{z_2\}, \{z_2'\}, \{z_2''\}\}$$
$$\sim_2 = \{\{z_1\}, \{z_1'\}, \{z_3, z_4, z_3', z_4'\}, \{z_2\}, \{z_2'\}, \{z_2''\}\}$$
$$\sim_3 = \{\{z_1\}, \{z_1'\}, \{z_3, z_4\}, \{z_3', z_4'\}, \{z_2\}, \{z_2'\}, \{z_2''\}\}$$
$$\sim_4 = \{\{z_1\}, \{z_1'\}, \{z_3, z_4\}, \{z_3', z_4'\}, \{z_2\}, \{z_2'\}, \{z_2''\}\}$$

2.

$$\sim_0 = \{\{z_1, z_2, z_3, z_4, z_5, z_6, z_1', z_2', z_3', z_4', z_5', z_6'\}\}$$
$$\sim_1 = \{\{z_1, z_1', z_2'\}, \{z_2\}, \{z_3, z_3', \}, \{z_4, z_5, z_6, z_4', z_5', z_6'\}\}$$
$$\sim_2 = \{\{z_1\}, \{z_1'\}, \{z_2'\}, \{z_2\}, \{z_3, z_3', \}, \{z_4, z_5, z_6, z_4', z_5', z_6'\}\}$$
$$\sim_3 = \{\{z_1\}, \{z_1'\}, \{z_2'\}, \{z_2\}, \{z_3, z_3', \}, \{z_4, z_5, z_6, z_4', z_5', z_6'\}\}$$

2.4 Das Bisimulationsspiel

Eine intuitive Charakterisierung der Bisimulation ermöglicht das Bisimulationsspiel. Hierbei handelt es sich um ein rundenbasiertes Spiel für zwei Personen mit dem untersucht werden kann, ob zwei gegebene Zustände in einem Transitionssystem bisimular sind.

Definition 2.4.1 : Bisimulationsspiel

Gegeben sei ein Transitionssystem $(Z, A, \rightarrow)$ und zwei Zustände $z_1, z_2 \in Z$. Dann ist das Bisimulationsspiel als Spiel zwischen zwei Spielern, die wir im Folgenden Angreifer (Spieler 1) und Verteidiger (Spieler 2) nennen, wie folgt definiert:

Ausgangssituation Das Spielfeld des Spiels ist das Transitionssystem $(Z, A, \rightarrow)$ und zu Beginn liegen zwei Spielsteine auf dem Spielfeld: Einer auf z_1 und einer auf z_2.

Ablauf einer Spielrunde Der Angreifer beginnt die Runde und wählt zunächst zu Beginn einer jeden Runde einen der beiden Spielsteine auf dem Spielfeld aus. Dieser Spielstein liegt auf einem Zustand z des Transitionssystems. Mit diesem Spielstein führt er dann einen Zug aus, indem er eine Aktion $a \in A$ auswählt und seinen Spielstein auf einen a-Nachfolger von z zieht. Der Verteidiger muss nun den anderen Spielstein verwenden und ebenfalls einen Zug zu einem a-Nachfolger des Zustandes durchführen, auf dem sich sein Spielstein befindet.

Ende des Spiels Das Spiel endet, wenn einer der beiden Spieler keinen Zug mehr ausführen kann.

Siegbedingung Der Angreifer gewinnt, falls er einen Zug ausgeführt hat und der Verteidiger keinen Antwortzug durchführen kann. Der Verteidiger gewinnt, falls

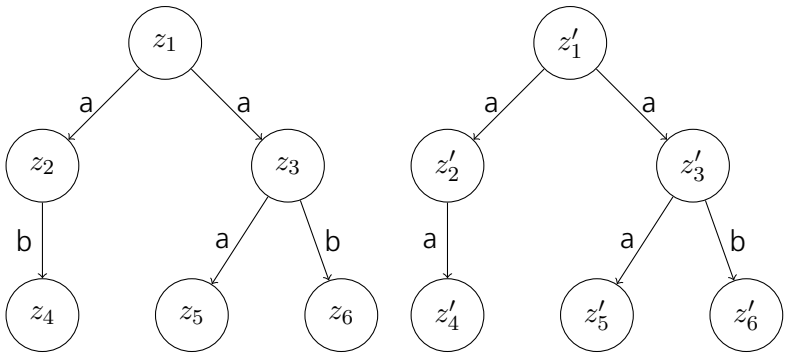
der Angreifer am Zug ist, aber keinen Zug mehr durchführen kann, oder wenn das Spiel unendlich lange dauert.

Möchte man das Bisimulationsspiel spielen, kann man das Spiel noch leicht modifizieren, indem man die Regel hinzufügt, dass das Spiel mit einem Sieg für den Verteidiger endet, wenn eine Spielsituation das zweite Mal in Folge erreicht wird. Wenn der Angreifer eine Siegstrategie besitzt, dann wird diese Situation nicht eintreten, wenn der Angreifer der Siegstrategie folgt. Auf diese Weise wird das Spiel garantiert zu einem Ende geführt. Die Benennung der Spieler erfolgt auf Grund ihrer Rolle in Hinblick auf die Annahme, dass die Ausgangszustände bisimular sind: Der Angreifer versucht, dem anderen Spieler zu zeigen, dass es ein beobachtbares Verhalten gibt, das die beiden Zustände trennt, wohingegen der Verteidiger die These verteidigt, dass die Zustände bisimular sind, indem er zeigt, dass jede Aktion des Angreifers immitiert werden kann.

Es ist wichtig zu beachten, dass der Angreifer in jeder Runde einen der beiden Zustände wählen kann, also insbesondere nach einem Zug auch den betrachteten Zustand wechseln kann. Das ist auch tatsächlich eine notwendige Möglichkeit, denn es gibt Fälle, in denen der Wechsel für den Angreifer zwingend notwendig ist, um den Sieg zu garantieren. Das können wir anhand eines einfachen Beispiels ersehen:

Beispiel 2.4.2

Wir betrachten das Transitionssystem, gegeben in grafischer Form wie folgt:



und untersuchen, ob der Angreifer ohne zwischen dem ersten und zweiten Zustand im Zustandspaar zu wechseln das Bisimulationsspiel gewinnen kann. Angenommen,

der Angreifer wählt den ersten Zustand im Paar, z_1 und zieht mit dem Eingabezeichen a zunächst von z_1 nach z_3, dann kann der Verteidiger durch einen Zug von z'_1 nach z'_3 antworten. Anschließend kann der Angreifer entweder einen a-Zug zu z_5 durchführen, den der Verteidiger mit einem Zug zu z'_5 beantworten kann, oder aber einen b-Zug zu z_6, den der Verteidiger wiederum mit einem b-Zug zu z_6 beantworten kann. In beiden Fällen sind anschließend keine weiteren Schritte möglich, also verliert der Angreifer. Analog sieht man, dass im Fall dessen, dass der Angreifer den zweiten Zustand wählt und einen a-Zug von z'_1 zu z'_3 durchführt, keinen Sieg erzwingen kann.

Beginnt der Angreifer hingegen mit einem a-Zug von z_1 nach z_2, so kann der Verteidiger mit einem a-Zug von z'_1 zu z'_3 antworten. Anschließend bleibt dem Angreifer nur, einen b-Zug von z_2 zu z_4 durchzuführen, den der Verteidiger aber mit einem Zug von z'_3 zu z'_5 beantworten kann. Da nun keine weiteren Züge für den Angreifer übrig bleiben, verliert er das Spiel.

Schließlich bleibt dem Angreifer nur noch die Möglichkeit, einen a-Zug von z'_1 nach z'_2 zu machen, den der Verteidiger allerdings mit einem a-Zug von z_1 nach z_3 beantworten kann. Anschließend bleibt dem Angreifer nurmehr ein a-Zug von z'_2 nach z'_4, den der Verteidiger mit einem Zug von z_3 nach z_5 beantworten kann, wiederum verliert der Angreifer, weil er keinen weiteren Zug durchführen kann.

Bei einem geschickt agierenden Verteidiger verliert der Angreifer also stets das Bisimulationsspiel ausgehend von (z_1, z'_1). Allerdings sind die Zustände z_1 und z'_1 nicht bisimular, denn es gibt keine Bisimulationsrelation R, die (z_1, z'_1) enthält: Wenn $(z_1, z'_1) \in R$ gilt, dann muss auch, nach der ersten Bedingung an eine Bisimulationsrelation mindestens eines der Paare (z_2, z'_2) oder (z_2, z'_3) in der Relation R enthalten sein. Ersteres ist nicht möglich, da von z_2 aus ein b-Übergang existiert, aber von z'_2 aus nicht (erste Bedingung an eine Bisimulationsrelation), zweiteres ist nicht möglich, weil von z'_3 aus ein a-Übergang existiert, aber von z_2 aus nicht (zweite Bedingung an eine Bisimulationsrelation).

Der Angreifer kann allerdings ein Spiel, das mit (z_1, z'_1) begonnen wird, tatsächlich gewinnen, er muss nur zwischendurch den betrachteten Zustand wechseln können. So kann der Angreifer beispielsweise wie folgt sicherstellen, dass er das Spiel gewinnt: Zunächst zieht der Angreifer von z_1 mit einer a-Transition nach z_2. Nun gibt es zwei Fälle. Falls der Verteidiger von z'_1 nach z'_2 zieht, so kann der Angreifer ohne Wechsel gewinnen, indem er einen Zug von z_2 mit b nach z_4 durchführt, auf den der Verteidiger nicht antworten kann. Zieht der Verteidiger hingegen nach z'_3, so muss der Angreifer, wie gerade bereits durchdacht, den Zustand wechseln und in seiner nächsten Runde einen Zug von z'_3 aus durchführen. Von z'_3 aus kann der Angreifer eine a-Transition zu z'_5 durchführen, auf die der Verteidiger nicht mehr reagieren kann.

Im vorherigen Beispiel steckt bereits implizit eine wichtige Beobachtung: Es reicht zum Nachweis, dass zwei Zustände bisimular sind, nicht aus, zu zeigen, dass es einen Spielablauf gibt, bei dem der Verteidiger gewinnt, denn alle Spielabläufe im obigen Beispiel, bei denen der Angreifer den Zustand nicht wechselt, sind auch legale Spielabläufe im Bisimulationsspiel. Umgekehrt reicht es auch nicht aus, dass es einen Spielablauf gibt, in dem der Angreifer das Bisimulationsspiel gewinnt, um zu beweisen, dass zwei Zustände

nicht bisimular sind. Stattdessen muss über die *Existenz einer Gewinnstrategie* gesprochen werden.

Eine Gewinnstrategie ist ein Entscheidungsregelsatz, der dem Spieler in jeder Spielsituation einen eindeutigen Spielschritt vorgibt, der die Eigenschaft hat, dass, wenn der Spieler dem Regelsatz folgt, er, unabhängig von der Spielweise seines Gegenspielers, das Spiel gewinnt. Wir werden nun zeigen, dass eine Gewinnstrategie für den Verteidiger genau dann existiert wenn das Ausgangspaar bisimular ist.

Satz 2.4.3

Es sei $(Z, A, \rightarrow)$ ein Transitionssystem und $z, z' \in Z$ Zustände, dann gilt:

- Falls der Verteidiger eine Gewinnstrategie für das Bisimulationsspiel auf $(Z, A, \rightarrow)$, ausgehend von (z, z') hat, dann sind z und z' bisimular.
- Falls z und z' bisimular sind, dann hat der Verteidiger eine Gewinnstrategie für das Bisimulationsspiel auf $(Z, A, \rightarrow)$, ausgehend von (z, z').

Es gilt also: Der Verteidiger hat *genau dann* eine Gewinnstrategie für das Bisimulationsspiel auf $(Z, A, \rightarrow)$ ausgehend von (z, z'), wenn z und z' bisimular sind.

Beweis. Wir zeigen dies in zwei Schritten:

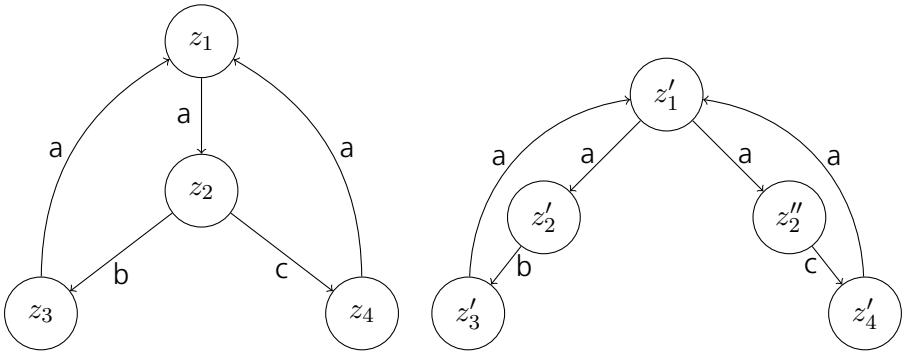
- Nehmen wir zunächst an, der Verteidiger hat eine Gewinnstrategie. Dann können wir eine Bisimulationsrelation, die (z, z') enthält, wie folgt aufbauen: Wir definieren R als die Menge aller Spielsituationen (zu Beginn einer Runde), die von (z, z') aus erreicht werden können, wenn der Verteidiger der Gewinnstrategie folgt. Zu zeigen ist nun, dass beide Bedingungen an eine Bisimulationsrelation für alle Paare $(z_1, z_2) \in R$ und alle Aktionen $a \in A$ erfüllt sind:
  1. Falls es einen Zustand z_1' gibt, so dass $z_1 \xrightarrow{a} z_1'$, dann kann der Angreifer mit einem a-Schritt von z_1 nach z_1' ziehen. Also muss der Verteidiger von z_2 aus einen Antwortschritt durchführen können, der der Gewinnstrategie folgt und einen Zustand z_2' erreicht. Nach Definition ist dann $(z_1', z_2') \in R$.
  2. Analog umgekehrt: Falls es einen Zustand z_2' gibt, so dass $z_2 \xrightarrow{a} z_2'$, dann kann der Angreifer mit einem a-Schritt von z_2 nach z_2' ziehen. Also muss der Verteidiger von z_1 aus einen Antwortschritt durchführen können, der der Gewinnstrategie folgt und einen Zustand z_1' erreicht. Nach Definition ist dann $(z_1', z_2') \in R$.
- Nehmen wir nun an, dass z und z' bisimular sind, es gilt also $(z, z') \in \sim$ und wir erinnern daran, dass $\sim$ selbst eine Bisimulationsrelation ist. Die Gewinnstrategie für den Verteidiger sieht nun wie folgt aus: Angenommen zu Beginn der Spielrunde befindet sich das Spiel in der Situation (z_1, z_2). Wir nehmen überdies an, dass z_1 und z_2 bisimular sind. Wenn der Angreifer einen a-Schritt von z_1 zu z_1' macht, dann muss es nach der ersten Bedingung für eine Bisimulationsrelation gelten, dass es einen

Zustand z_2' gibt, so dass $z_2 \xrightarrow{a} z_2'$ und $(z_1', z_2') \in \sim$: Dann kann der Verteidiger also den Zustand z_2' auswählen und die neue Spielsituation (z_1', z_2') stellt ebenfalls sicher, dass das betrachtete Paar bisimular ist. Der Verteidiger kann also auf jeden Schritt des Angreifers antworten und das Spiel endet entweder nie – wenn der Angreifer unendlich viele Schritte machen kann – oder es endet damit dass der Angreifer keine weiteren Schritte mehr machen kann. In beiden Fällen gewinnt der Verteidiger.

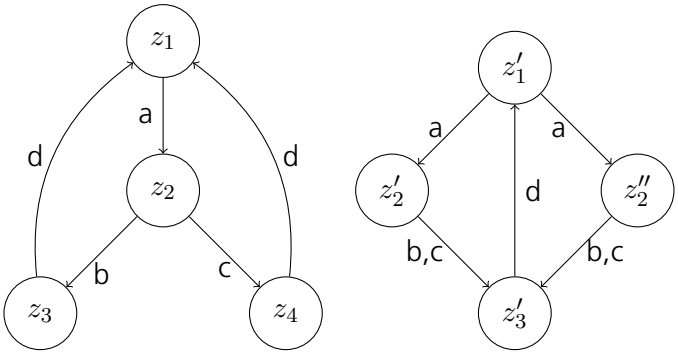
□

Selbsttestaufgabe 2.4.1

1. Demonstrieren Sie, dass der Angreifer eine Siegstrategie für das Spiel auf dem folgenden Paar von Transitionssystemen hat, wenn man das Spiel mit dem Paar (z_1, z_1') beginnt.



2. Demonstrieren Sie, dass der Verteidiger eine Siegstrategie für das Bisimulationssiel dem folgenden Paar von Transitionssystemen hat, wenn man das Spiel mit dem Paar (z_1, z_1') beginnt.



Musterlösung zu Selbsttestaufgabe 2.4.1

1. Der Angreifer kann zunächst einen Zug von z_1' mit a nach z_2' durchführen. Darauf muss der Verteidiger mit einem a-Zug von z_1 nach z_2 antworten. An-

schließend kann der Angreifer das Token wechseln und einen c-Zug von z_2 nach z_4 durchführen, auf den der Verteidiger nicht antworten kann. Der Angreifer gewinnt also das Bisimulationsspiel. Der Angreifer hat übrigens sogar eine Siegstrategie, bei der er das System nicht wechseln muss. Hierzu muss der Angreifer mit einem a-Zug von z_1 nach z_2 beginnen. Dann hat der Verteidiger allerdings für seinen Antwortzug die Wahl aus einem Zug zu z_2' oder z_2''. Abhängig von der Wahl des Verteidigers muss der Angreifer nun einen b- oder c-Zug ausführen, so dass der Verteidiger nicht antworten kann.

2. Wenn der Angreifer einen a-Zug von z_1 zu z_2 durchführt, kann der Verteidiger einen Zug von z_1' zu z_2' durchführen. So ergibt sich die neue Spielsituation (z_2, z_2'). Alternativ kann der Angreifer in der Spielsituation (z_1, z_1') einen a-Schritt zu z_2' oder z_2'' durchführen. In beiden Fällen muss der Verteidiger mit einem a-Schritt zu z_2 reagieren. Wir erhalten also die neue Spielsituation (z_2, z_2''), da die Spielsituation (z_2, z_2') bereits gefunden wurde. Weitere Zugmöglichkeiten hat der Angreifer nicht. Wir müssen also die Spielsituationen (z_2, z_2') und (z_2, z_2'') untersuchen.

   In der Spielsituation (z_2, z_2') kann der Angreifer einen b-Zug von z_2 nach z_3 durchführen. Hierauf muss der Verteidiger mit einem b-Zug von z_2' nach z_3' durchführen, so dass die Spielsituation (z_3, z_3') entsteht. Alternativ kann der Angreifer einen c-Schritt von z_2 nach z_4 durchführen. Hierauf muss der Verteidiger mit einem c-Schritt von z_2' zu z_3' reagieren und es entsteht die Spielsituation (z_4, z_3'). Der Angreifer kann aber auch von z_2' aus einen b-Schritt zu z_3' durchführen. In diesem Fall muss der Verteidiger mit einem b-Schritt von z_2 nach z_3 antworten. Die sich ergebende Spielsituation (z_3, z_3') haben wir aber bereits gefunden. Schließlich kann der Angreifer auch einen c-Schritt von z_2' nach z_3' durchführen. In diesem Fall muss der Verteidiger mit einem c-Schritt von z_2 nach z_4 antworten, was die Spielsituation (z_4, z_3') ergibt, die wir aber ebenfalls bereits gefunden haben.

   Die Spielsituation (z_2, z_2'') kann völlig analog zur Spielsituation (z_2, z_2') behandelt werden. Es bleiben die beiden Spielsituationen (z_3, z_3') und (z_4, z_3') zu betrachten.

   In der Spielsituation (z_3, z_3') kann der Angreifer einen d-Schritt von z_3 nach z_1 durchführen, worauf der Verteidiger mit einem d-Schritt von z_3' nach z_1' reagieren muss. Wir erhalten die Spielsituation (z_1, z_1'), das ist allerdings die Ausgangssituation. Alternativ kann der Angreifer einen d-Schritt von z_3' nach z_1' durchführen. In diesem Fall muss der Verteidiger mit einem d-Schritt von z_3 zu z_1 reagieren, wiederum wird die Spielsituation (z_1, z_1') erreicht. Die Spielsituation (z_4, z_3') kann völlig analog behandelt werden.

2.5 Zusammenfassung der wesentlichen Lernziele der Kurseinheit

Wenn Sie diese Lektion durchgearbeitet haben, sollten Sie die folgenden wichtigen Fragestellungen beantworten können:

- Was ist ein Transitionssystem?
- Welche Formen der Sprachäquivalenz gibt es?
- Wieso reichen Sprach-basierte Verhaltensdefinitionen nicht aus, um Transitionssysteme zu analysieren?
- Wie ist ein Verweigerungspaar ausgebaut?
- Was bedeutet es, dass zwei Zustände bisimular sind?
- Welcher Zusammenhang besteht zwischen dem Bisimulationsspiel und Bisimularität?

Sie sollten außerdem die folgenden Techniken beherrschen:

- Sie können die endliche und unendliche Sprache eine Zustandes in einem LTS angeben.
- Sie können alle oder bestimmte Verweigerungspaare eines Zustandes angeben und nachweisen oder widerlegen, ob ein mögliches Verweigerungspaar eines Zustandes tatsächlich ein Verweigerungspaar ist.
- Sie können mit dem Bisimulationsalgorithmus nachweisen, dass zwei Zustände (nicht) bisimular sind
- Sie können den Zusammenhang zwischen einer Bisimulationsrelation und der Bisimularität erläutern
- Sie beherrschen das Bisimulationsspiel und können anhand des Bisimulationsspiels argumentieren, wieso zwei Zustände (nicht) bisimular sind.

Lektion 4: Videospielentwicklung mit einer Spieleengine (Unity)

Für viele angehende Programmierer ist ein früher Motivator um sich mit der Programmierung und Informatik im Allgemeinen zu befassen, die Entwicklung eines eigenen Videospiels. Allerdings sind Videospiele typischerweise eine sehr komplexe Software, die viele algorithmische und gestalterische Herausforderungen mit sich bringen. Kontinuierliche Bewegungen von Spielfiguren, Kollisionserkennung zwischen verschiedenen Spielobjekten und ein sinnhaftes Verhalten von so genannten NPCs (Non Player Characters, zu Deutsch: Nicht-Spieler-Figuren) sind Beispiele für solche Probleme, denen man in der Entwicklung einer Vielzahl von Videospielen begegnet.

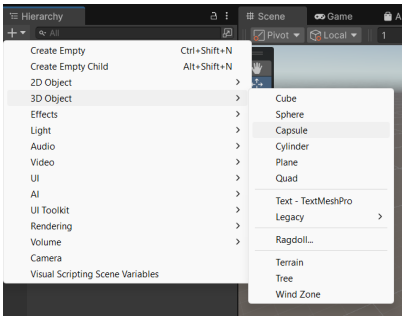
Es ist daher äußerst unüblich geworden, Videospiele von Grund auf zu entwickeln und alle diese Probleme selbst zu lösen. Im Sinne der Wiederverwendbarkeit haben sich im Verlauf der letzten Jahre und Jahrzehnte einige so genannte Videospiel-Engines herausgebildet, die in der Entwicklung von Videospielen Verwendung finden. Die drei derzeit verbreitetsten allgemeinen Videospiel-Engines, die für alle Entwickler nutzbar sind, sind Unity Engine (Marktanteil 2024 lt. Daten von VGInsights: ca. 51%), Unreal Engine (28%) und Godot (5%). Es gibt noch weitere Engines, die entweder für spezialisierte Anwendungen (z. B. GameMaker für 2D-Spiele) gedacht sind, oder aber individuelle Lösungen für den Einsatz bei einem einzelnen Spiele-Unternehmen sind (z. B. Frostbite von Electronic Arts).

Wir werden in dieser Lektion die Konzepte am Beispiel der Unity-Engine (wo relevant: am Beispiel der Version 6) vorstellen. Es gibt aber natürlich viele Gemeinsamkeiten zwischen der Unity-Engine und anderen großen Videospiele-Engines. Beispielsweise findet die wichtige `Update`-Methode der Unity-Engine ihre Entsprechung in der `tick`-Methode der Unreal-Engine und der `process`-Methode von Godot. Die Lektion kann natürlich nicht als vollständige Einführung in die Entwicklung eines Videospiels mit Unity verstanden werden und hat nicht den Anspruch, eine Unity-Anleitung darzustellen. Stattdessen sollen wichtige Konzepte und und Herangehensweisen illustriert werden, um Ihnen einen Einstiegspunkt in die Spieleentwicklung mit Unity oder einer anderen Videospiel-Engine zu bieten.

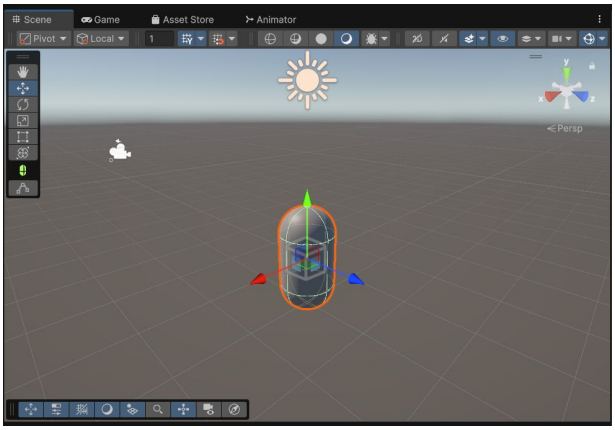
Beispiel 4.1 : Hoppz: Das laufende Beispielprojekt

Wir besprechen die Unity-Konzepte gestützt mit einem laufenden Beispiel, dem 3D Jump & Run „Hoppz“[a]. Videospiele haben natürlich eine umfassende artistische Komponente, die wir in diesem Kurs – der ja ein informatischer Kurs ist – möglich weitge-

hend ausblenden wollen. Daher werden wir die Spielwelt und die Spielfiguren möglichst einfach nur aus geometrischen Bausteinen zusammensetzen. Um uns mit dem hierarchischen Objekt-System in der Benutzeroberfläche vertraut zu machen, erzeugen wir in einem frisch erstellten Projekt zunächst unsere Hauptfigur. In diesem Beispiel verwende ich einen Hasen-Roboter auf einem Rad, um eine möglichst einfache Animation ohne Rückgriff auf Unitys Animationssystem zu ermöglichen. Hierzu arbeiten wir zunächst im Hierarchie-Fenster, das sich üblicherweise links im Fenster befindet und mit „Hierarchy“ überschrieben ist.

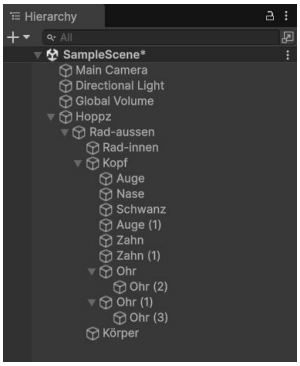


Über die Plus-Schaltfläche erzeugen wir zunächst mit „3D Object“ und „Capsule“ ein Wurzelobjekt für unsere Spielfigur, die auch ihre physische Repräsentanz im Spiel darstellt. Die erzeugte Kapsel werden wir im weiteren Verlauf unsichtbar machen, wir nutzen sie aber für die Gestaltung unserer Spielfigur als Referenz, denn die Spielfigur sollte ihre physische Repräsentanz in der Spielwelt möglichst eng ausfüllen, das heißt die optische Darstellung sollte sich nah an der Grundform orientieren, die für Kollisionserkennungen verwendet wird. Andererseits ist es dringend zu empfehlen, ein Primitiv wie Capsule zu verwenden, um die Spielfigur physisch zu repräsentieren und nicht etwa die genaue optische Form in der Kollisionsabfrage abzubilden. Der Grund hierfür ist, dass die Spielfigur anderenfalls in der Spielwelt leicht an Umgebungsgeometrie hängen bleiben kann – im Falle von Hoppz beispielsweise mit der Nase an dem Rand einer Plattform – und ihr Verhalten für Spieler intransparent sein kann.



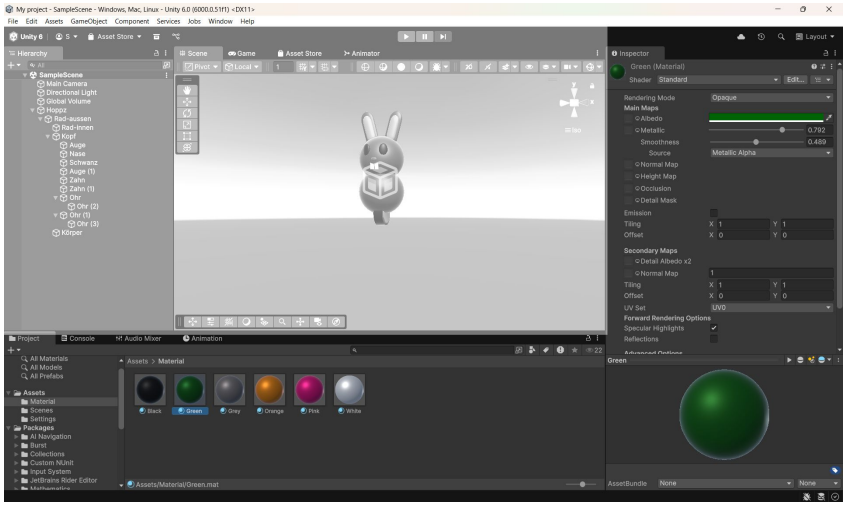
In der „Scene“-Ansicht – standardmäßig oben in der Mitte angeordnet – können wir dann die Kapsel, die wir gerade hinzugefügt haben, sehen und können sie, wenn gewünscht, mit einigen Werkzeugen anpassen. Mit dem Hand-Symbol können wir den Sichtpunkt in der 3D-Umgebung bewegen, mit dem (im Screenshot ausgewählten, blau eingefärbten) Bewegungs-Werkzeug kann das Objekt entlang der drei Achsen bewegt werden, indem man wahlweise die entsprechende farbige Achse anklickt und die Maus zieht, oder aber die Figur frei im Raum bewegen, indem man innerhalb des kleinen Sechsecks klickt – je nach gewählter farblicher Komponente wird eine Dimension hierbei festgeschrieben, da die Maus nur in zwei Dimensionen

bewegt werden kann. Unter dem Bewegungswerkzeug finden wir das Rotationswerkzeug, das eine Rotation entlang der drei Grundachsen erlaubt und darunter das Skalierungstool, mit dem die Größe angepasst werden kann.



Für unser Beispielprojekt kann beispielsweise ein einfacher Hasenroboter aus primitiven Bausteinen – Zylinder, Kugeln und Quader – zusamengestellt werden. Hierzu ist es empfehlenswert, die optische Repräsentation hierarchisch als Kindobjekte der physischen Repräsentation der Spielfigur zu erstellen, indem man mit der rechten Maustaste auf das Elterobjekt klickt und unter dem Menüpunkt „3D Object“ die gewünschte Grundform auswählt. In dem Beispiel haben wir die gesamte optische Repräsentation von Hoppz dem Hasen als Kind des Rades angelegt. Das ist eine Designentscheidung, um später einfach zumindest eine kleine Animation umzusetzen – nämlich einem Neigungswinkel für Hoppz, wenn er sich bewegt.

Um die Farben (oder später ggf. Texturen) anzupassen empfiehlt es sich, im Asset-Ordner im Project-Fenster (standardmäßig unten im Unity-Fenster) einen Ordner Material zu erstellen und in diesem Ordner für jede Farbe, die Sie verwenden wollen, per rechtem Mausklick und dem Menüpunkt „Create“, dann „Material“ ein eigenes Material zu erzeugen. Dieses Material können Sie dann im Inspector anpassen und (vorerst) beispielsweise einfach eine entsprechende Farbe einstellen. Im folgenden Bild sind Inspector- und Project-Fenster hervorgehoben und der Rest des Unity-Fensters blass gezeichnet:

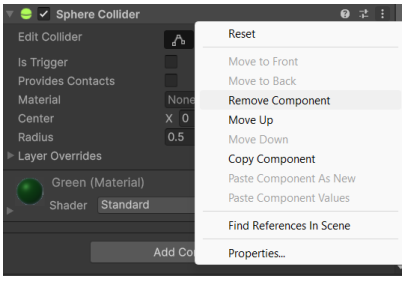


Wir werden im Zuge dieser Veranstaltung nicht ausführlich auf Materialien eingehen, aber es sei an dieser Stelle kurz erwähnt, dass die Materialien, die Unity bietet, eine Vielzahl verschiedener Erscheinungsbilder und Effekte, inklusive Reflexionen,

Transparenz und sogar Animationen von Texturen erlauben. Auch Materialien, die das Erscheinungsbild alter Spielkonsolen wie des Nintendo 64 (Vertex Lit) nachbauen, stehen zur Auswahl. Experimentieren Sie gerne ein wenig damit herum bei der Ausgestaltung Ihres Beispielprojekts. In einem kommerziellen Produkt ist die ausschließliche Verwendung einfacher Farb-Materialien selten geboten.

Sollten Sie ein Objekt versehentlich nicht direkt als Kind erzeugt haben, können Sie es einfach per Drag & Drop im Hierarchy-Fenster auf das gewünschte Elter-Objekt ziehen. Die Grafik links zeigt die Hierarchie der Objekte für die Erstellung unseres Hasen-Roboters und die Grafik rechts, wie der Hasen-Roboter in der Scene-View aussieht. Das Projekt bis zu diesem Punkt wird auf der Moodle-Seite zum Download zur Verfügung gestellt, aber ich empfehle Ihnen, eine eigene kleine Figur zu erstellen, um sich mit der Erzeugung von geometrischen Primitiven vertraut zu machen und Ihrem Projekt einen individuellen Charakter zu verleihen.

Bevor wir die rein optische Repräsentation unserer Spielfigur abschließen, sollten Sie noch einmal einen Blick auf das „Inspector"-Fenster, standardmäßig auf der rechten Seite im Unity-Fenster, werfen. Sie werden sehen, dass alle Objekte, die Sie gerade erzeugt haben, eine Komponente haben, deren Name auf „Collider" endet.



Diese Collider ermöglichen die automatische Behandlung von Kollisionen. Allerdings sollte bei unserer Spielfigur ausschließlich die Kapsel an der Wurzel Kollisionen erkennen – die optische Repräsentation der Spielfigur ist in diesem Sinne dekorativer Natur. Insofern ist es empfehlenswert, die Collider-Komponenten von allen Gestaltungselementen der Spielfigur zu entfernen. Dazu klicken Sie auf die ⋮-Schaltfläche direkt neben dem Collider-Objekt und dann auf „Remove Component". Führen Sie diese Schritte für alle Collider Ihrer optischen Repräsentation aus. Umgekehrt sollte die physische Repräsention der Spielfigur im Regelfall nicht sichtbar sein. Da es für die Fehlersuche gelegentlich dennoch hilfreich sein kann, die physische Repräsentation auch sehen zu können, würde ich empfehlen, in dem Kapsel-Objekt, das die Wurzel Ihrer Spielfigur ist, „Renderer"-Komponente nicht zu enfernen, sondern nur zu deaktivieren, dazu entfernen Sie einfach den Haken in der Checkbox neben dem Namen der Komponente.

[a] Die Figur Hoppz ist als Nebenfigur im Zuge meiner selbstständigen nebenberuflichen Tätigkeit entstanden und wird in diesem Kurstext zur Veranschaulichung verwendet. Um die Verwendung in

Studentenprojekten zu ermöglichen, wird die Figur Hoppz und alle Quellcodes und Projektdaten in diesem Kurstext unter die CC-BY Lizenz gestellt.

Bis hierhin haben wir ausschließlich in der grafischen Benutzeroberfläche von Unity gearbeitet und noch nichts programmiert. In der Konsequenz ist das „Spiel", das wir bisher entwickelt haben auch völlig ereignislos. Wenn man auf den Play-Button drückt, um das Spiel auszuprobieren, zeigt die Kamera auf unsere Spielfigur, wir können aber in keiner Weise mit dem Spiel interagieren. Um die Interaktion zu gestalten, müssen wir programmieren. Nun ist natürlich die Frage, wie man mit Unity ein Spiel programmiert, denn offenkundig arbeiten wir hier nicht mit einer `Main`-Methode als Einstieg. Stattdessen schreibt man in Unity sogenannte Skripte. Hierbei handelt es sich um C#-Klassen, die in aller Regel (direkt oder indirekt) von `MonoBehaviour` abgeleitet sind und so in das Ablaufkonzept von Unity greifen. Es ist zu empfehlen, im Projektordner einen Unterordner für Skripte anzulegen, um die Navigierbarkeit des Projekts zu erhalten. Je nach Größe des Projekts kann auch eine kontextuelle Organisationsstruktur sinnvoll sein – für erste Gehversuche in Unity hat sich die funktionale Struktur aber bewährt.

Wenn Sie ein `MonoBehaviour`-Skript erstellen [1], werden Sie bemerken, dass zwei Klassen direkt mit erzeugt werden: `Start` und `Update`. Diese beiden Methoden sind von `MonoBehaviour` geerbt, müssen aber nicht explizit überschrieben werden. Sie dienen der Unity-Laufzeitumgebung dazu, Einstiegspunkte zu wählen, um die Skripte, die sich in der Spielszene befinden, zu verwalten und auszuführen. Bevor wir die Funktion der `Start`- und `Update`-Methode besprechen, sollten wir zunächst darüber sprechen, was es bedeutet, dass sich ein Skript in einer Szene befindet.

Als Szene bezeichnen wir die Repräsentation von Levels in der Unity-Engine. Wir haben bereits mit einer Szene gearbeitet; der Inhalt einer Szene wird über das Hierarchy-Fenster bestimmt. Die Objekte in einer Szene sind im Hierarchy-Fenster hierarchisch (wir erinnern an die Strukturierung der Spielfigur) gelistet. Jedes einzelne Objekt kann angeklickt werden und dann im Inspector-Fenster bearbeitet werden. Ein Objekt kann mit mehreren verschiedenen Komponenten versehen sein. Wir haben in dieser Hinsicht bereits den Renderer und den Collider kennengelernt. Skripte können nun ebenfalls als Komponente zu einem Objekt hinzugefügt werden. Hierzu klickt man im Inspector-Fenster einfach auf „AddComponent" und wählt dann die Klasse aus, die man hinzufügen möchte. Nur was bedeutet es, dass ein Skript zu einem Objekt hinzugefügt wird? Es wird eine Instanz der Klasse erzeugt und mit dem Objekt verknüpft. Innerhalb der von `MonoBehaviour` abgeleiteten Klasse kann man dann mit dem Attribut `gameObject` auf das Gesamtobjekt zugreifen. Ein einzelnes Skript kann zu einer Vielzahl von Objekten hinzugefügt werden und grundsätzlich kann sogar einem Objekt mehr als eine Instanz eines Skripts hinzugefügt werden – wenngleich das recht fehleranfällig ist und idealerweise vermieden werden sollte.

Wenn eine Szene geladen wird, ermittelt die Unity-Laufzeitumgebung alle Instanzen von Skripten und unterscheidet drei Kategorien von (Instanzen von) Skripten:

- Skripte, die an Objekten hängen, die aktiv sind (Haken direkt neben dem Namen des Objekts im Inspektor ist gesetzt) und aktiviert sind (Haken neben dem Namen des

[1] Formal ist ein Unity-Skript eine Klasse, aber wir werden die Nomenklatur von Unity im Folgenden nutzen.

Skripts im Inspektor ist gesetzt) – wir nennen diese Skripte die aktiven (engl. active) und aktivierten Skripte (engl. enabled).

- Skripte, die nicht aktiviert sind (egal ob das Objekt aktiv ist oder nicht) – wir nennen diese Skripte deaktiviert (engl. disabled).

- Skripte, die an Objekten hängen, die nicht aktiv sind (egal ob die Skripte aktiviert sind oder nicht) – wir nennen diese Skripte inaktiv (engl. inactive).

Das Verhalten von Skripten wird von Unity automatisch über folgende zeitlich gesteuerte Methoden bestimmt:

- `Awake` – wird bei der Erzeugung des Objektes, an dem das Skript hängt, ausgeführt – auch dann, wenn das Skript nicht aktiviert ist.

- `Start` – wird beim erstmaligen Aktivieren eines Skripts (an einem aktiven Objekt) ausgeführt.

- `Update` – wird bei aktiven und aktivierten Skripten einmal pro berechnetem Bild („Frame“) ausgeführt.

- `LateUpdate` – wird bei aktiven und aktivierten Skripten einmal pro berechnetem Bild ausgeführt – aber nachdem alle `Update`-Methoden abgearbeitet sind.

- `FixedUpdate` – wird bei aktiven und aktivierten Skripten einmal pro festgelegter Zeiteinheit (standardmäßig: 0,02 Sekunden) ausgeführt.

Es ist insbesondere zu beachten, dass zunächst alle `Awake`-Methoden ausgeführt werden, bevor eine `Start`-Methode ausgführt wird. Bei der Konfiguration der Startbedingungen von Skripten ist es sinnvoll, den Teil der Konfiguration, der sich auf das Objekt selbst beziehen, in der `Awake`-Methode vorzunehmen und verknüpfende Konfigurationsteinstellungen, die ggf. den Zugriff auf andere Objekte bedarf, in `Start` durchzuführen.

Unser erstes Ziel ist es nun, unsere Spielfigur mit einem Analogstick oder den Pfeiltasten zu steuern. Hierzu verwenden wir die `Input`-Funktionalität von Unity, die in jedem Update-Zyklus die Nutzereingaben aktualisiert und ermöglicht, sie abzufragen. Für analoge Eingabemöglichkeiten (wie Analogsticks und Trigger) kann ein `float`-Wert abgefragt werden, wohingegen digitale Eingabemöglichkeiten (d.h. Knöpfe auf einem Controller und Tasten auf der Tastatur) drei Zustände kennen:

- `GetButtonDown`: Diese Methode gibt `true` zurück, wenn der übergebene Button in dem Frame gedrückt wurde, das heißt, er war im vorherigen Frame nicht gedrückt und wurde im aktuellen Thread neu betätigt.

- `GetButton`: Diese Methode gibt `true` zurück, wenn der übergebene Button in dem Frame gehalten wurde, das heißt, er würde in diesem oder einem vorherigen Frame gedrückt und seitdem nicht wieder losgelassen.

- `GetButtonUp`: Diese Methode gibt `true` zurück, wenn der übergebene Button in dem Frame losgelassen wurde. Im vorherigen Frame wurde der Knopf also noch heruntergedrückt (gehalten), jetzt nicht mehr.

Wir wollen unsere Erkenntnisse nutzen, um unser laufendes Beispiel mit einem ersten Ansatz zur Interaktion zu erweitern.

Beispiel 4.2 : Erste Steuerungsversuche

Wir erzeugen ein neues `MonoBehavior`-Skript, um das Verhalten von Hoppz zu definieren. In diesem Skript fragen wie in `Update` die Steuerungseingaben ab und bewegen Hoppz gemäß der Nutzereingaben über den Boden. Unser erster Ansatz könnte wie folgt aussehen:

```
public class Hoppz : MonoBehaviour
{
    float speed = 1f;
    // Update is called once per frame
    void Update()
    {
        float h = Input.GetAxis("Horizontal");
        if (Input.GetKey(KeyCode.LeftArrow)) h = -1;
        if (Input.GetKey(KeyCode.RightArrow)) h = 1;
        float v = Input.GetAxis("Vertical");
        if (Input.GetKey(KeyCode.DownArrow)) v = -1;
        if (Input.GetKey(KeyCode.UpArrow)) v = 1;
        transform.position = transform.position
          + new Vector3(h, 0, v) * Time.deltaTime * speed;
    }
}
```

Wir können dieses Skript unserer Spielfigur (in der Wurzel) als Komponente hinzufügen, anschließend kann Hoppz mit dem linken Analogstick eines Controllers oder aber mit den Pfeiltasten über einen imaginären Boden gesteuert werden. Wir nutzen für die Verarbeitung der Eingaben die `Input`-Klasse, über die wir mit `GetAxis` respektive `GetKey` den Zustand des Analogsticks respektive der Pfeiltasten abfragen. Alle Informationen zu den räumlichen Eigenschaften unseres Objekts können wir über die (geerbte) Variable `transform` ansprechen. In unserem Fall wollen wir die Position im Raum gemäß der Eingabe anpassen. Da die Bildwiederholrate (engl. Framerate) variabel ist, würde es aber zu erratischem Verhalten kommen, wenn wir die Spielfigur einfach um einen konstanten `speed`-Faktor verschieben würden, so dass wir zur Skalierung noch die Methode `Time.deltaTime` verwenden – das ist die Zeit, die für die Berechnung des letzten Frames aufgewendet wurde.

Diese Implementation ist aber natürlich in vielerlei Hinsicht nicht zufriedenstellend:

- Die Spielfigur bewegt sich völlig unabhängig von der (statischen) Kamera und kann somit aus dem dargestellten Bild herauslaufen
- Die Spielfigur hat keinerlei Kollision: Man kann die Spielfigur einfach durch andere geometrische Objekte hindurchbewegen. Probieren Sie das einmal aus, indem Sie beispielsweise einen Quader in der Szene platzieren.
- Die Spielfigur rotiert nicht mit der Eingabe.
- Die Spielfigur kennt keine Gravitation, sie schwebt einfach in der Luft.
- Die Bewegung ist völlig unbeschleunigt. Das kann je nach Spiel gewünscht sein, die meisten Spiele setzen aber auf eine beschleunigte Bewegung.

Diese Probleme werden wir nach und nach angehen.

Bevor wir die konkreten Probleme angehen, beobachten wir, dass der Ansatz, die Steuerung direkt über Unitys `Input`-Klasse abzufragen, zu einer schlechten Wartbarkeit führt: Wenn wir verschiedene Eingabemöglichkeiten auf die gleiche Aktion abbilden wollen, müssen wir die entsprechende Disjunktion an jeder Stelle durchführen, und wenn wir eine Belegung ändern wollen, müssen wir ggf. über eine ganze Reihe von Skripten iterieren und Modifikationen durchführen. Daher ist es empfehlenswert, solche Schnittstellen-Funktionen zur Engine in eine Klasse zu kapseln, die die Anfragen an die Schnittstelle Unitys zusammenfassen und somit an einem Ort bündeln. Hierzu verwenden wir ein Entwurfsmuster, das auch in anderen Programmierkontexten eine Rolle spielt: Das Singleton-Pattern:

Definition 4.3 : Singleton-Pattern

Als Singleton bezeichnet man eine Klasse, die sicherstellt, dass es zu jedem Zeitpunkt nur (höchstens) eine Instanz der Klasse gibt. Zudem ermöglicht eine Singleton-Klasse den Zugriff auf die (einzige) Instanz ihrer Klasse über ein statisches Attribut. Prototypisch ist eine Singleton-Klasse in Unity wie folgt zu definieren:

```
class A
{
  public static A instance;
  Awake()
  {
    if(instance!=this) Destroy(this); else instance = this;
  }
  ...
}
```

Das Singleton-Pattern findet auch außerhalb der Videospielentwicklung vielerlei Anwendung. In unserem Fall kann man das Pattern wie folgt für die Kapselung der Steuerungs-

eingaben verwenden:

```
public class GamePadScript : MonoBehaviour
{
  public static GamePadScript instance;
  void Awake()
  {
    if (instance != this && instance != null) Destroy(this);
    else instance = this;
  }
  public float xAxis()
  {
    float h = Input.GetAxis("Horizontal");
    if (Input.GetKey(KeyCode.LeftArrow)) h = -1;
    if (Input.GetKey(KeyCode.RightArrow)) h = 1;
    return h;
  }
  public float yAxis()
  {
    float v = Input.GetAxis("Vertical");
    if (Input.GetKey(KeyCode.DownArrow)) v = -1;
    if (Input.GetKey(KeyCode.UpArrow)) v = 1;
    return v;
  }
  public float left()
  {
    if (xAxis() < 0) return -xAxis();
    return 0;
  }
  public float right()
  {
    if (xAxis() > 0) return xAxis();
    return 0;
  }
  public float up()
  {
    if (yAxis() > 0) return yAxis();
    return 0;
  }
  public float down()
  {
    if (yAxis()< 0) return -yAxis();
    return 0;
  }
```

```
  public bool jumpButtonDown()
  {
    return Input.GetKeyDown(KeyCode.Space)
      || Input.GetButtonDown("Jump");
  }
  public bool jumpButtonUp()
  {
    return Input.GetKeyUp(KeyCode.Space)
      || Input.GetButtonUp("Jump");
  }
  public bool jumpButton()
  {
    return Input.GetKey(KeyCode.Space)
      || Input.GetButton("Jump");
  }
}
```

Das Skript sollte im Verlauf der Entwicklung um weitere Funktionalitäten für weitere Eingaben erweitert werden; das werden wir im Kursmaterial nicht explizieren, aber das Schema, nach dem die Methoden umgesetzt werden können, sollte ersichtlich sein. Wichtig an der Stelle ist noch, dass bei der Abfrage von Buttons von Controllern über ihren Namen die Namen genutzt werden müssen, die im Unity-Editor unter dem Menüpunkt Edit – Project Settings – Input Manager konfiguriert sind. Bei Bedarf können die Button-Namen hier auch geändert oder zusätzliche Button-Namen definiert werden.

4.1 Orientierung und Rotation in Unity

Beim Erstellen der Repräsentation unserer Spielfigur in der Spielwelt haben wir bereits im Inspector Rotationen für 3D-Objekte im Raum modifiziert. Wenn wir die Rotation der gesamten Spielfigur an die Eingaben des Spielers anpassen möchte, kann man aber nicht über den Inspector arbeiten, sondern muss die Rotation in einem Skript über den Zugriff auf das Attribut `transform.rotation` modifizieren. Betrachtet man den Datentyp für die Rotation, wird man aber zunächst womöglich überrascht, denn anders als im Inspector wird die Rotation nicht über die Angabe dreier Rotationswinkel angegeben, sondern über so genannte Quaternionen. Wir werden im Folgenden besprechen, was Quaternionen sind und wieso diese für die Orientierung von Spielobjekten im Raum verwendet werden, anstatt mit drei Rotationswinkeln wie im Inspector zu arbeiten.

Definition 4.1.1 : Quaternionen

Ein Quaternion ist ein Viertupel $(x, y, z, w) \in \mathbb{R}^4$. Die vier Komponenten von Quaternionen werden, ähnlich wie Real- und Imaginärteil bei komplexen Zahlen, mittels imaginärer Einheiten interpretiert. Das heißt, ein Quaternion kann geschrieben wer-

den als

$$x + y \cdot i + z \cdot j + w \cdot k,$$

dabei sind i, j, k die imaginären Einheiten, die den folgenden Rechenregeln genügen:

$$i \cdot i = -1 \quad j \cdot j = -1 \quad k \cdot k = -1 \quad i \cdot j \cdot k = -1$$

Quaternionen bilden einen Schiefkörper mit den folgenden Operationen:

- *Addition:* Zwei Quaternionen $x = a + bi + cj + dk$, $y = e + fi + gj + hk$ werden *komponentenweise* addiert, d. h.

$$x + y = (a + e) + (b + f)i + (c + g)j + (d + h)k$$

- *Multiplikation:* Zwei Quaternionen $x = a + bi + cj + dk$, $y = e + fi + gj + hk$ werden multipliziert wie folgt:

$$\begin{aligned} x \cdot y &= (a + bi + cj + dk) \cdot (e + fi + gj + hk) \\ &= (ae - bf - cg - dh) + (af + be + ch - dg)i \\ &\quad + (ag + bh + ce - df)j + (ah + bg - cf + de)k \end{aligned}$$

Zur Erinnerung: Dass Quaternionen Schiefkörper sind, bedeutet, dass sie alle Eigenschaften eines Körpers haben, mit Ausnahme der Kommutativität der Multiplikation. Für zwei Quaternionen x, y gilt also nicht notwendigerweise $x \cdot y = y \cdot x$.

Selbsttestaufgabe 4.1.1

Wir machen uns mit den Rechenregeln für Quaternionen vertraut.

1. Zeigen Sie für alle Paare von imaginären Einheiten i, j, k, welchen Wert ihr Produkt hat.
2. Verifizieren Sie die Rechenregel für die Multiplikation von Quaternionen.
3. Zeigen Sie an einem Beispiel, dass die Multiplikation von Quaternionen nicht kommutativ ist.

Sie dürfen ohne Beweis verwenden:

- Die Multiplikation von Quaternionen und reellen Zahlen ist kommutativ, also wenn q ein Quaternion und x eine reelle Zahl ist, dann ist $q \cdot x = x \cdot q$.
- Das multiplikative Inverse von links und rechts sind bei allen Quaternionen identisch. Also wenn $a \cdot b = 1$, dann ist auch $b \cdot a = 1$.

Musterlösung zu Selbsttestaufgabe 4.1.1

1. **ij**: $ijk = -1 \Leftrightarrow ijkk = -k \Leftrightarrow -ij = -k \Leftrightarrow ij = k$

   jk: $ijk = -1 \Leftrightarrow iijk = -i \Leftrightarrow -jk = -i \Leftrightarrow jk = i$

   ji: $ijk = -1 \Leftrightarrow jiijk = -ji \Leftrightarrow -jjk = -ji \Leftrightarrow -k = ji$

   kj: $ijk = -1 \Leftrightarrow ijkkj = -kj \Leftrightarrow -ijj = -kj \Leftrightarrow i = -kj \Leftrightarrow kj = -i$

   ki: $ijk = -1 \Leftrightarrow ijki = -i \Leftrightarrow iijki = -ii \Leftrightarrow -jki = 1 \Leftrightarrow ki = j$

   ik: $ijk = -1 \Leftrightarrow ijiijk = -iji \Leftrightarrow -ijjk = -iji \Leftrightarrow ik = -iji \Leftrightarrow ik = -ki = -j$

   Alternativ kann man etwas einfacher auch argumentieren, indem man multiplikative Inverse verwendet.

2. $$\begin{aligned}
   &(a + bi + cj + dk) \cdot (e + fi + gj + hk) \\
   =&ae + afi + agj + ahk + bei + bfii + bgij + bhik + cej + cfji + cgjj \\
   &+ chjk + dek + dfki + dgkj + dhkk \\
   =&ae + afi + agj + ahk + bei - bf + bgk - bhj + cej - cfk - cg \\
   &+ chi + dek + dfj - dgi - dh \\
   =&ae - bf - cg - dh + afi + bei + chi - dgi \\
   &+ agj - bhj + cej + dfj + ahk + bgk - cfk + dek \\
   =&(ae - bf - cg - dh) + (af + be + ch - dg)i \\
   &+ (ag - bh + ce + df)j + (ah + bg - cf + de)k
   \end{aligned}$$

3. Bei der Ermittlung der Rechenregeln haben wir bereits gezeigt: $ik = -ki$ und wir wissen auch, dass $ik \neq 0$ ist (der einzige Fall, in dem $ik = -ki$ mit Kommutativität kompatibel ist), denn $ik = j$ und $j \cdot j = -1 \neq 0$.

Nun stellt sich natürlich die Frage, was der Zusammenhang zwischen dem Schiefkörper der Quaternionen und Rotationen im $\mathbb{R}^3$ ist. Um das verstehen zu können, brauchen wir noch zwei Begriffe, die Ihnen von komplexen Zahlen und Vektorrechnung her bekannt sein dürften:

Definition 4.1.2 : Normierung und Konjugation

Gegeben sei ein Quaternion $q = a + bi + cj + dk$, dann ist

- Die Norm von q definiert als

$$|q| = \sqrt{a^2 + b^2 + c^2 + d^2}$$

- Der normierte (in Englisch und in Unity: normalized) Vektor zu q ist definiert als $\frac{q}{|q|}$ – der normierte Vektor zu q hat im $\mathbb{R}^4$ die gleiche Richtung wie q, aber die Länge (Norm) 1.
- Die Konjugierte zu q ist definiert als $\overline{q} = a - bi - cj - dk$

Normierte Quaternionen können schließlich verwendet werden, um Rotationen eines Vektors des $\mathbb{R}^3$ um beliebige Achsen im Raum zu realisieren. Hierzu wird eine Transformation genutzt, die unter dem Namen Quaternion-Sandwich-Formel bekannt ist – Sandwich deswegen, weil der in den $\mathbb{R}^4$ eingebettete zu rotierende Vektor von links und rechts mit einem Quaternion multipliziert wird.

Lemma 4.1.3

Gegeben sei ein (nicht notwendigerweise normierter) Vektor $v \in \mathbb{R}^3$ und eine (normierte) Rotationsachse $u \in \mathbb{R}^3$. Weiterhin sei ein Winkel α gegeben. Die Rotation von v im Winkel von α um die Achse u kann mithilfe der Quaternion-Sandwich-Formel berechnet werden. Hierzu wird $v = (v_1, v_2, v_3)$ in den Raum der Quaternionen wie folgt eingebettet:

$$v_q = (0, v_1, v_2, v_3)$$

und wir bilden den (normierten) Roationsquaternion

$$q = (\cos(\alpha/2), \sin(\alpha/2) \cdot u_1, \sin(\alpha/2) \cdot u_2, \sin(\alpha/2) \cdot u_3)$$

Dann berechnen wir

$$v'_q = q \cdot v_q \cdot \overline{q}.$$

Es sei noch einmal daran erinnert, dass $\overline{q}$ das konjugierte Quaternion von q ist. Der Vektor $v' = ((v'_q)_2, (v'_q)_3, (v'_q)_4)$ ist dann der gesuchte rotierte Vektor.

Die Intuition hinter der Rotation mit Quaternionen lässt sich aus der Polarkoordinaten-Schreibweise für komplexe Zahlen ableiten. Komplexe Zahlen können – neben ihrer Darstellung als Real- und Imaginärteil als $a + bi$ – mittels Polarkoordinaren dargestellt werden. Hierfür bedarf es auch zweier Datenpunkte, den Betrag der Zahl r und den Winkel φ in der Ebene zwischen der Realteil-Achse und der Imaginärteil-Achse, wir schreiben dann die Zahl als $r \cdot e^{i \cdot \varphi}$. Aus dieser Darstellung kann man bereits ersehen: Wenn man mit normierten komplexen Zahlen arbeitet, also solchen, bei denen $r = 1$ ist, dann entspricht die Multiplikation von komplexen Zahlen der Rotation in der Fläche.

Beispiel 4.1.4

Wir betrachten den Vektor $v = (1, 2, 3)$ im $\mathbb{R}^3$ und wollen ihn um die y-Achse (im Unity-Koordinatensystem) rotieren und zwar um 90. Die normierte Rotationsachse ist

also $u = (0, 1, 0)$. Es ergibt sich der Rotationsquaternion

$$q = (\cos(45), \sin(45) \cdot 0, \sin(45) \cdot 1, \sin(45) \cdot 0) = (\cos(45), 0, \sin(45), 0)$$

Damit ist zu berechnen:

$$v'_q = q \cdot v_q \cdot \overline{q} = (\cos(45) + \sin(45)j) \cdot (i + 2j + 3k) \cdot (\cos(45) - \sin(45)j)$$

Wir verwenden unsere Formel für die Multiplikation für Quaternionen und erhalten:

$$\begin{aligned}
v'_q =& q \cdot v_q \cdot \overline{q} = (-2\sin(45) + (\cos(45) + 3\sin(45))i \\
&+ (2\cos(45))j + (3\cos(45) - \sin(45))k) \cdot (\cos(45) - \sin(45)j) \\
=& -2\sin(45)\cos(45) + 2\cos(45)\sin(45) \\
&+ (\cos(45) \cdot \cos(45) + 3\sin(45)\cos(45) - \sin(45) \cdot \sin(45) + 3\cos(45)\sin(45))i \\
&+ (2\sin(45)\sin(45) + 2\cos(45)\cos(45))j \\
&+ (\cos(45) + 3\sin(45))(-\sin(45)) + (3\cos(45) - \sin(45))\cos(45)k \\
=& 0 + (0,5 + 1,5 - 0,5 + 1,5)i + (1 + 1)j + (-0,5 - 1,5 + 1,5 - 0,5)k \\
=& 0 + 3i + 2j - 1k
\end{aligned}$$

Wenn wir nun den Vektor wieder extrahieren, erhalten wir den korrekt rotierten Vektor: $v' = (3, 2, -1)$ – dass der Vektor korrekt rotiert wurde, kann einfach von Hand verifiziert werden.

Aus den einführenden Mathematik-Kursen kennen Sie vermutlich die Rotationsmatrizen als alternative Möglichkeit, Rotationen darzustellen. Unity verwendet Quaternionen auf Grund einiger Vorteile, die sich gerade in Spiele-Anwendungen (aber auch Simulationsanwendungen) ergeben:

- Einfache Interpolation zwischen zwei Rotationen: Gerade wenn man Rotationen in Animationen verwendet, ist es eine wichtige Operation, zwischen einer Start- und einer Zielrotation zu interpolieren und die Rotationen in regelmäßigen Abständen darzustellen, die sich zwischen Start- und Zielrotation ergeben. Das ist mit Quaternionen einfacher zu realisieren als mit Rotationsmatrizen.
- Kompaktere Darstellung: Eine Rotationsmatrix im Raum hat insgesamt neun Einträge – es handelt sich um eine 3×3-Matrix. Ein Quaternion hingegen hat nur vier Einträge und verbraucht somit weniger Speicher und in Rechnungen weniger Operationen.
- Größere numerische Stabilität der Operationen.

Es ist in Unity allerdings im Regelfall nicht notwendig, über Rotationen in der Welt der Quaternionen nachzudenken, da die Klasse der Quaternionen in Unity eine Vielzahl von Methoden anbietet, um Rotationen aus Euler-Winkeln zu konstruieren. Immer dann, wenn es nicht notwendig ist, Quaternionen in Euler-Winkel zu transformieren und insbesondere wenn der Übergang zwischen zwei Rotationen stabil interpoliert werden soll, ist es sinnvoll, im Raum der Quaternionen zu arbeiten.

Mit dieser Vorbereitung können wir die Rotation der Spielfigur an die angegebene Bewegungsrichtung anpassen:

```
void Update()
{
    float h = GamePadScript.instance.xAxis();
    float v = GamePadScript.instance.yAxis();
    Vector3 newPosition = transform.position
        + new Vector3(h, 0, v) * Time.deltaTime * speed;
    transform.LookAt(newPosition);
    transform.position = newPosition;
}
```

Nun schaut die Figur immer in die Richtung, in die man sie bewegt, allerdings kann die Spielfigur schnell aus dem sichtbaren Ausschnitt des Spiels verschwinden, wenn man sie nach links, rechts, oder in Richtung der Kamera bewegt. Weniger drastisch, aber ebenfalls nicht unbeschränkt praktikabel ist, dass die Spielfigur bei der Bewegungsrichtung weg von der Kamera immer kleiner wird. Sofern das Spiel nicht auf einem sehr beschränkten Spielfeld, wie beispielsweise bei dem Spielhallen-Klassiker Pac-Man, spielen soll, ist es notwendig, die Kamera mitzubewegen. Die Kamera wird beim Erzeugen einer neuen Spielszene von Unity direkt mit angelegt und so findet sich auch in unserem Projekt eine Kamera, die wir mit einem entsprechenden Skript, in unserem Fall `CameraScript` genannt, versehen können, das die Kamerabewegung steuert.

Wie man eine Kamera in einem 3D-Spiel umsetzt, ist eine individuelle Designentscheidung. Die einfachste Lösung, die für viele Spieltypen einigermaßen gut funktioniert, ist, die Kamera als Kindobjekt der Spielfigur einzustellen und die Kamera in einer festen Position relativ zur Spielfigur zu fixieren. Das ist beispielsweise für Shooter-Spiele und Rennspiele eine praktikable Option, Hoppz soll aber ein 3D Jump & Run werden und hierbei ist eine solche feste Kameraperspektive zwar nicht gänzlich ausgeschlossen, aber doch ungewöhnlich. Da es in einem solchen Spiel essenziell ist, sich in der dreidimensionalen Welt zu orientieren und eine gute Kameraperspektive für unterschiedliche Sprungpassagen zu erhalten, ist eine autonom agierende Kamera wünschenswert. Zudem kann es, je nach Designansatz, auch erstrebenswert sein, dass die Kamera eine gewisse Stabilität bietet, sich also nicht notwendigerweise bei jeder Bewegung der Spielfigur mitbewegt, sondern die Spielfigur in einem geeigneten Rahmen im Bild behält.

Hierzu erhält das `CameraScript` einen Referenzpunkt vom Typ `Vector3` – also einen Punkt im Raum – den die Kamera fokussiert. Dieser Referenzpunkt platziert einen konzeptuellen Zylinder im Raum, innerhalb dessen sich Hoppz frei bewegen kann, ohne, dass die Kamera ihre Position ändert. Wenn Hoppz über den Rand des Zylinders hinausgeht, zieht der Referenzpunkt allerdings soweit nach, dass Hoppz sich wieder innerhalb des Zylinders befindet. Die folgende Grafik demonstriert das Konzept:

Innerhalb des Zylinders kann Hoppz sich frei bewegen

Überschreitet Hoppz den Rand des Zylinders, zieht der Zylinder (im selben Frame) so weit nach, dass Hoppz wieder im Zylinder ist.

Die Implementation der Kamera bis hierhin kann beispielsweise so aussehen:

```
public class CameraScript : MonoBehaviour
{
  // Referenzpunkt, den die Kamera fokussiert
  public Vector3 referencePoint;
  // Quadrierter Radius des Zylinders
  public float maxXZSqrDistance;
  // Radius des Zylinders
  float maxXZDistance;
  // Höhe des Zylinders
  public float maxYDistance;
  private void Start()
  {
    // Zu Beginn ist Hoppz Position der Referenzpunkt
    referencePoint = Hoppz.instance.transform.position;
    maxXZDistance = Mathf.Sqrt(maxXZSqrDistance);
  }

  void LateUpdate()
  {
    // Buffern von Hoppz Position und Strecke zum Referenzpunkt
    Vector3 userPosition = Hoppz.instance.transform.position;
    Vector3 userToReference = referencePoint-userPosition;
    // Koordinaten des neuen Referenzpunktes
    float x = referencePoint.x;
    float y = referencePoint.y;
    float z = referencePoint.z;
    // Korrektur der y-Koordinate (falls nötig)
    if (Mathf.Abs(userToReference.y) > maxYDistance)
      y = Mathf.Sign(userToReference.y) * maxYDistance
        + userPosition.y;
    // Korrektur von x- und z-Koordinate (falls nötig)
    if(userToReference.x*userToReference.x
      + userToReference.z*userToReference.z > maxXZSqrDistance)
    {
      Vector3 referenceXZ = new Vector2(userToReference.x,
        userToReference.z).normalized * maxXZDistance;
      x = userPosition.x + referenceXZ.x;
      z = userPosition.z + referenceXZ.y;
    }
    Vector3 newReference = new Vector3(x, y, z);
    // Setzen der Position der Kamera und des Referenzpunkts
    transform.position += newReference - referencePoint;
```

```
    referencePoint = newReference;
    transform.LookAt(referencePoint);
  }
}
```

Wir verwenden hier `LateUpdate` um sicherzustellen, dass die Kamera gegenüber Hoppz nicht einen Frame hinterher läuft. Je nach Spielkonzept ist diese Implementation einer Kamera bereits ausreichend, aber für ein 3D Jump & Run wie Hoppz kann Spielerkontrolle über die Kamera-Rotation und Zoom wünschenswert sein. Hierzu wird in der Regel der rechte Analogstick auf dem Controller verwendet. Wir wollen zwei verschiedene Interaktionsformen für die Kamera vorsehen. Wenn man den rechten Stick entlang der horizontalen Achse bewegt, soll die Kamera um den Referenzpunkt herum rotieren, wird der Stick entlang der vertikalen Achse bewegt, soll die Kamera herauszoomen und höher positioniert werden respektive hereinzoomen und niedriger positioniert werden. Die seitliche Rotation um den Referenzpunkt kann durch eine einfache Ergänzung des obigen Codes (einzufügen vor `transform.LookAt(referencePoint)`) umgesetzt werden; die hier genutzte `RotateAround`-Funktion ist eine wichtige Grundfunktionalität der Quaternionen-Implementation in Unity und sollte in jedem Fall gegenüber einer manuellen Rotationsberechnung bevorzugt werden:

```
if (GamePadScript.instance.xAxisC() != 0)
{
  transform.RotateAround(referencePoint, Vector3.up,
    GamePadScript.instance.xAxisC() * speed);
}
```

Für die Kamerasteueurng entlang der verikalen Achse verwalten wir eine neue `float`-Variable `cameraZoomPercentage`, die zwischen den extremalen Zoom- und Rotationswerten vermittelt. Da wir in diesem Fall, anders als bei der Rotation um die horizontale Achse, keinen Kreis beschreiben, sondern entlang einer Kurve die Kamera nach oben und unten bewegen, benötigen wir diese Extremalwerte; die eigentliche Zoom-Berechnung lagern wir in eine separate Methode aus:

```
if (GamePadScript.instance.yAxisC()!= 0)
{
  cameraZoomPercentage -= GamePadScript.instance.yAxisC()
    * speed / 100;
  if (cameraZoomPercentage > 1) cameraZoomPercentage = 1;
  if (cameraZoomPercentage < 0) cameraZoomPercentage = 0;
  updateZoom();
}
```

Um nun die Methode `updateZoom` zu implementieren, müssen wir die Strecke zwischen Referenzpunkt und Kamera neu festlegen. Hierzu müssen wir zunächst die Rotation entlang der senkrechten Achse festschreiben, neu zu berechnen sind die Länge der Strecke, die weiterhin im Referenzpunkt beginnt, sowie der Winkel zur xz-Ebene. Entscheidend ist

hierbei das Kreuzprodukt, das es uns erlaubt, den Vektor zu errechnen, der sowohl senkrecht auf der senkrechten Achse (y-Achse) als auch auf der Projektion der Verbindungsstrecke zwischen Kamera und Referenzpunkt auf die xz-Ebene steht. Das erlaubt uns dann, die projizierte Verbindungsstrecke mit dem gewünschen Winkel nach oben zu rotieren. Es ergibt sich die folgende Implementation:

```
void updateZoom()
{
  // Ermitteln der Distanz und des Winkels basierend auf
  // den Extremalwerten
  cameraDistance = cameraZoomPercentage * cameraDistanceMax
    + (1 - cameraZoomPercentage) * cameraDistanceMin;
  cameraAngle = (1 - cameraZoomPercentage) * cameraAngleMax
    + cameraZoomPercentage * cameraAngleMin;
  // Projektion der Verbindungsstrecke zwischen Kamera und
  // Referenzpunkt auf die xy-Ebene
  Vector3 cameraToReference = referencePoint-transform.position;
  cameraToReference.y = 0;
  // Ermittlung der Rotationsachse mit dem Kreuzprodukt
  Vector3 horizontal = cameraToReference.normalized;
  Vector3 axis = Vector3.Cross(horizontal, Vector3.up);
  // Rotation der projizierten Verbindungsstrecke nach oben um
  // den gewünschten Winkel
  Vector3 tiltedDirection =
    (Quaternion.AngleAxis(cameraAngle, axis) * horizontal);
  // Festlegen der Kameraposition gemäß der gewünschten neuen
  // Distanz zwischen Kamera und Referenzpunkt
  transform.position = referencePoint
    - tiltedDirection * cameraDistance;
}
```

Zum Abschluss des Abschnitts über Rotationen in Unity betrachten wir, dass die Steuerung der Spielfigur in der derzeitigen Form von der Kamera-Rotation losgelöst ist: Sobald man die Kamera dreht, fährt Hoppz nicht mehr von der Kamera weg, wenn man den linken Analogstick nach oben drückt. Das kann desorientierend sein und ist der Spielbarkeit nicht zuträglich. Es gibt zwei gängige Lösungen – entweder die Steuerung wird abhängig von der Rotation der Spielfigur gemacht, drückt man nach oben, läuft die Figur aus ihrer Perspektive also nach oben, wohingegen ein Druck nach links oder rechts die Figur in die jeweilige Richtung rotiert. Diese Steuerung wird beispielsweise in Horror-Spielen traditionell verwendet, aber auch in einzelnen 3D Jump & Runs wurde auf dieses Konzept zurückgegriffen. Die Alterantiv ist, die Steuerung relativ zur Kameraperspektive zu interpretieren.

Selbsttestaufgabe 4.1.2

Wählen Sie eine der beiden Varianten für die Steuerung des Charakters aus – entsprechend Ihrer persönlichen Präferenz – und modifizieren Sie die `Update`-Methode der Klasse Hoppz entsprechend.

Musterlösung zu Selbsttestaufgabe 4.1.2

```
void Update()
{
  float h = GamePadScript.instance.xAxis();
  float v = GamePadScript.instance.yAxis();
  // Überführe Steuerungsrichtung in den Kamera–Raum:
  Vector3 movementVector
    = CameraScript.instance.transform.TransformVector
      (new Vector3(h, 0, v));
  movementVector.y = 0;
  movementVector.Normalize();
  Vector3 newPosition = transform.position
    + movementVector * Time.deltaTime * speed;
  transform.LookAt(newPosition);
  transform.position = newPosition;
}
```

Selbsttestaufgabe 4.1.3

Derzeit gibt Hoppz kein visuelles Feedback, wenn er bewegt wird. Implementieren Sie eine Methode, die Hoppz nach vorn neigt, wenn er fährt.

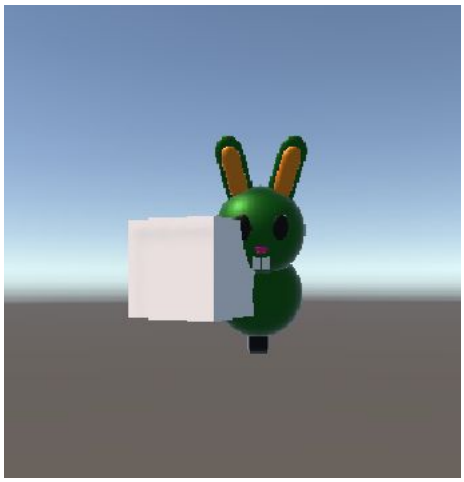
Musterlösung zu Selbsttestaufgabe 4.1.3

```
void Update()
{
  if (Time.timeScale == 0) return;
  if (currentInclination < 20 && (GamePadScript.instance
      .xAxis() != 0 || GamePadScript.instance.yAxis() != 0))
    currentInclination+=2;
  if (currentInclination > 0 && GamePadScript.instance.
      xAxis() == 0 && GamePadScript.instance.yAxis() == 0)
    currentInclination-=2;
  Vector3 currentRot = transform.localRotation.eulerAngles;
```

```
    currentRot.x = -currentInclination;
    transform.localRotation = Quaternion.Euler(currentRot);
}
```

4.2 Algorithmen zur Kollisionserkennung

Wir können Hoppz nun durch die (leere) Spielwelt bewegen und die Kamera justieren, doch ergibt sich hieraus mitnichten ein interessantes Spiel. Hierzu bräuchte es eine interessante Spielwelt und auch noch mehr Interaktionsformen für Hoppz. Um eine Spielwelt für Hoppz zu schaffen, können wir – da die Einführung in die Verwendung von 3D-Modellierungssoftware den Rahmen dieser Lektion sprengen würde – auf geometrische Grundformen zurückgreifen, wie wir das bereits bei der Gestaltung von Hoppz' Repräsentation getan haben. Wir fügen also beispielhaft einen Würfel in die Spielwelt ein und platzieren ihn vor Hoppz. Wenn man das Spiel danach startet, wird man aber schnell feststellen: Hoppz kümmert sich um den Würfel herzlich wenig und kann einfach durch den Würfel hindurchfahren:



Unser Spiel muss also erkennen, ob Hoppz mit Levelgeometrie kollidiert und idealerweise verhindern, dass Hoppz durch die Geometrie hindurchfahren kann. Hierzu ist das Problem der Kollisionserkennung zu lösen:

Definition 4.2.1 : Kollisionserkennung

Gegeben seien zwei geometrische Figuren im Raum. Das Problem der Kollisionserkennung ist die Fragestellung, ob die beiden Figuren kollidieren, d.h. ob sie sich überschneiden. Ein Algorithmus zur Kollisionserkennung sollte als „ja“ zurückgeben, wenn die gegebenen geometrischen Figuren einen (oder mehrere) Punkt(e) im Raum teilen und „nein“ sonst.

Die Kollisionserkennung von Unity basiert auf Nvidias PhysX-Physikframework und kann Kollisionen zwischen verschiedenen Typen von Objekten erkennen. Die 3D-Primitive, die in Unity Verwendung finden, sind:

- Quader (Box) – für beliebig rotierte Quader im Raum
- Kugeln (Sphere) – für Kugeln; Achtung: der Collider funktioniert nicht für Kugel-Meshes, die durch unterschiedliche Skalierung entlang der verschiedenen Achsen zu einem echten Ellipsoiden deformiert wurden.
- Kapsel (Capsule) – für Zylinder mit halbkugelförmigen Abschlüssen
- Mesh – für beliebige geometrische Formen, die Kollisionserkennung basiert auf den Polygonen ihres Meshes. Achtung: Unterstützt nur Kollisionen mit Collidern eines anderen Typs und konvexen Mesh-Collidern und ist bedeutend weniger performant als die übrigen Collider, sollte also spärlich eingesetzt werden.

Hinzu kommen zwei spezialisierte Collider für Unity-Terrain (Mesh-Collider mit Height-Map) und Räder für Rennspiele (Kollisionserkennung über automatisch verwaltete Raycasts), auf die wir in diesem Kurs nicht näher eingehen werden.

Die Kollisionserkennung erfolgt in zwei Phasen:

- *Broad-Phase:* Hier wird mit einer groben Überapproximation getestet, ob die zwei geometrischen Objekte kollidieren oder nicht. Dabei wird ein einseitiger Fehler garantiert: Wenn in der Broad-Phase die Antwort lautet „keine Kollision", dann ist diese Antwort notwendigerweise korrekt. Wird hingegen eine Kollision festgestellt, so kann nicht sicher gefolgert werden, dass tatsächlich eine Kollision vorliegt.
- *Narrow-Phase:* In diese Phase wird nur übergegangen, wenn in der Broad-Phase eine Kollision erkannt wurde. In diesem Fall ist eine genauere Untersuchung notwendig, jetzt werden also keine approximativen, sondern geometrisch exakte Verfahren verwendet, um die Kollision zu erkennen. In der Narrow-Phase gilt also, dass eine „Ja"-Antwort auch notwendigerweise bedeutet, dass es zu einer Kollision gekommen ist.

Bemerkung 4.2.2 : Fehlerquellen

In der Verwendung von Collidern kann es dennoch – auch abseits der stets möglichen Probleme der nummerischen Stabilität der Verfahren, besonders wenn man sich weit vom Ursprungspunkt der Szene entfernt – gelegentlich zu Fehlerkennungen der Art kommen, dass Kollisionen nicht erkannt werden, die man eigentlich erwarten würde. Hier sind vor allem zwei Fehlerquellen zu nennen:

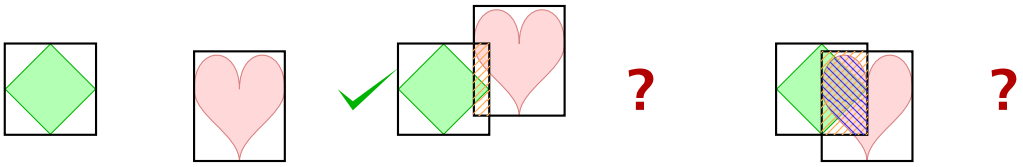
- Bewegung eines Objektes, die komplett durch ein anderes hindurch führt. Man stelle sich eine sich schnell bewegende Figur vor, die auf eine Wand zu läuft, die verhältnismäßig dünn ist. Es kann passieren, dass die Spielfigur auf Grund ihrer hohen Geschwindigkeit und der begrenzten Physik-Schrittgröße von üblicherweise 0,02 Sekunden die Spielfigut im einen Frame komplett vor, im an-

deren komplett hinter der Wand ist. Bei einer Veränderung der Position wie wir sie bisher vorgenommen haben – direktes Setzen der Position in `transform` – wird korrekterweise keine Kollision festgestellt. Unity-Komponenten wie der `CharacterController` erlauben es allerdings, kontinuierliche Bewegungen umzusetzen und dieses Problem zu vermeiden. Dieses Problem ist auch als „Tunneling“ bekannt.

- Überlastung der Physik-Engine. Gerade auf schwächeren Systemen oder wenn viele komplexe Meshes aufeinander stoßen, kann es passieren, dass nicht alle potenziellen Kollisionen im Physik-Zeitfenster überprüft werden können. So kann es passieren, dass wichtige Kollisionen nicht erkannt werden und so Fehlverhalten im Spiel zu beobachten ist. Besonders wenn Sie für mobile Spielkonsolen oder Handys entwickeln, sollten Sie diese Fehlerquelle im Kopf haben und ggf. testweise im Fall von Kollisionserkennungsfehlern Teile der Geometrie, besonders der nicht-statischen, ausschalten, um solche Fehler zu identifizieren.

4.2.1 Kollision zwischen Achsenparallelen Quadern

In der Broad-Phase wird unter anderem die folgende Approximation verwendet: Die beiden zu vergleichenden geometrischen Objekte werden mit einem achsenparallelen Begrenzungsquader (kurz und im Folgenden AABB, für engl. axis-aligned bounding box) umhüllt, das heißt, es wird der kleinste nicht rotierte Quader gewählt, der das jeweilige Objekt komplett enthält. Anschließend wird überprüft, ob diese beiden AABB sich überschneiden. Wenn sich die beiden AABB nicht überschneiden, dann überschneiden sich auch die darin enthaltenen geometrischen Objekte nicht. Wenn sich die beiden AABB hingegen überschneiden, dann lässt sich daraus nicht sicher schließen, ob sich die beiden geometrischen Objekte überschneiden und wir müssen das in der zweiten Phase, der Narrow-Phase, genauer überprüfen. In zwei Dimensionen lässt sich dieses Verfahren illustrieren, wobei die AABB hier die Form eines Begrenzungsrechtecks statt eines Quaders annimmt. Diese Illustration ist auch prototypisch für approximative Lösungen mit einseitigem Fehler, also Lösungen bei denen eine „ja“-Antwort immer korrekt, eine „nein“-Antwort aber möglicherweise falsch sein kann (oder umgekehrt), und ist somit ein wichtiger Rückgriff zu unserer Lektion zur Verifikation: Neben dem von uns vorgestellten Ansatz des Model Checkings mittels Modelleinschränkung ist der zweite essenzielle Ansatz für Verifikation eine Überapproximation des zu untersuchenden Problems mit einseitigem Fehler:



Nun stellt sich natürlich die Frage, wieso man diese Approximation überhaupt durchführt und der Grund ist, dass im Regelfall davon auszugehen ist, dass die meisten Kollisionsabfragen mit der AABB-Approximation bereits zuverlässig beantwortet werden können und die AABB-Approximation erlaubt eine Kollisionsabfrage mit besonders geringer Berech-

nungskomplexität:

Algorithmus 4.2.3 : Kollisionsabfrage zwischen zwei AABB

Gegeben seien zwei AABB a und b mit den Koordinaten x_0^a, x_0^b (x_1^a, x_1^b) für die x-Koordinate an der linken (rechten) Grenzfläche der AABB a und b, y_0^a, y_0^b (y_1^a, y_1^b) für die y-Koordinate an der unteren (oberen) Grenzfläche der AABB a und b und z_0^a, z_0^b (z_1^a, z_1^b) für die z-Koordinate an der vorderen (hinteren) Grenzfläche der AABB a und b.

Wir nutzen folgende Beobachtung: Zwei AABB kollidieren genau dann, wenn sie sich auf allen drei Achsen überschneiden, das Verfahren kann also wie folgt umgesetzt werden

- Falls $x_1^b < x_0^a$ gibt `false` zurück *(die rechte Kante von b liegt links von der linken Kante von a)*
- Falls $x_1^a < x_0^b$ gibt `false` zurück *(die rechte Kante von a liegt links von der linken Kante von b)*
- Falls $y_1^b < y_0^a$ gibt `false` zurück *(die obere Kante von b liegt unter der unteren Kante von a)*
- Falls $y_1^a < y_0^b$ gibt `false` zurück *(die obere Kante von a liegt unter der unteren Kante von b)*
- Falls $z_1^b < z_0^a$ gibt `false` zurück *(die hintere Kante von b liegt vor der vorderen Kante von a)*
- Falls $z_1^a < z_0^b$ gibt `false` zurück *(die hintere Kante von a liegt vor der vorderen Kante von b)*
- Gib `true` zurück

Alternativ in kompakter mathematischer Form:

$$\begin{aligned}\texttt{true} \Leftrightarrow & x_0^a \leq x_1^b \wedge x_0^b \leq x_1^a \wedge \\ & y_0^a \leq y_1^b \wedge y_0^b \leq y_1^a \wedge \\ & z_0^a \leq z_1^b \wedge z_0^b \leq z_1^a\end{aligned}$$

Das Verfahren kommt also ohne geometrische Transformationen nur mit dem Größenvergleich der Koordinaten aus – im Fall von Unity also gerade einmal sechs Vergleiche zwischen Paaren von `float`-Zahlen. Wenn nun dieses Verfahren `false` ausgibt, muss das Paar von geometrischen Objekten nicht weiter untersucht werden, die Kollisionserkennung kann direkt mit der Ausgabe `false` beendet werden. Anderenfalls muss unterschieden werden, welche Grundtypen miteinander verglichen werden müssen. In der Tat muss für jedes Paar von Grundformen ein eigener Algorithmus gefunden werden. Es ergeben sich

insgesamt die folgenden Probleme: bei den Kollisionen, bei denen kein allgemeiner Mesh involviert ist, ist außerdem die grundlegende Idee der Kollisionserkennung angegeben.

1. Kugel-Kugel-Kollision: Berechne den Abstand zwischen den Mittelpunkten und vergleiche mit den Radien.

2. Kugel-Quader-Kollision: Berechne den Abstand zwischen dem Mittelpunkt und dem Quader und vergleiche mit dem Radius.

3. Kugel-Kapsel-Kollision: Berechne den Abstand zwischen dem Mittelpunkt der Kugel und der Mittelstrecke der Kapsel und vergleiche mit den Radien.

4. Quader-Quader-Kollision: Überprüfe, ob es eine separierende Ebene zwischen den beiden Quadern gibt.

5. Quader-Kapsel-Kollision: Berechne den Abstand zwischen der Mittelstrecke der Kapsel und allen Quaderseitenstrecken und vergleiche mit dem Radius der Kapsel.

6. Kapsel-Kapsel-Kollision: Berechne den Abstand zwischen den beiden Mittelstrecken der Kapseln und vergleiche mit den Radien.

Hinzu kommen die Mesh-Kollisionen. Bei den Mesh-Kollisionen muss ein Vergleich mit jedem Dreieck, aus dem der Mesh besteht, erfolgen:

7. Quader-Mesh-Kollision

8. Kugel-Mesh-Kollision

9. Kapsel-Mesh-Kollision

10. Konvexer Mesh-Mesh-Kollision

Wir werden im Folgenden nicht alle diese Verfahren besprechen, für eine vollständige mathematische und informatische Diskussion der Kollisionserkennung sei das folgende Buch nahegelegt, an dem sich auch die Darstellung der Algorithmen zur Kollisionserkennung orientiert, die wir in diesem Kapitel besprechen:

- Christer Ericson: Real-Time Collision Detection, Taylor & Francis, 2004

Bevor wir die Kollisionsalgorithmen besprechen, noch eine Anmerkung zur numerischen Stabilität: Da wir mit Fließkommazahlen arbeiten, ist es ratsam, Vergleiche nicht exakt durchzuführen, sondern zu prüfen, ob zwei zu vergleichende Zahlen eine Differenz kleiner als ein (kleines zu wählendes) Epsilon aufweisen. Im Sinne der einfacheren Lesbarkeit werden die Algorithmen in diesem Kapitel alle ohne Beachtung dieses Umstandes umgesetzt; wenn Sie diese Verfahren für einen Produktiveinsatz implementieren wollen, sollten Sie diese einfache Opimierung aber vornehmen.

4.2.2 Kollision zwischen zwei Kugeln

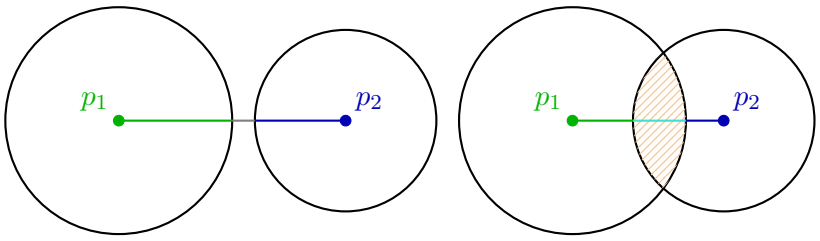
Der einfachste genaue Algorithmus für die Kollisionsbestimmung ist die Kollision zwischen zwei Kugeln:

Algorithmus 4.2.4 : Kugel-Kugel-Kollisionserkennung

Gegeben: Zwei Kugeln über ihre Mittelpunkte `p1, p2` und ihre Radien `r1, r2`
Gesucht: Ja, falls die beiden Kugeln einen gemeinsamen Punkt haben, nein sonst.

```
bool collisionSphereSphere(Vector3 p1, float r1,
  Vector3 p2, float r2)
{
  float sqrDist = (p1-p2).sqrMagnitude;
  float maxSqrDist = (r1+r2)*(r1+r2);
  return sqrDist < maxSqrDist;
}
```

Die Idee des Verfahrens ist in der folgenden Grafik in zwei Dimensionen illustriert:



Keine Kollision Kollision mit Überschneidung

Aus Optimierungsgründen wird nicht der Abstand der Punkte miteinander verglichen, sondern der quadrierte Abstand (über die Unity-Funktion `sqrMagnitude`) – da uns nur der Vergleich mit einem Schwellenwert interessiert, ist das eine Option und man spart sich die aufwendige Berechnung der Quadratwurzel bei der Berechnung des Abstandes zwischen den beiden gegebenen Punkten.

4.2.3 Der Trennungssatz von Eidelheit

Für die übrigen fünf Vergleiche zwischen geometrischen Grundformen spielt der Vergleich zwischen Punkten und Strecken, sowie zwischen Strecken im $\mathbb{R}^3$ eine entscheidende Rolle.

Um zu überprüfen, ob zwei Quader kollidieren, die nicht notwendigerweise achsenparallel sind (im Englischen spricht man auch von OBB, object bounding box) verwenden wir den Trennungssatz (auch: Satz von Eidelheit) in seiner konkreten Instanziierung auf den $\mathbb{R}^3$. Der Satz besagt:

Satz 4.2.5 : Trennungssatz (von Eidelheit)

Gegeben seien zwei nicht-leere, abgeschlossene [a], kompakte [b] und konvexe [c] Mengen $M_1, M_2 \subset \mathbb{R}^3$ mit $M_1 \cap M_2 = \emptyset$, dann gibt es eine Ebene E, die die beiden trennt, dass heißt M_1 liegt vollständig auf der einen Seite von E, M_2 auf der anderen.

[a] Eine Menge M ist abgeschlossen, wenn ihr Komplement $\mathbb{R}^3 \setminus M$ offen ist

[b] Kompakt bedeutet, dass aus jeder Überdeckung mit offenen Mengen eine endliche Teilüberdeckung ausgewählt werden kann.

[c] Konvex bedeutet, dass für jedes Paar von Punkten in der Menge auch die Verbindungslinie komplett in der Menge liegt.

Beweisskizze: Da M_1 und M_2 beide kompakt und disjunkt sind, gibt es zwei Punkte $p_1 \in M_1$ und $p_2 \in M_2$ minimalen Abstands, also

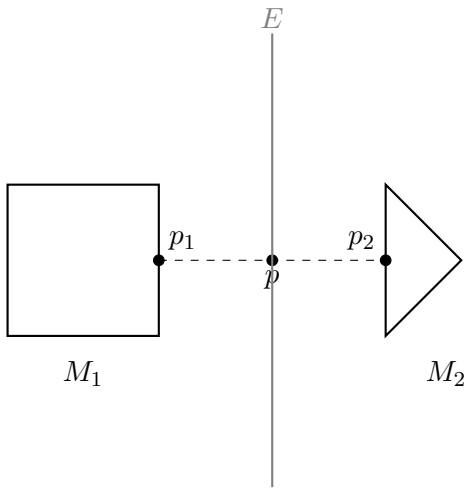
$$|p_1 - p_2| = \min\{|x - y| \mid x \in M_1, y \in M_2\}.$$

Wir bilden die Ebene E, die durch den Mittelpunkt $p = \frac{p_1+p_2}{2}$ zwischen p_1 und p_2 geht und orthogonal zur Verbindungsstrecke zwischen p_1 und p_2 liegt:

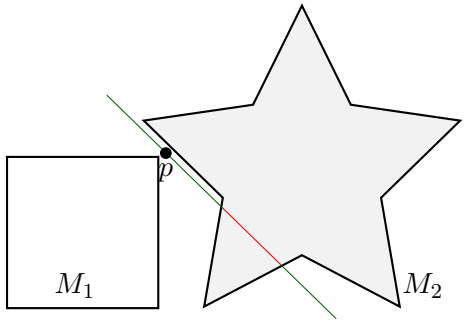
$$E = \{x \in \mathbb{R}^3 : (x - p) \cdot (p_2 - p_1) = 0\}$$

Da p_1 der Punkt aus M_1 ist, der am nächsten an E liegt und p_2 der Punkt aus M_2 ist, der am nächsten an E liegt und beide Mengen konvex sind, liegen alle Punkte von M_1 auf der einen und alle Punkte von M_2 auf der anderen Seite von E.

Das Ergebnis lässt sich in zwei Dimensionen wie folgt skizzieren, beispielsweise mit einem Quadrat und einem Dreieck, die Trennebene ist in zwei Dimensionen eine Gerade:



Die Konvexität ist entscheidend für diesen Satz, wie das folgende Gegenbeispiel demonstriert:



Die rote Teilstrecke zeigt, wo die Trennebene die nicht konvexe Menge M_2 schneidet.

Eine wichtige Operation, die wir für die weiteren Kollisionserkennungsverfahren benötigen, ist der Clamp (engl. für Einklemmen). Der Clamp ist im Grunde eine ganz simple Operation, bei der ein Zahlenwert auf ein bestimmtes Intervall beschränkt und bei Unter- oder Überschreiten der Bereichsgrenze auf den Wert der Bereichsgrenze gesetzt wird. Hierbei ist allerdings wichtig, die Reihenfolge der unteren und der oberen Schranke zu beachten. Vertauscht man die beiden Grenzen, ist die Ausgabe des Clamping unbrauchbar.

Algorithmus 4.2.6 : Clamping

```
float clamp(float x, float lower, float upper)
{
  if(x<lower) return lower;
  if(x>upper) return upper;
        return x;
}
```

4.2.4 Kollision zwischen Kugel und Kapsel

Das nächste Verfahren, das wir erarbeiten möchten, ist der Vergleich zwischen einer Kugel und einer Kapsel. Eine Kapsel kann interpretiert werden als eine Strecke (oder auch Liniensegment) um die herum unendlich viele Kugeln gleichen Radius gelegt werden. Eine Kapsel und eine Kugel schneiden sich demnach, wenn der Abstand zwischen der Stecke im Zentrum der Kapsel und dem Mittelpunkt der Kugel kleiner ist als die Summe der Radien der Kugel und der Kapsel (Versuchen Sie, mit dem Trennungssatz nachzuvollziehen, wieso es anderenfalls eine Trennebene geben muss!). Wir geben also einen Algorithmus an, um den Punkt auf einer Stecke anzugeben, der am nächsten an einem gegebenen Punkt liegt.

Algorithmus 4.2.7

```
Vector3 closestPointSegmentPoint(Vector3 segmentA,
  Vector3 segmentB, Vector3 point)
{
  Vector3 ab = segmentB-segmentA;
  // Betrachte die Gerade, auf der das Segment
  // zwischen segmentA und segmentB liegt.
  // Berechne, wie weit man von segmentA nach segmentB
  // laufen muss, um eine Gerade zu finden, die senkrecht
  // auf der Geraden liegt und durch point läuft.
  float positionOnLine = Vector3.Dot(point-segmentA, ab)
    / Vector3.Dot(ab, ab);
  // Berechne den Punkt auf der Geraden, die wir gerade
  // charakterisiert haben und clampe ihn auf die Strecke
  // von segmentA nach segmentB
  return segmentA + clamp(positionOnLine, 0, 1) * ab;
}
```

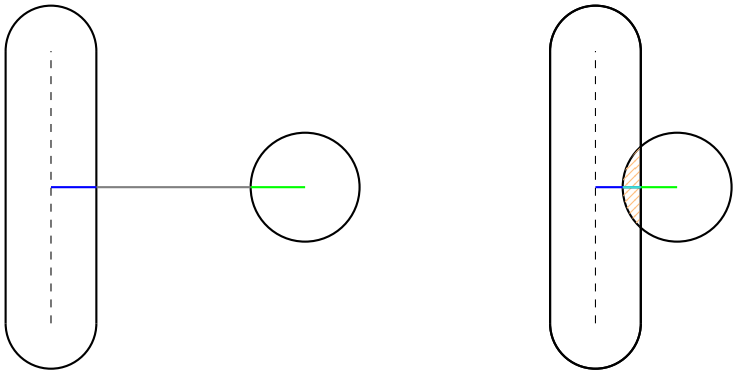
Damit ist die Kollision zwischen Kapsel und Kugel schließlich leicht zu erkennen:

Algorithmus 4.2.8 : Kugel-Kapsel-Kollisionserkennung

```
bool collisionSphereCapsule(Vector3 centerSphere,
  float radiusSphere, Vector3 segmentA, Vector3 segmentB,
  float radiusCapsule)
{
  Vector3 pointOnCapsule = closestPointSegmentPoint(
    segmentA, segmentB, centerSphere);
  float sqrDistance = (pointOnCapsule
    -centerSphere).sqrMagnitude;
  float maxDistance = (radiusCapsule+radiusSphere)
    * (radiusCapsule+radiusSphere);
  return sqrDistance <= maxDistance;
}
```

Wie schon bei der Kollisionserkennung zwischen zwei Kugeln vergleichen wir hier wieder die quadrierten Abstände statt die Abstände selbst, um die aufwendige Berechnung der Quadratwurzel einzusparen.

Die folgende Skizze illustriert das Verfahren in zwei Dimensionen, im linken Fall liegt keine Kollision vor, im rechten schon.



4.2.5 Kollisionserkennung zwischen zwei Kapseln

Die grundlegende Idee für die Kollisionserkennung zwischen zwei Kapseln ist simpel: Wir bestimmen den quadratischen Abstand zwischen den beiden Mittelstrecken der beiden Kapseln und wenn dieser kleiner oder gleich das Quadrat der Summe der Radien der beiden Kapseln ist, schneiden sich die Kapseln:

Algorithmus 4.2.9 : Kaspel-Kapsel-Kollisionserkennung

```
bool collisionCapsuleCapsule(Vector3 c1SegmentA, Vector3
  c1SegmentB, float c1Radius, Vector3 c2SegmentA, Vector3
  c2SegmentB, float c2Radius)
{
  float sqrDistance = distanceSegments(c1SegmentA,
    c1SegmentB, c2SegmentA, c2SegmentB);
  return sqrDistance <= (c1Radius+c2Radius)
    * (c1Radius+c2Radius);
}
```

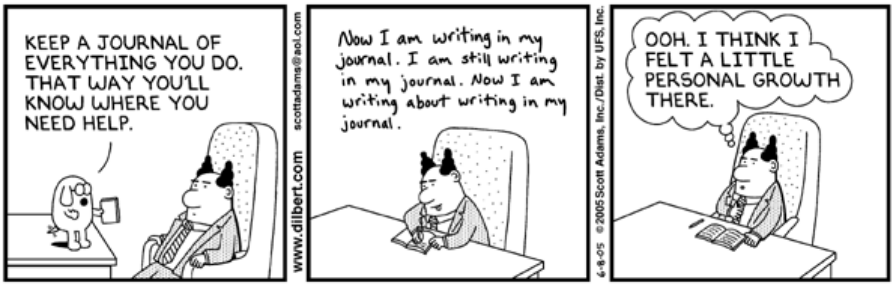
Dieser Algorithmus nutzt eine Methode `distanceSegments`, die die Distanz zwischen zwei Strecken ermittelt. Ein zentraler Schritt besteht darin, zunächst die Punkte auf den beiden Geraden zu bestimmen, die den geringsten Abstand zueinander haben. Eine Strecke s kann über zwei Punkte a und b angegeben werden und die Punkte auf der Strecken können ermittelt werden gemäß

$$a + (b - a) \cdot t, t \in [0, 1].$$

Die Gerade, durch die die Strecke verläuft, erhält man, indem man einfach die Bedingung $t \in [0, 1]$ streicht. In unserem Fall haben wir also zwei Strecken s_1 und s_2 mit den definierenden Formeln

$$s_1 = \{s_1(t) = a_1 + (b_1 - a_1) \cdot t \mid t \in [0, 1]\} \qquad s_2 = \{s_2(t) = a_2 + (b_2 - a_2) \cdot t \mid t \in [0, 1]\}.$$

Kurseinheit 6: Metaprogrammierung



© Scott Adams, Inc./Dist. by UFS, Inc.

Von Metaprogrammierung spricht man, wenn ein Programm, das Metaprogramm, ein anderes (oder sich selbst) oder seine Ausführung beobachtet oder sogar manipuliert. Metaprogrammierung ist rekursiv anwendbar, was soviel heißt wie dass Metaprogramme selbst der Metaprogrammierung unterliegen können.

unverzichtbare Metaprogramme

Zu den bekanntesten Metaprogrammen zählen Programmierwerkzeuge wie Compiler, Interpreter, Optimierer, Werkzeuge zur Programmanalyse (Metrikwerkzeuge, Kontrollflussanalyse etc.) sowie Refactoring-Werkzeuge und Debugger. All diese Programme haben andere Programme zum Gegenstand — sie helfen sie zu erzeugen, zu analysieren und zu korrigieren. Tatsächlich wäre ohne Metaprogrammierung die Programmierung heute undenkbar, nur nehmen wir diese Metaprogramme in der Regel nicht als solche wahr.

herausfordernde Metaprogramme

Zu den interessantesten Metaprogrammen gehören solche, die auf sich selbst angewendet werden. Besonders umstritten ist dabei sog. selbstmodifizierender Code, da er in gewisser Weise zu einer Verselbständigung der Programmierung führt, die für ein Leser des Programms nur noch sehr schwer nachzuvollziehen ist. Entsprechend sind auch sämtliche Möglichkeiten zur Verifikation von selbstmodifizierendem Code (Type checking etc.) stark eingeschränkt.

Voraussetzung der Metaprogrammierung

Eine wichtige Voraussetzung der Metaprogrammierung ist, dass Programme selbst die Form von Daten haben. Dies ist trivialerweise bei Quellcode immer der Fall: Ein Programm kann als Text eingelesen und in der Folge analysiert bzw. manipuliert werden. Voraussetzung ist dann allerdings, dass man aus dem Programm heraus Zugriff auf einen Compiler bzw. Interpreter hat, was nicht immer der Fall ist (es sei denn, beim Metaprogramm handelt es sich selbst um einen Compiler bzw. Interpreter). Wenn man diese erst nachbilden muss, wird die Metaprogrammierung zu einer recht beschwerlichen Angelegenheit.

Einfacher und direkter ist die Metaprogrammierung dann, wenn sich ein ausführbares Programm syntaktisch nicht von seinen Daten unterscheidet. Dies ist zum Beispiel in den Programmiersprachen LISP und PROLOG der Fall. Beide verwenden einheitlich Listen bzw. Terme, um sowohl Programme als auch Daten zu repräsentieren. Da die Möglichkeit besteht, in Variablen gespeicherte Listen bzw. Terme zur Ausführung zu bringen, ist die Metaprogrammierung diesen Sprachen inhärent.

Ursprung der Metaprogrammierung

Die Metaprogrammierung ist so alt wie die Programmierung selbst. In einer frühen Auseinandersetzung zwischen Alan Turing und John von Neumann ging es darum, ob man eine bedingte Verzweigung als Maschineninstruktion benötigt. Turing vertrat dabei die Ansicht, dass man die von einer Bedingung abhängige Sprungadresse auch berechnen und hinter der Sprunganweisung in den Programmspeicher schreiben könne. Turing wollte also ausnutzen, dass im Maschinencode Daten und Programme das gleiche sind, nämlich einfach Zahlen. Heute ist man allerdings verbreitet der Ansicht, dass es sich zumindest beim schreibenden Zugriff eines Programms auf den Speicherbereich, in dem es abgelegt ist, um einen Programmierfehler (oder Angriff) handelt, und ist bemüht, diesen so früh wie möglich (durch das Betriebssystem oder gar durch eine physikalische Segmentierung des Speichers in Programm- und Datenbereiche) zu verhindern. Tatsächlich waren zumindest früher nicht wenige der sog. Systemabstürze darauf zurückzuführen, dass ein Programm statt in den Daten- in den Programmbereich schrieb und damit in der Regel unsinnige Anweisungen vorgab.

Einsatzmöglichkeiten der Metaprogrammierung

Neben den eben schon geschilderten Standardanwendungen der Metaprogrammierung, den Programmierwerkzeugen, hat sie noch weitere interessante Einsatzgebiete. Da sind zum einen Spracherweiterungen bzw. -anpassungen, die immer dann gewünscht werden, wenn einer verwendeten Programmiersprache für ein bestimmtes Anwendungsproblem geeignete Abstraktionen fehlen. Anstatt die Sprache selbst (und damit alle daran hängenden Werkzeuge) zu ändern, erlauben Sprachen mit Möglichkeit zur Metaprogrammierung, die neuen Sprachkonstrukte als eine Art Bibliotheksfunktionen hinzuzufügen. Unter den Sprachen, die in der kommerziellen Programmierung eingesetzt werden, seien hier insbesondere C++ und SMALLTALK genannt, wobei erstere das Problem der Spracherweiterung auf den Präprozessor abwälzt, während letztere, von wenigen Primitiven abgesehen, ganz in sich selbst definiert ist (und daher ohne die Metaprogrammierung gar nicht wiederzuerkennen wäre; vgl. Kurs 01814).

Zum anderen kommt die Metaprogrammierung im Kontext dynamisch konfigurierter Systeme und Dienste zum Einsatz. Überall dort, wo die Schnittstellen der zu verknüpfenden Komponenten zur Übersetzungs- (bzw. Binde-) Zeit nicht bekannt sind, kann auch kein Zueinanderpassen dieser Schnittstellen gewährleistet werden. Mit den Mitteln der Metaprogrammierung ist es jedoch möglich, die Schnittstellen zu inspizieren und bei Bedarf Adapter zur Verfügung zu stellen, die zwischen den Komponenten vermitteln.

Künstliche Intelligenz

Nicht zuletzt ist die *Künstliche Intelligenz* ein klassisches Anwendungsgebiet der Metaprogrammierung. Regelbasierte Systeme können selbst neue Regeln dazulernen — wie dies zu geschehen hat, kann wieder in Regeln ausgedrückt werden. In der sog.

genetischen Programmierung werden bestehende Programme nach bestimmten Regeln, teilweise zufällig, verändert (mutiert) und zur Ausführung gebracht. Das Ergebnis der Ausführung wird mit dem erwarteten Ziel verglichen; das betreffende Programm bleibt abhängig davon weiter in der Population oder wird ausgesondert (Selektion). Sowohl für die Mutation als auch für die Selektion ist Metaprogrammierung notwendig.

6.1 Metaprogrammierung auf sich selbst: Reflexion

In der Programmierung versteht man unter **Reflexion** die Fähigkeit eines Programms, sich selbst zu (er)kennen und zu behandeln. Dazu enthält ein reflexives Programm seinen eigenen Interpreter oder zumindest Teile davon. Ein reflexives Programm ist also ein besonderes Metaprogramm, nämlich eines, bei dem Metaprogramm und Programm eins sind.

Ein vollständig reflexives System besitzt die Fähigkeit, seinen eigenen Aufbau und seine eigene Ausführung zu beobachten (die sog. *Introspektion*), seinen Ablauf zu beeinflussen (die sog. *Interzeption* oder *Interzession*) sowie ganz allgemein seine Bestandteile zu ändern, zu ergänzen oder auszutauschen (die *Modifikation*). So kann ein vollständig reflexives System beispielsweise Methodenaufrufe abfangen und bei Bedarf deren Ausführung verhindern oder auf andere, auch neue, Methoden umleiten.

6.1.1 Reflexieren ohne zu verändern: Introspektion

Weniger mächtig, aber einfacher zu realisieren als die allgemeine Reflexion ist die sog. **Introspektion**, also die Reflexion ohne die Möglichkeit der Interzession und Modifikation. Sie ist beispielsweise in JAVA umgesetzt. Introspektion wird oft mit Reflexion gleichgesetzt — so heißt beispielsweise das introspektive Sprachpaket JAVAS irreführenderweise `java.lang.reflect`.

Bei der Introspektion hat ein Programm die Möglichkeit, über seinen eigenen Aufbau zu reflexieren, ohne ihn jedoch verändern zu können. Das Programm weiß also beispielsweise, aus welchen Klassen es besteht, welche Methoden in diesen Klassen implementiert sind, welche Typen deren Parameter haben usw. So ermöglicht es die Introspektion einem Programm beispielsweise, sich ohne Zugriff auf den Quellcode zumindest teilweise selbst auszudrucken. Ein anderes Anwendungsbeispiel verlangt, dass eine Schnittstelle selbst darüber Auskunft gibt, welche Funktionen sie anbietet. Diese Selbstauskunft kann auf einfache Weise über Introspektion realisiert werden.

indirekter Zugriff

Wirklich interessant wird die Introspektion jedoch erst, wenn sie nicht nur über Programmelemente Auskunft geben, sondern auch auf diese zugreifen oder diese zur Ausführung bringen kann. So wird es beispielsweise möglich, den Wert eines durch einen String benannten Feldes auszulesen oder zu verändern oder auch eine Methode nur anhand ihres Namens (wiederum als String vorliegend) auszuwählen und aufzurufen. Man beachte, dass dabei das vorliegende Programm und insbesondere der durch seine Strukturen vorgegebene Kontrollfluss nicht verändert werden. Im Falle eines solchen Methodenaufrufs ist

Tabelle 6.1: JAVA-Klassen mit Beteiligung am Reflection API.

KLASSE	BESCHREIBUNG
Array	Erzeugung von und Zugriff auf Arrays variablen Typs
Class	Instanzen repräsentieren Klassen und Interfaces
Constructor	Aufruf von Konstruktoren variablen Typs
Field	Information über und Zugriff auf Felder variabler Typen
Method	Information über und Aufruf von Methoden variabler Typen
Modifier	Information über Access Modifier von Typen und Methoden
Object	getClass() Methode
–	instanceof testet den Typ eines Objekts
–	.class liefert das die Klasse repräsentierende Objekt

lediglich aus dem Programmtext nicht ersichtlich, welche Methode aufgerufen wird; die aufgerufene Methode selbst muss jedoch existieren (und somit schon zur Übersetzungszeit existiert haben, ohne dass eine entsprechende Prüfung durch den reflexiven Aufruf hätte veranlasst werden können). Man kann allerdings mit Introspektion neue Kombinationen von bereits bestehenden Elementen erzeugen, also Systeme dynamisch konfigurieren.

6.1.2 Introspektion in JAVA

Um JAVA zu einer introspektiven Sprache zu machen, verfügt die Klassenbibliothek des JDK über Klassen, die die Elemente eines Programms repräsentieren. So stehen Instanzen der Klasse Class für die Klassen eines Programms, Instanzen der Klasse Field für die Felder (Klassen- und Instanzvariablen) und Instanzen der Klasse Method für Methoden. Zusätzlich gibt es noch Klassen für Arrays (Array), für Konstruktoren (Constructor) sowie für die Zugriffsmodifizierer public, private etc. (Modifier). Nicht zuletzt besitzt die Klasse Object mit ihrer Methode getClass() einen introspektiven Anteil. Und schließlich ist auch die Tatsache, dass in JAVA die Klasse eines Objekts zur Laufzeit bekannt ist, in die Syntax der Sprache fest eingebaut: Der Operator instanceof erlaubt, zu testen, ob ein gegebenes Objekt Instanz einer bestimmten Klasse ist. Wenn im Programmtext selbst eine Klasse K als Literal (also als ein Wert) auftauchen soll, dann schreibt man K.class (nicht jedoch bei instanceof). Tabelle 6.1 gibt einen Überblick.

Beispiel Nachfolgender Code zeigt, wie man in JAVA Objekte beliebiger, von ReflectiveObject abgeleiteter Klassen serialisieren, d. h. auf einem Stream ausgeben kann:

```
1493 class ReflectiveObject {
1494     void serializeOn(PrintStream aStream) {
1495         Class myClass = getClass();
1496         aStream.println(myClass.getName() + "(");
1497         for (Field field : myClass.getFields()) {
1498             aStream.print(field.getName() + " == ");
1499             try {
1500                 Object value = field.get(this);
```

```
1501                if (field.getType().isPrimitive() || value == null)
1502                    aStream.println(value);
1503                else
1504                    ((ReflectiveObject) value).serializeOn(aStream);
1505            }
1506            catch …
1507        }
1508        aStream.println(")");
1509    }
1510 }
```

Introspektive Zugriffe auf Programmteile entziehen sich der syntaktischen Prüfung durch den Compiler. Daher können Laufzeitfehler, die auf eine mangelnde Bindung von Namen zurückzuführen sind, nicht durch den Compiler ausgeschlossen werden. Introspektive Aufrufe müssen also immer durch Try-catch-Blöcke gesichert werden:

```
1511 try {
1512    o.getClass().getMethod("hello",new Class[] {String.class}).
                invoke(o, new Object[] {"world"});
1513 } catch(NoSuchMethodException ex) {
1514    throw new IllegalArgumentException("Methode nicht gefunden");
1515 } catch (IllegalAccessException ex) {
1516    throw new Exception("Methode nicht zugreifbar");
1517 } catch (InvocationTargetException ex) {
1518    throw ex.getTargetException(); //Methode hat Exception ausgelöst
1519 }
```

Problem Refactoring

Des Weiteren werden introspektive Abfragen und Aufrufe nicht durch Refactoring-Werkzeuge erfasst: Das Umbenennen einer Methode geht an den Funktionen, die einen String zusammenbasteln, der eben diese Methode benennen soll, gänzlich vorbei. Vorsicht ist also auch bei der Introspektion geboten.

Die Einsatzmöglichkeiten der Introspektion sind vielseitig. So macht sich z. B. JUNIT die introspektiven Eigenschaften JAVAs zunutze, um Testfälle, die als Methoden eines Programms vorliegen, von JUNIT aber als Objekte verwaltet werden, zu erkennen, intern zu repräsentieren und zu Testsuiten zusammenzufassen. Auch verwenden Persistenzframeworks wie das Java Persistence API (JPA) Reflexion, um zur Laufzeit die Abbildung von Java-Klassen auf Tabellen einer relationalen Datenbank zu steuern.

Selbsttestaufgabe 6.1

In JAVA berücksichtigt das *dynamische Binden* eines Methodenaufrufs an eine Implementierung nur beim Empfängerobjekt den tatsächlichen (dynamischen) Typ. Bei den Parametern hingegen wird lediglich der deklarierte (statische) Typ herangezogen. Dieses Verhalten ist jedoch nicht immer erwünscht.

Versuchen Sie, mittels Introspektion eine Lösung vorzuschlagen, mit Hilfe derer das Dispatching bei Methodenaufrufen immer auf Empfänger- und Parametertypen stattfindet.

6.1.3 Interzession

Zwar erlaubt die Introspektion (Reflexion à la JAVA), auf Programmelemente indirekt, also ohne explizite Erwähnung der Elemente im Quellcode, zuzugreifen, doch bestehende, „fest verdrahtete" Zugriffe lassen sich auf diese Weise ebenso wenig abändern wie neue Elemente einführen. Was man manchmal aber gerne hätte, ist die Möglichkeit, Methodenaufrufe oder Variablenzugriffe abzufangen und beispielsweise auf ihre Zulässigkeit zu überprüfen oder situationsbezogen umzulenken. Derartiges, **Interzession** genannt, ist in JAVA zur Zeit nur über entsprechende Entwurfsmuster (wie z. B. das *Proxy Pattern* [36]) möglich; da es sich dabei aber nicht um Metaprogrammierung im strengen Sinne handelt, werden wir an dieser Stelle auch nicht darauf eingehen. Die aspektorientierte arbeitet aber sehr wohl auf Basis der Interzession.

6.1.4 Modifikation

Um ein Programm sich selbst um neuen Code ergänzen oder vorhandenen modifizieren zu lassen — die sog. **Modifikation** —, muss das Programm Zugriff auf seinen eigenen Quellcode und einen Compiler haben. Für JAVA beispielsweise besteht die Möglichkeit, Klassen in Form von Quellcode-Dateien zu erzeugen, dann diese Dateien zu kompilieren (bis einschließlich JAVA 5 nur über den Umweg eines Betriebssystemaufrufs, seit JAVA 6 durch eine eigene API) und über einen Class loader zu importieren. Alternativ kann man natürlich auch direkt Bytecode erzeugen und laden (wer es mag). Ein Hauptvorteil der Verwendung von Compilern geht dabei jedoch stets verloren: die Erkennung von Programmfehlern zur Übersetzungszeit (die heute häufig mit der Entwurfszeit gleichzusetzen ist). Stattdessen fallen Übersetzungsfehler eines selbstmodifizierenden Programms zur Laufzeit (der Modifikationszeit) an. Dass die meisten Sprachen, die selbstmodifizierende Programme erlauben, interpretiert werden, ist insofern nur konsequent.

6.1.5 Bewertung der Reflexion

Zweifellos bietet die Reflexion gewichtige Vorteile. Neben der Möglichkeit der Ergänzung von Funktionalität eines Programms zur Laufzeit (durch Ergänzung zusätzlicher Anweisungen oder Programmteile) ist hier vor allem die Erweiterung der Ausdrucksstärke (nicht in formalem Sinne — die Sprache bleibt natürlich Turing-äquivalent) zu nennen. So kann eine Programmiersprache mit den Mitteln der Reflexion vergleichsweise einfach um fehlende Konzepte erweitert werden. Aber auch unter theoretischen Gesichtspunkten ist die Reflexion durchaus interessant: Ohne Reflexion ließe sich zum Beispiel das Halteproblem nicht formulieren.

Diesen Vorteilen stehen aber auch einige gewichtige Nachteile gegenüber. Zum einen sind Programme, die die Reflexion verwenden (die also selbst Metaprogramme sind), schwerer lesbar. Dies ist insbesondere dann der Fall, wenn der durch die Metaprogrammierung erstellte Code nicht im Klartext vorliegt, sondern durch das Programm eigenständig zusammengesetzt wird (was charakteristischerweise der Fall ist). Aus der schlechteren Lesbarkeit

ergibt sich auch eine schlechtere Wart- und Testbarkeit. Zum anderen verwischt die Grenze zwischen Übersetzungszeit und Laufzeit: Während das Metaprogramm selbst noch zur Übersetzungszeit prüfbar ist, können syntaktische und andere Fehler in meta-, also automatisch programmierten Codeteilen, erst zur Laufzeit festgestellt werden.

6.2 Zusammenfassung und Ausblick

Die Metaprogrammierung ist ein mächtiges Konzept, das das gemeine Programmierer allzu leicht überfordert. Da es keinen Quellcode im üblichen Sinne (also in Form eines zusammenhängend lesbaren Textes) gibt, ist Metaprogrammiertes nur schwer nachzuvollziehen. Überraschungen sind praktisch unvermeidlich und die Wahrscheinlichkeit, dass man metaprogrammierte Applikationen nur per Versuch und Irrtum richtig hinbekommt, ist groß. So ist es auch nicht verwunderlich, dass Sprachen, denen die Metaprogrammierung inhärent ist (wie z. B. PROLOG oder LISP), in der Wirtschaft keine besonders weite Verbreitung gefunden haben. Dem gegenüber stehen die vergleichsweise hohe Kompaktheit (niedrige Redundanz, d. h. wenig gleicher oder auch nur ähnlicher Code) sowie die extreme Flexibilität, die sich mit den Mitteln der Metaprogrammierung erzielen lässt. Weiterführende Literatur

[36] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns — Elements of Reusable Software* (Addison-Wesley, 1995). Dublette zu [1].

[37] DL Parnas „On the criteria to be used in decomposing systems into modules" *Communications of the ACM* 15:12 (1972) 1053–1058. (Dublette zu [3])

[38] EW Dijkstra *A Discipline of Programming* (Prentice Hall, Englewood Cliffs, New Jersey, 1976).

[39] R Laddad *ASPECTJ in Action* (Manning, 2003).

[40] J Hannemann, G Kiczales: „Design pattern implementation in Java and AspectJ" *OOPSLA* (2002) 161–173.

[41] KJ Lieberherr, DH Lorenz, J Ovlinger „Aspectual collaborations: combining modules and aspects" *The Computer Journal* 46:5 (2003) 542–565.

[42] F Steimann „The paradoxical success of aspect-oriented programming" in: *OOPSLA 2006 — Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications* (2006) 481–497.

6.3 Lösungen der Selbsttestaufgaben

Selbsttestaufgabe 6.1 (Seite 225)

```
1520 class Object {
1521     public Object perform(String methodname, Object[] parameters) {
1522     //lookup method, terminate if not available
```

```
1523        …
1524        //create parameter type array
1525        …
1526        //find best matching signature, terminate if not available
1527        …
1528        //invoke method with parameters
1529        …
1530        //return result
1531        …
1532    }
```