Seminararbeit Software Modellierung:

„Z-Notation und Unified Modelling Language (UML)“

Student:	Christian Schwörer
Matrikel-Nummer:	[redacted]
vorgelegt am:	29. Februar 2008
Vorlesung:	Software Modellierung
Studiengang:	Computer Science And Media (Master)
Betreuender Professor:	Prof. Dr. Edmund Ihler
Studienhalbjahr:	1. Semester

Inhaltsverzeichnis

Einleitung

„Wir bauen Software wie Kathedralen:
Zuerst bauen wir – Dann beten wir.“
(Prof. Dr. Gerhard Chroust)

Auch wenn dieses Zitat bereits aus dem Jahr 1992 stammt, charakterisiert es auch heute noch viele Probleme der modernen Softwareentwicklung: Die zunehmende Komplexität der Anwendungen und die daraus resultierende Notwendigkeit der formalen Systembeschreibung erfordern Modellierungssprachen und Notationen, mittels denen Anforderungen, Strukturen und innere Abläufe möglichst genau spezifiziert werden können. Erst damit wird es möglich, eine Spezifikation für Entwickler, Management, Benutzer und andere Beteiligte möglichst verständlich zu machen.

Diese Arbeit entstand im Rahmen der Vorlesung „Software Modellierung“ im ersten Semester des Master-Studiengangs *Computer Science and Media* an der HdM Stuttgart.
Sie beinhaltet die Auseinandersetzung mit zwei unterschiedlichen Ansätzen zur Spezifikation von Softwaresystemen: die *formale Spezifikation mit der Z-Notation* und die *Modellierung mit der Unified Modeling Language (UML)*.

Dazu werden in den ersten beiden Kapiteln die Grundlagen geschaffen, um dann im dritten Kapitel eine in Z verfasste Fallstudie in die UML zu übertragen. Im vierten Teil findet abschließend ein allgemein gehaltener Versuch der Übertragung von bestimmten Bestandteilen der Z-Notation in die UML statt.

1. Formale Spezifikation mit der Z-Notation

In diesem Kapitel wird ein Überblick über *formale Spezifikation mit der Z-Notation* gegeben. Eine vollständige Übersicht ist im Rahmen dieser Arbeit nicht möglich, daher werden lediglich die wesentlichen Bestandteile erklärt, die für das Verständnis der Fallstudie „A Drinks Dispensing Machine" in Kapitel 3 benötigt werden. Ausführliche Informationen und die vollständige Anleitung zur Z-Notation sind in entsprechender Fachliteratur zu finden.[1)]

1.1 Formale Spezifikation

Eine *formale Spezifikation* beschreibt auf eine präzise, konsistente und unmissverständliche Weise die Eigenschaften und das Verhalten eines Informationssystems, ohne dabei genauer darauf einzugehen, wie dieses Verhalten erreicht wird. Das heißt, es wird beschrieben *WAS* ein System tun muss, nicht aber *WIE* dies erreicht wird. [2)]

Die Beschreibung erfolgt dabei mittels einer *mathematischen Notation*, deren Semantik eindeutig definiert ist.[3)] Diese Eindeutigkeit ermöglicht es Modelle konsistent zu beschreiben und auf ihre Korrektheit zu überprüfen.

1.2 Allgemeines zur Spezifikationssprache Z

Der Name der *Z-Notation* leitet sich von Ernst **Z**ermelo ab, dem Begründer der axiomatischen Mengenlehre mit den Axiomen der Zermelo-Mengenlehre.

Z wurde Ende der 70er Jahre von Jean-Raymond Abrial mit Hilfe von Steve Schuman und Bertrand Meyer entwickelt. Die Weiterentwicklung fand durch die *Programming Research Group (PRG)* der Oxford University (unter Beteiligung von industriellen Partnern wie IBM) statt.[4)] Im Jahr 2002 wurde Z durch die International Organization for Standardization (ISO) standardisiert (ISO 13568).

Allgemein ausdrückt ist Z eine formale Sprache zur Beschreibung mathematischer Sachverhalte. Einsatz findet sie in erster Linie bei der formalen Spezifikation sequentieller Informationssysteme, das heißt bei der Beschreibung und Modellierung von Rechnersystemen (Hard- und Software).

---

1) Vgl. dazu ausführlich: Spivey, J.M. (1992)

2) Vgl. Spivey, J.M. (1992), S. 1

3) Vgl. Jacky, J. (1997), S. 4

4) Vgl. dazu ausführlich: http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/zstandards/index.html

Eine in Z verfasste Spezifikation ist eine Mischung aus formalen mathematischen Ausdrücken und informellem erläuterndem Text. Die formale Komponente liefert eine präzise Beschreibung des gewünschten Systemverhaltens, die informelle Komponente macht ein Z-Dokument für den Leser der Spezifikation zugänglicher und verständlicher.[5)]

Die Z-Notation beruht auf Modellannahmen: zur Modellierung eines Systems wird in der Regel sein *Zustand (state)* – eine Sammlung von Zustandsvariablen mit ihren Werten – abgebildet. Dieser Zustand kann durch *Operationen (operations)* verändert werden.[6)]

1.3 Grundkonzepte und mathematischer Hintergrund der Z-Notation

Wie in den vorhergehenden Kapiteln beschrieben, benutzt die formale Spezifikation eine mathematische Notation. Die Spezifikationssprache Z basiert auf den mathematischen Prinzipien der *Zermelo-Fraenkel-Mengenlehre (ZF)*[7)] und der *Prädikatenlogik erster Stufe*[8)] und verwendet auch große Teile dieser Notationen.

Ein weiteres elementares Konzept ist die *hierarchische (De-)Komposition durch die Verwendung von Schemata*. Ein komplexes System wird durch die Aufteilung und stückweise Darstellung einzelner Aspekte beschrieben. Diese Aspekte werden dann in Beziehung gesetzt und kombiniert. Durch diese einfache Form der Modularisierung können auch komplizierte Sachverhalte durch ihre einfachen Bestandteile dargestellt werden. Der Aufbau und die Verknüpfung von Schemata werden ausführlich in Kapitel 1.6 beschrieben.

1.4 Elemente in Z: Deklarationen, Ausdrücke und Prädikate

In diesem Abschnitt werden drei Grundkonstrukte von Z vorgestellt: *Deklarationen (declarations)*, *Ausdrücke (expressions)* und *Prädikate (predicates)*.

Deklarationen führen *Variablen* ein, Ausdrücke kennzeichnen die *Werte*, die diese Variablen annehmen können und Prädikate drücken die *Bedingungen* aus, die bestimmen welche Werte die Variablen tatsächlich haben.[9)]

---

5) Vgl. Sheppard, D. (1995), S. 175 f.

6) Vgl. Jacky, J. (1997), S. 31 f.

7) Vgl. dazu ausführlich: Ebbinghaus, H.-D. (2003)

8) Vgl. dazu ausführlich: Barwise J. / Etchemendy J. (2005)

9) Vgl. Jacky, J. (1997), S. 63

1.4.1 Mengen und Typen, Deklarationen und Variablen

Wie oben beschrieben, basiert die Spezifikationssprache Z auf dem mathematischen Prinzip der Zermelo-Fraenkel-Mengenlehre. Viele der hier beschriebenen Elemente sind deshalb sehr stark durch diese Mengenlehre beeinflusst, vor allem in ihrer Notation.

Bei der Modellierung von Systemen werden einzelne Informationsobjekte sinnvoll zusammengefasst. Eine solche Zusammenfassung kann als ***Menge*** aufgefasst werden. Mengen sind zentrale Elemente von Z. Eine Menge kann beispielsweise durch Aufzählung ihrer Elemente beschrieben werden:[10)]

$$\{1,2,4,8,16\}$$

Dabei spielt die Reihenfolge der Elemente keine Rolle, ebenso ist das redundante Aufzählen eines Elementes wirkungslos. Eine Menge mit keinen Elementen wird auch als *leere Menge* bezeichnet und mit dem Symbol Ø dargestellt.[11)]

Die folgenden beiden Beispiele zeigen eine Menge WUERFEL, mit den Elementen 1 bis 6 und eine Menge AMPEL mit den drei Elementen rot, gelb und grün.

$$WUERFEL == 1..6$$

$$AMPEL == \{rot, gelb, grün\}$$

Elemente müssen etwas gemeinsam haben, damit sie zu einer Menge zusammengefasst werden können. Man kann also sagen, sie müssen vom selben ***Typ*** sein. Das bedeutet, dass Z mit typisierten Mengen arbeitet. Z stellt per Definition lediglich einen Datentyp zur Verfügung: $\mathbb{Z}$ für ganze Zahlen.
Typen und Mengen sind sehr stark miteinander verwoben: Jeder Typ ist eine Menge (so steht zum Beispiel der Typ $\mathbb{Z}$ für die Menge aller Ganzzahlen), nicht jede Menge ist jedoch ein Typ (die Menge der natürlichen Zahlen $\mathbb{N}$ gehört zum Typ der Ganzzahlen, also zum Typ $\mathbb{Z}$). [12)]

Viel entscheidender ist in Z aber die Möglichkeit, Typen selbst zu definieren. So ist beispielsweise FARBEN ein *frei definierter Typ*, der durch eine Aufzählung aller möglichen Elemente bestimmt wird:

$$FARBEN ::== rot \mid grün \mid blau \mid gelb \mid cyan \mid magenta \mid weiss \mid schwarz$$

---

10) Vgl. Spivey, J.M. (1992), S. 25

11) Vgl. Jacky, J. (1997), S. 63

12) Vgl. Jacky, J. (1997), S. 64 f.

Neben der oben gezeigten freien Typdefinition gibt es die Möglichkeit der *Basistyp-Definition*. Dies macht immer dann Sinn, wenn im Vorfeld noch nicht alle Elemente angegeben werden sollen, etwa wenn die Menge zu viele Elemente beinhaltet. Wenn zum Beispiel ein Typ definiert werden soll, der alle Namen eines Telefonbuchs beinhaltet, ist es wenig sinnvoll alle möglichen Namen aufzuzählen. In diesem Fall kann mittels [NAME] ein Basistyp definiert werden, dessen Elemente nicht angegeben werden müssen.[13)]

Deklarationen führen neue ***Variablen*** ein und verknüpfen sie mit einer Menge bzw. einem Typ.[14)]
Im Folgenden werden einige beispielhafte Variablendeklarationen aufgeführt, um die Vorgehensweise zu skizzieren (erläuternde Kommentare sind in eckigen Klammern[15)] hinzugefügt):

$i : \mathbb{Z}$ [i ist eine Ganzzahl]
w_1, w_2 : WUERFEL [w_1 und w_2 sind zwei Zahlen im Bereich 1 bis 6]
signal : AMPEL [signal ist eine der Farben rot, gelb oder grün]

Neben einzelnen Werten können Variablen in Z aber auch ganze Mengen bezeichnen. In diesem Fall wird das Symbol ℙ (das mathematische Zeichen für die Potenzmenge) genutzt.
Demzufolge entspricht eine Variable kombination, die folgendermaßen in Abhängigkeit von der Menge AUSWAHL deklariert wird,

AUSWAHL == {X, Y, Z}
kombination : ℙ AUSWAHL

der Potenzmenge von AUSWAHL, also der Menge aller Teilmengen von AUSWAHL.
Mögliche Werte für kombination wären also z.B. {X}, {Y}, {Z}, {X, Y}, {X, Z}, {Y, Z} und {X, Y, Z} aber auch die leere Menge Ø.[16)]

---

13) Vgl. Jacky, J. (1997), S. 70 f.

14) Vgl. Spivey, J.M. (1988), S. 56

15) Anmerkung: Diese Form des Kommentierens entspricht nicht dem Z-Standard.

16) Vgl. Jacky, J. (1997), S. 69

1.4.2 Ausdrücke und Operatoren

Ausdrücke (expressions) kennzeichnen die Werte, die Variablen annehmen. Im vorherigen Kapitel wurden die möglichen Werte dadurch beschrieben, dass sie explizit aufgezählt wurden (z.B. {1, 2, 4, 8, 16} oder rot) oder dass Namen definiert wurden, die für bestimmte Werte stehen (z.B. signal oder WUERFEL). Ausdrücke erlauben somit Werte im Sinne von Namen und Literalen, die bereits definiert wurden, zu beschreiben.

Wesentlicher Bestandteil von Ausdrücken sind ***Operatoren***. In der Mathematik gibt es eine Vielzahl von Operatoren, über die Werte vorgegeben und verändert werden können. Aber wie können diese Operatoren in Z verwendet werden? Die geläufigsten Operatoren werden durch das *Mathematical Tool-kit* von Z[17] formalisiert und stehen somit zur Verfügung. Hier ein kurzer Auszug einiger Ausdrücke:[18]

Arithmetische Operatoren:

Das Mathematical Tool-kit definiert die üblichen arithmetischen Operatoren Addition (+), Subtraktion (-) und Multiplikation (*). Im Tool-kit ist keine Möglichkeit vorgesehen Gleitkommazahlen abzubilden, da reelle oder rationale Zahlen nicht definiert werden. Deshalb gibt es lediglich eine ganzzahlige Division div und mod zur Berechnung des ganzzahligen Restwerts aus der Division zweier ganzer Zahlen.

Beispiele:

$$5 * 3 = 15$$

$$12 \text{ div } 5 = 2$$

$$12 \text{ mod } 6 = 0$$

Mengenoperatoren

Da Mengen ein zentraler Bestandteil von Z sind, wird im Mathematical Tool-kit eine Vielzahl von Operatoren für sie definiert. So zum Beispiel der Vereinigungsoperator (∪), der Operator für die Schnittmenge (∩) und der Operator für die Differenz zwischen zwei Mengen (∖).

Beispiele:

$$\{1, 2, 3\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$$

$$\{1, 2, 3\} \cap \{2, 3, 4\} = \{2, 3\}$$

$$\{1, 2, 3, 4\} \setminus \{2, 3\} = \{1, 4\}$$

---

[17] Vgl. dazu ausführlich: Spivey, J.M. (1992), S. 86 ff.

[18] Vgl. Jacky, J. (1997), S. 72 ff.

1.4.3 Prädikate

Mit der Deklaration (siehe Kapitel 1.4.1) wird geregelt, welcher Menge oder welchem Typ eine Variable angehört. Damit ist schon die erste Bedingung für die Variable festgelegt (die „Mengenzugehörigkeit“). Durch das Formulieren weiterer Bedingungen wird die Menge der tatsächlich möglichen Zustände einer Variablen eingegrenzt. Hierbei bedient man sich des mathematischen Konzepts der Prädikatenlogik.
Ein Prädikat ist allgemein formuliert eine Folge von Wörtern mit „Platzhaltern“, die zu einer – wahren oder falschen – Aussage wird, wenn diese Platzhalter ausgefüllt werden. Prädikate sind daher in Z keine Ausdrücke, da sie die Werte nicht kennzeichnen, sondern Aussagen über Werte treffen.

Werden Prädikate bereits bei der Deklaration mit angegeben, geschieht dies gemäß nachstehendem Aufbau:

Deklarationen
Prädikate

Wird auf diese Weise eine Variablendeklaration global vorgenommen, das heißt so dass die Variable nach ihrer Deklaration im gesamten Z-Dokument verwendet werden kann, spricht man von einer *axiomatischen Definition*.[19)]
So könnte zum Beispiel eine Deklaration von zwei Variablen d_1 und d_2 mit einem Prädikat folgendermaßen aussehen:

$d_1, d_2 : WUERFEL$
$d1 + d2 = 7$

In der Deklaration wird festgelegt, dass d_1 und d_2 zwei Zahlen sind, die zur Menge WUERFEL gehören, das heißt es handelt sich um Ganzzahlen zwischen 1 und 6. Das Prädikat schränkt die zwei Zahlen weiter ein, so dass deren Summe 7 ergeben muss. Somit sind also nur noch $d_1 = 1$ und $d_2 = 6$ oder $d_1 = 2$ und $d_2 = 5$ usw. erlaubt, nicht jedoch $d_1 = 1$ und $d_2 = 5$ usw.. Weitere Prädikate (beispielshalber $d1 < d2$) können bei Bedarf zusätzlich mit aufgeführt werden.[20)]

[19)] Vgl. Sheppard, D. (1995), S. 188 f.

[20)] Vgl. Jacky, J. (1997), S. 74 ff.

Z stellt viele verschiedenartige Formen von Prädikaten zur Verfügung, wie etwa Quantoren (∀ und ∃), Verknüpfungen (∧ und ∨) oder Zugehörigkeit (∈ und ∉), um nur einige Beispiele zu nennen.[21)]

1.5 Strukturen in Z: Tupel, Relationen und Funktionen, Sequences und Bags

Nachdem im letzen Kapitel drei wesentliche Grundkonzepte von Z vorgestellt wurden, werden nun weitere wichtige Bestandteile erklärt, mit denen es möglich wird Datenstrukturen zu modellieren. Auch diese Strukturen sind wichtig für das Verständnis der Fallstudie in Kapitel 3.

1.5.1 Tupel, Relationen und Funktionen

Die Reihenfolge von Elementen in einer Menge spielt keine Rolle (siehe Kapitel 1.4.1). Es kann aber notwendig sein, dass individuelle Elemente aus verschiedenen Mengen miteinander verknüpft werden sollen. In diesem Fall muss eine Reihenfolge gegeben sein. Z realisiert dies über die Bildung von ***Tupeln***.

Nimmt man zum Beispiel eine Mitarbeiterdatenbank einer Firma, die mehrere Informationen (Name, Mitarbeiter-ID und Abteilung) über die erfassten Personen speichert, könnte die Deklaration wie folgt aussehen: [22)]

$$[NAME]$$

$$ID == \mathbb{N}$$

$$ABTEILUNG ::== Verwaltung \mid Produktion \mid Forschung$$

Über das *kartesische Produkt* kann nun ein Tupel gebildet werden, das alle drei Elemente beinhaltet:

$$MITARBEITER == ID \times NAME \times ABTEILUNG$$

Somit wäre nun folgende Variablendeklaration möglich:

$$Meier, Schmid : MITARBEITER$$

$$Meier = (110, Klaus\ Meier, Produktion)$$

$$Schmid = (198, Elke\ Schmid, Verwaltung)$$

21) Vgl. dazu ausführlich: Spivey, J.M. (1992), S. 67 ff.

22) Vgl. Jacky, J. (1997), S. 78 f.

Ein Tupel ist folglich eine Instanz des kartesischen Produkts von Mengen.[23] Wichtig ist dabei, dass die Reihenfolge in Tupeln im Gegensatz zu Mengen eine Bedeutung spielt, das heißt die Mengen $\{1, 2, 3\}$ und $\{3, 1, 2\}$ sind gleich, die Tupel $(1, 2, 3)$ und $(3, 1, 2)$ hingegen nicht.[24]

In der Regel arbeitet man nicht mit einzelnen Tupeln, sondern mit einer Menge von Tupeln. Eine solche Menge wird als ***Relation*** bezeichnet. Mathematische Relationen werden dazu benutzt, um Beziehungen zwischen Daten auszudrücken. Somit lassen sich Tabellen und Datenbanken mittels Relationen modellieren.[25]

Ein Auszug aus einer Tabelle in der Mitarbeiterdatenbank aus vorigem Beispiel könnte folgendermaßen aussehen:[26]

ID	NAME	ABTEILUNG
110	Klaus Meier	Produktion
198	Elke Schmid	Verwaltung
325	Herbert Knebel	Forschung

Die entsprechende Z-Notation hätte folgenden Aufbau:

$$
\begin{array}{|l}
\hline
mitarbeiter : \mathbb{P}\ MITARBEITER \\
\hline
mitarbeiter = \{ \\
\quad (110, Klaus\ Meier, Produktion), \\
\quad (198, Elke\ Schmid, Verwaltung), \\
\quad (325, Herbert\ Knebel, Forschung), \\
\quad \} \\
\end{array}
$$

In dem Beispiel ist zu erkennen, dass die Werte innerhalb der Tupel eine bestimmte Reihenfolge haben müssen, die Reihenfolge der Tupel allerdings keine Rolle spielt, da die Relation, wie oben beschrieben, lediglich eine *Menge von Tupeln* darstellt.

---

[23] Vgl. Spivey, J.M. (1992), S. 25 f.

[24] Vgl. Sheppard, D. (1995), S. 93

[25] Vgl. Sheppard, D. (1995), S. 103 ff.

[26] Vgl. Jacky, J. (1997), S. 80

Eine besondere Bedeutung kommt in diesem Zusammenhang *Paaren* und *binären Relationen* zu. *Paare* sind Tupel mit nur zwei Komponenten, etwa ein Name und eine Telefon-Durchwahl: $(\text{Klaus Meier}, 133)$. Hierfür stellt Z eine eigene Syntax zur Verfügung: $\text{Klaus Meier} \mapsto 133$. Der Pfeil dient dabei sozusagen als Operator zum Bilden eines Paares.
Binäre Relationen sind eine Menge von Paaren. Auch hier hat Z eine alternative Darstellung: Statt $\mathbb{P}\ (\text{NAME} \times \text{DURCHWAHL})$ kann auch $\text{NAME} \leftrightarrow \text{DURCHWAHL}$ formuliert werden.[27)]

Funktionen sind eine besondere Art von Relationen: Funktionen sind binäre Relationen, bei denen ein Element nur einmal als das erste Element eines Paares auftauchen darf. Man kann also sagen, dass jedes Element der „Grundmenge" höchstens einem Element in der „Wertemenge" zugeordnet werden darf. Vereinfacht formuliert sind Relationen n:n-Beziehungen, Funktionen dagegen können weder n:n- noch 1:n-Beziehungen sein. Vielmehr handelt es sich dabei um n:1- oder 1:1-Beziehungen.[28)] Diese sehr generell gehaltene Feststellung wird in Kapitel 4 genauer differenziert.

Angenommen in dem obigen Beispiel mit der Zuordnung einer Telefon-Durchwahl zu einem Namen wird festgelegt, dass jeder Mitarbeiter maximal eine Durchwahl hat (das heißt auch jeder Name kommt nur einmal vor), teilweise aber mehrere Mitarbeiter unter einer Durchwahl erreichbar sind, dann lässt sich das in Z wie folgt modellieren:

$$\text{telefon}_F : \text{NAME} \nrightarrow \text{DURCHWAHL}$$

$$\text{telefon}_F = \{$$
$$\text{Klaus Meier} \mapsto 133.$$
$$\text{Elke Schmid} \mapsto 181.$$
$$\text{Herbert Knebel} \mapsto 133.$$
$$\}$$

Die Funktion telefon_F nutzt den Funktionspfeil $\nrightarrow$. Dadurch wird die Asymmetrie der funktionalen Relation ausgedrückt: Jeder Mitarbeiter kann nur eine Durchwahl haben, eine Durchwahl kann aber mehreren Mitarbeitern zugeordnet sein.

[27)] Vgl. Jacky, J. (1997), S. 81 f.

[28)] Vgl. Jacky, J. (1997), S. 88 ff.

Der Funktionspfeil $\nrightarrow$ drückt somit aus, dass es sich bei $telefon_F$ um eine *partielle Funktion* handelt. NAME ist die Menge aller möglichen Namen, allerdings werden die meisten Namen nicht im Telefonverzeichnis der Firma eingetragen sein. Eine partielle Funktion muss also nicht jedem möglichen Element der „Grundmenge“ ein Element in der „Wertemenge“ zuordnen.
Wenn jedes Element in der „Grundmenge“ mit einem Element in der „Wertemenge“ verknüpft wird, spricht man von einer *totalen Funktion*. Der Z-Funktionspfeil für totale Funktionen ist nicht durchgestrichen: $\longrightarrow$ Ein Beispiel für eine totale Funktion wäre evtl. in einer Lagerverwaltung, bei der jeder verfügbaren Produktnummer ein Lagerbestand zugeordnet wird.[29)]
Die verschiedenen Typen von Funktionen in der Z-Notation werden ausführlich in Kapitel 4 beschrieben.

1.5.2 Sequences und Bags

Bisher wurden mathematische Strukturen vorgestellt, die es erlauben Sammlungen von Elementen als Mengen und Beziehungen zwischen ihnen als Relationen und Funktionen zu modellieren. Wie bereits erwähnt ist bei Mengen die Reihenfolge der Elemente unerheblich. Bei der Abbildung von Daten ist es aber in vielen Fällen notwendig eine Reihenfolge vorzugeben. [30)]
Diese Form der „geordneten Menge“ wird in Z mittels ***Sequences*** realisiert. Sequenzen können Arrays, Listen, Queues und andere sequentielle Strukturen abbilden.

In einem Beispiel sollen zwei Sequences definiert werden: wochentag und wochenende.[31)]
Dazu müssen zuerst alle Wochentage deklariert werden:

$$TAGE ::== Dienstag \mid Freitag \mid Montag \mid Mittwoch \mid Samstag \mid Donnerstag \mid Sonntag$$

In dieser Deklaration ist keine Reihenfolge vorgegeben. Deshalb definiert man zwei Sequences:

$$wochentag : seq\ TAGE$$
$$wochentag = \langle Montag, Dienstag, Mittwoch, Donnerstag, Freitag \rangle$$

$$wochenende: seq\ TAGE$$
$$wochenende = \langle Samstag, Sonntag \rangle$$

[29)] Vgl. Sheppard, D. (1995), S. 118 f.

[30)] Vgl. Sheppard, D. (1995), S. 143 ff.

[31)] Vgl. Jacky, J. (1997), S. 93 f.

In dem Beispiel lässt sich erkennen, dass eine Sequenz durch das Schlüsselwort seq definiert wird. Die Elemente der Sequenz sind von angewinkelten Klammern ⟨…⟩ umschlossen.
Verschiedene Operatoren können auf Sequences angewendet werden. So können mit dem Verknüpfungsoperator ⌢ zwei Sequences zu einer zusammengefügt werden, indem das zweite Argument an das erste angehängt wird:

$$woche == wochentag \frown wochenende$$

In diesem Fall würde woche alle Wochentage in der Reihenfolge von Montag bis Sonntag beinhalten.
Weitere Operatoren sind beispielsweise head, der das erste Element einer Sequenz ermittelt, oder tail, welcher alle Elemente außer dem Ersten liefert.[32)]

Eine weitere Besonderheit stellen ***Bags*** dar. Eine Bag voller Elemente ist ähnlich zu einer Menge, allerdings hat die Anzahl des Auftretens eines Elements eine Bedeutung. Die Reihenfolge dagegen ist irrelevant. Bags finden Anwendung wenn die Vervielfältigung eines Elements auftreten kann.[33)]

So könnte etwa eine Bag abschlussnoten folgendermaßen aussehen:

$$abschlussnoten : bag \mathbb{N}$$
$$abschlussnoten == [[1, 3, 5, 3, 2, 1, 1, 5, 3, 4, 2, 2, 3, 4]]$$

Bags werden durch das Schlüsselwort bag definiert, die Elemente werden von den Klammern [[…]] umschlossen.
Auch für Bags gibt es eigene Operatoren, wie etwa count, der ermittelt wie oft ein Element in einer Bag enthalten ist.[34)]

Zusammenfassend lässt sich der Unterschied zwischen Mengen, Sequences und Bags so skizzieren:

- Mengen: $\{1, 2, 2, 2\} = \{1, 2, 2\} = \{2, 1, 2\} = \{1, 2\} = \{2, 1\}$
- Bags: $[[1, 2, 2, 2]] \neq [[1, 2, 2]] = [[2, 1, 2]] \neq [[1, 2]] = [[2, 1]]$
- Sequences: $\langle 1, 2, 2, 2 \rangle \neq \langle 1, 2, 2 \rangle \neq \langle 2, 1, 2 \rangle \neq \langle 1, 2 \rangle \neq \langle 2, 1 \rangle$

---

32) Vgl. dazu ausführlich: Spivey, J.M. (1992), S. 115 ff.

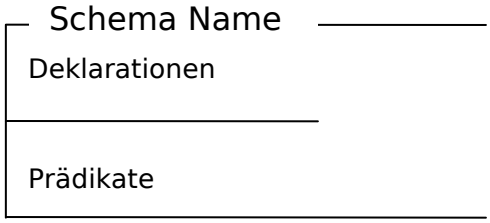
33) Vgl. Sheppard, D. (1995), S. 154 ff.

34) Vgl. dazu ausführlich: Spivey, J.M. (1992), S. 124 ff.

1.6 Z Schema und Schema Calculus

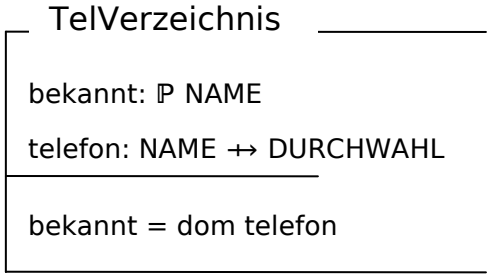
Wie bereits in Kapitel 1.3 kurz angedeutet, stellt Z mit dem Konzept der ***Schemata*** ein Mittel zur Verfügung, mit dem komplexe Sachverhalte in mehrere kleinere und besser handhabbare Aspekte aufgeteilt werden können. Dies hat den Vorteil, dass man sich auf die Spezifikation eines überschaubaren Bereichs konzentrieren kann ohne ständig den Blick „fürs große Ganze“ haben zu müssen.[35] Konkret heißt das, Schemata helfen bei der Modellierung von Rechnersystemen, denn sie erlauben einzelne Zustände und Zustandsänderungen zu modellieren. [36]

Die grundsätzliche Notation eines Schemas sieht wie folgt aus:



Die verschiedenen Formen von Schemata lassen sich am besten an einem kleinen Beispiel darstellen. Nehmen wir an, eine Datenbank mit den Durchwahlen aller Mitarbeiter eines Unternehmens (siehe Beispiel in Kapitel 1.5.1) soll modelliert werden.

Dazu müssten zuerst die zwei Basisdatentypen [NAME] und [DURCHWAHL] definiert werden. Dann würde das erste Schema, das *Zustandsschema*, so modelliert werden:



Das Zustandsschema besagt, dass der Zustand von TelVerzeichnis aus einer Anzahl Namen (bekannt) und einer partiellen Funktion telefon besteht. Das Prädikat sagt aus, dass die Menge von bekannt dieselbe sein muss, wie der Wertebereich (dom)[37] der Funktion telefon. Der Operator *dom* wird detailliert in Kapitel 4.1 beschrieben.

---

[35] Vgl. Sheppard, D. (1995), S. 192 ff.

[36] Vgl. Jacky, J. (1997), S. 122 ff.

[37] Vgl. dazu ausführlich: Sheppard, D. (1995), S. 109 f.

Als nächstes wird der initiale Zustand unseres Telefonverzeichnisses definiert:

Init

TelVerzeichnis

---

bekannt = ∅

Der Startzustand des Systems ist also dadurch gekennzeichnet, dass bekannt eine leere Menge ist, das heißt im System sind noch keine Einträge. Das *Initialisierungsschema* beinhaltet die *Schema Referenz* TelVerzeichnis. Das bedeutet, dass Init alle Deklarationen und Prädikate von TelVerzeichnis inkludiert.

Die nächste Form sind *Operationsschemata*, über die Zustandsänderungen modelliert werden.
Im Telefonverzeichnis-Beispiel soll eine Operation definiert werden, über die ein neuer Eintrag hinzugefügt werden kann:

NeueDurchwahl

Δ TelVerzeichnis

name? : NAME

durchwahl? : DURCHWAHL

---

name? ∉ bekannt

telefon' = telefon ∪ {name? ↦ durchwahl?}

Durch die Schema Referenz Δ TelVerzeichnis wird angezeigt, dass es sich um eine *Änderungsoperation* handelt: Der Zustand von TelVerzeichnis wird verändert. Bei den Deklarationen zeigt das Fragezeichen (?) an, dass es sich um Eingabevariablen handelt.
Das Prädikat name? ∉ bekannt beschreibt die *Vorbedingung*, die erfüllt sein muss, damit die Operation ausgeführt werden kann. In diesem Fall heißt das, dass der Name, der hinzugefügt werden soll, dem System noch nicht bekannt sein darf. Das Hochkomma (') drückt aus, dass telefon' den Zustand nach der Operation beschreibt. Die *Nachbedingung* telefon' = telefon ∪ {name? ↦ durchwahl?} sagt demzufolge aus, dass telefon nach der Operation um das Paar aus name und durchwahl erweitert wird.

Eine weitere gewünschte Operation im Telefonverzeichnis ist das Finden einer Durchwahl zu einem Namen. Auch dies wird mit einem Operationsschema modelliert:

FindeDurchwahl

Ξ TelVerzeichnis

name? : NAME

durchwahl! : DURCHWAHL

---

name? ∈ bekannt

durchwahl! = telefon (name?)

Das Symbol Ξ in der Schema Referenz indiziert, dass es sich hier um eine Operation handelt, die den Zustand von TelVerzeichnis nicht ändert. Es wird lediglich ein Name ausgelesen. Das Ausrufezeichen (!) bei der Deklaration kennzeichnet durchwahl als Ausgabevariable.
Die Vorbedingung sagt hier aus, dass der eingegebene Name dem System bekannt sein muss. Wenn diese Bedingung erfüllt ist, wird über die telefon-Funktion die zugehörige Durchwahl ermittelt.

Bei den beiden beschrieben Operation handelt es sich momentan noch um *partielle Operationen*. Partiell deshalb, da bisher nicht alle möglichen Fälle abgedeckt wurden.[38)]
Bisher ist mit dem Schema NeueDurchwahl nicht geklärt, was passiert wenn die Vorbedingung name? ∉ bekannt nicht erfüllt wird. Erst wenn dieser Fall auch abgedeckt ist, kann eine *totale Operation* formuliert werden.

Dazu wird zuerst ein freier Typ MELDUNG ::= ok | schon bekannt definiert. Damit lässt sich anschließend ein Schema Erfolg für den Erfolgsfall spezifizieren:

Erfolg

ergebnis! : MELDUNG

---

ergebnis! = ok

38) Vgl. Sheppard, D. (1995), S. 127

Für den Fall, dass eine Durchwahl zu einem Namen, der dem System schon bekannt ist, eingeben wird, folglich also die Vorbedingung $name? \notin bekannt$ nicht erfüllt ist, müsste das zugehörige Schema BereitsBekannt diesen Aufbau haben:

BereitsBekannt

Ξ TelVerzeichnis

name? : NAME

ergebnis! : REPORT

---

name? $\in$ bekannt

ergebnis! = schon bekannt

Für die Operation FindeDurchwahl bzw. deren Vorbedingung $name? \in bekannt$ muss dieser Schritt entsprechend durchgeführt werden.

Der Vorteil der Aufteilung in einzelne Aspekte führt unweigerlich dazu, dass die Einzelteile zu einem bestimmten Zeitpunkt wieder zu einer großen Gesamt-Spezifikation zusammengeführt werden müssen. Das Manipulieren und Zusammenfügen von Schemata erfolgt in Z mittels des ***Schema Calculus***.[39)]

Schema Calculus stellt verschiedenste *Operatoren* zur Verfügung. Die zwei Wichtigsten sind *Schema Konjunktion*, zur Kombination von Anforderungen, und *Schema Disjunktion*, um Alternativen abzubilden. Schema Konjunktion nutzt den logischen Verknüpfungsoperator UND ($\wedge$), Schema Disjunktion den Verknüpfungsoperator ODER ($\vee$).[40)]

Im obigen Beispiel wurden für die Operation „Hinzufügen einer Durchwahl“ mehrere Schemata definiert. Diese lassen sich nun über die Operatoren des Schema Calculus kombinieren.

$$T\_NeueDurchwahl \mathrel{\widehat{=}} (NeueDurchwahl \wedge Erfolg) \vee BereitsBekannt$$

Auf der linken Seite dieser *Schema Calculus Formel* steht der Name des neuen Schemas, rechts steht ein *Schema Ausdruck*, bestehend aus Schema Namen bzw. Schema Referenzen und Schema Operatoren.

---

39) Vgl. Sheppard, D. (1995), S. 200 ff.

40) Vgl. Jacky, J. (1997), S. 127 ff.

Somit erhält man eine totale Operation T_NeueDurchwahl, die alle möglichen Fälle abdeckt. Trotzdem wurden die einzelnen Aspekte separat modelliert und erst abschließend zusammengefasst. Prinzipiell hätte man auch nur ein Schema T_NeueDurchwahl formulieren können, dass alle Deklarationen und Prädikate enthält. Das wäre aber sicherlich komplexer und unübersichtlicher geworden als die Modellierung und Verknüpfung der einzelnen Aspekte.

Anzumerken bleibt, dass diese Modularisierung durch Schemata bei der späteren Implementierung nicht analog erfolgen muss. Bei der Umsetzung in Programmcode kann es durchaus Sinn machen, bestimmte Teile nicht so stark aufzutrennen oder unter Umständen manche Aspekte noch feingranularer zu gestalten.

1.7 Struktur eines Z-Dokuments

Ein Z-Dokument besteht aus einer Mischung von formalen Z-Formulierungen und informellen beschreibenden Texten. Die hier vorgestellte Struktur ist lediglich als Vorschlag zu sehen und nicht als verbindlicher Standard.

Die vorgeschlagene Dokumentenstruktur beinhaltet folgende Abschnitte in dieser Reihenfolge: [41)]

1.) **Textuelle Einleitung**
   Eine umfassende informelle Ausführung des Zwecks und der Ziele des spezifizierten Systems.
2.) **Einführung in die relevanten Datenarten und globale Variablen, die in der gesamten Spezifikation benutzt werden**
   Die gegebenen Mengen repräsentieren die Datenarten, die das zu beschreibende Software Engineering Problem charakterisieren. Diese Mengen werden in der gesamten Spezifikation verwendet und müssen daher früh im Dokument eingeführt werden. Gleich verhält es sich mit globalen Variablen und Konstanten. Zum Beispiel findet in diesem Bereich die Definition der Basistypen statt (siehe Kapitel 1.4.1).
3.) **Darlegung der allgemeinen Theorien, die in der Spezifikation verwendet werden und für das Verständnis des Dokuments notwendig sind**
   Bestimmte spezielle Annahmen und Theorien bedürfen der Erklärung. Dies sollte früh im Dokument erfolgen.

---

41) Vgl. Sheppard, D. (1995), S. 178 ff.

4.) **Vorstellung und Beschreibung des abstrakten Zustands des Systems**

Als erster Aspekt wird der Zustandsraum des Systems in einem Schema beschrieben.

5.) **Beschreibung der geeigneten Initialisierungen des Systems**

Bevor Operationen den Zustand eines Systems verändern können, muss ein Anfangszustand spezifiziert werden.

6.) **Definition des Verhaltens der Operationen unter „normalen" Bedingungen**

Zuerst wird nur der „Gutfall" der Operationen beschrieben, das heißt es wird von einer Erfüllung der Vorbedingungen ausgegangen. Die Beschreibung der Fehlerfälle erfolgt in Z separat (siehe die beiden folgenden Abschnitte).

7.) **Ableiten der Vorbedingungen für alle beschriebenen Operationen**

Ausnahmen (exceptions) treten auf, wenn Vorbedingungen nicht erfüllt werden. In diesem Abschnitt werden die Vorbedingungen der in Schritt 6 definierten Operationen ermittelt und geordnet.

8.) **Detaillieren der Fehlerbedingungen, die für die definierten Operationen auftreten können, unter Berücksichtigung von Abschnitt 7**

Vorbedingungen beschreiben die Gegebenheiten, unter denen Operationen rechtmäßig durchgeführt werden. Die Umkehrung der Vorbedingungen beschreiben folglich die Fälle, die als Ausnahmen behandelt werden müssen. Erst dadurch wird aus einer partiellen eine totale Operation (siehe Kapitel 1.6).

9.) **Erstellen einer Zusammenfassung und eines Index für die Spezifikation**

Ab einer bestimmten Größe des Dokuments sollten dem Leser Übersichten wie eine Zusammenfassung und ein Index zur Verfügung gestellt werden.

2. Relevante Modellelemente der Unified Modelling Language (UML)

In diesem Kapitel werden einige Bestandteile der *Unified Modelling Language (UML)* vorgestellt, die für das Verständnis der Modellierung in Kapitel 3 hilfreich sind.

Die Einführung wird sehr viel knapper ausfallen, als die Vorstellung der formalen Spezifikation mit der Z-Notation im vorhergehenden Kapitel. An dieser Stelle wird davon ausgegangen, dass der Leser bereits Erfahrung mit der UML gesammelt hat und deshalb nur eine Erläuterung einiger weniger Besonderheiten notwendig ist. Detaillierte Informationen zur Unified Modelling Language in der Version 2 finden sich in zahlreichen Fachbüchern.[42)]

Auch UML gehört grundsätzlich zu den Spezifikationssprachen, wie sie in Kapitel 1.1 beschrieben wurden. Allerdings arbeitet UML nicht mit einer mathematischen sondern in erster Linie mit einer *visuellen Notation*. Damit wird es möglich, komplexe Software- und andere Systeme zu modellieren, zu dokumentieren und zu visualisieren, unabhängig von einer speziellen Programmiersprache.[43)] Allerdings muss der Vollständigkeit halber erwähnt werden, dass UML mit einem *objektorientierten Hintergrund* entwickelt wurde.

In den folgenden beiden Kapiteln werden nun zwei Modellelemente von UML vorgestellt: *Aufzählungstypen (Enumerations)* und *Zustandsautomaten (State machines).*

2.1 Aufzählungstyp (Enumeration)

Ein *Aufzählungstyp (Enumeration)* ist ein spezieller Datentyp mit einer begrenzten Menge benutzerdefinierter Werte. Ein solcher Wert nennt sich *Aufzählungswert (EnumerationLiteral)* und spezifiziert einen Wert, den ein Element mit dem Aufzählungstyp zur Laufzeit annehmen darf.[44)]

Der Aufzählungstyp ist in der UML-Superstructure folgendermaßen definiert:
„An enumeration is a data type whose values are enumerated in the model as enumeration literals. […] An enumeration literal is a user-defined data value for an enumeration."[45)]

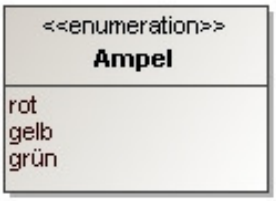
[42)] Vgl. dazu ausführlich: Miles, R. / Hamilton, K. (2006) und Jeckle, M. / Rupp, C. et al. (2005)

[43)] Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 12

[44)] Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 46

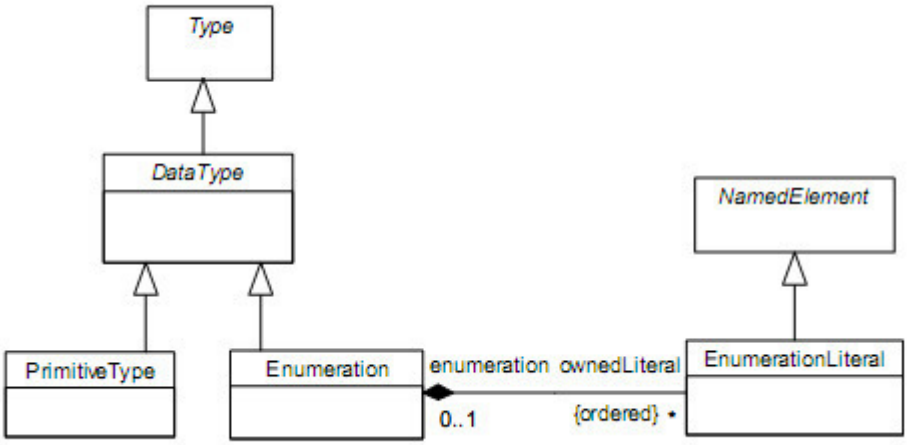
[45)] OMG (Hrsg.) (2005), S. 63 f.

Die Notation eines Aufzählungstyps zeigt das folgende Beispiel:



Unter dem Schlüsselwort <<*enumeration*>> steht der Name (*Ampel*) des Aufzählungstyps. Die drei Aufzählungswerte (*rot, gelb, grün*) werden in einem separaten Abschnitt unterhalb des Namens angegeben.

Das Metamodell ordnet die Klassen Enumeration und EnumerationLiteral folgendermaßen ein:[46)]



Aus der Generalisierungsbeziehung lässt sich erkennen, dass Enumeration ein spezieller Datentyp ist. Ein Aufzählungstyp kann beliebig viele EnumerationLiterals umfassen, die geordnet sind.

2.2 Zustandsautomat (State machine)

Zustandsautomaten (State machines) bieten die Möglichkeit, das Verhalten eines Systems zu modellieren. Sie spezifizieren dieses Verhalten mittels *Zuständen (States)* und *Übergängen (Transitions)* zwischen diesen Zuständen, die durch interne oder externe *Ereignisse (Events)* initiiert werden.

Ein Zustandsautomat beschreibt demzufolge, wie sich das System in einem bestimmten Zustand bei gewissen Ereignissen verhält.[47)]

[46)] Entnommen aus: OMG (Hrsg.) (2003), S. 100

[47)] Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 335

Bei der Modellierung von Zustandautomaten werden zwei vereinfachende Annahmen getroffen:[48)]

- Das System befindet sich zu einem bestimmten Zeitpunkt exakt in einem *Zustand (State)*
- Der *Übergang (Transistion)* von einem Zustand in den nächsten erfolgt ohne jegliche zeitliche Verzögerung

Im Folgenden werden nun die wichtigsten Notationselemente zur Modellierung von Zustandsautomaten vorgestellt:

Ein ***Zustand (State)*** bildet eine Situation ab, in deren Verlauf eine spezielle Bedingung gilt. Dabei kommen folgende Regeln zum Tragen:[49)]

- Ein Zustand wird betreten, wenn eine Transition, die ihn als Endpunkt besitzt, durchlaufen wird.
- Ein Zustand wird verlassen, wenn eine vom Zustand wegführende Transition durchlaufen wird.
- Durch Betreten des Zustands wird er *aktiv*, durch Verlassen *inaktiv*.
- Sofort nach Betreten wird das *Eintrittsverhalten (Entry Behaviour)* des Zustands ausgeführt. Beim Verlassen wird entsprechend das *Austrittsverhalten (Exit Behaviour)* ausgeführt.
- Das *Zustandsverhalten (State Behavior)* eines Zustands ist das Verhalten, das nach Beendigung des Eintrittsverhaltens aufgerufen wird.

Die Definition von Zustand in der UML-Superstructure lautet:

“A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. […].”[50)]

Die Notation zeigt ein Rechteck mit abgerundeten Ecken, das den Zustandsnamen beinhaltet:[51)]

In einem abgetrennten Abschnitt können Eintrittsverhalten (*entry*), Zustandsverhalten (*do*) und Austrittsverhalten (*exit*) angegeben werden.

Wird der Name weggelassen, handelt es sich um einen *anonymen Zustand*.

---

48) Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 336

49) Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 344

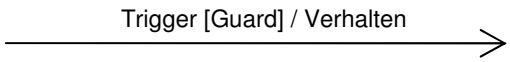
50) OMG (Hrsg.) (2005), S. 531

51) Vgl. OMG (Hrsg.) (2005), S. 537 ff.

Eine ***Transition*** schafft den Übergang von einem Ausgangs- zu einem Zielzustand. Sie verbindet dadurch einen Quell- und einen Zielknoten.

Die exakte Definition besagt:
„A transition is a directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type." [52)]

Dargestellt wird eine Transition durch eine durchgezogene, gerichtete und im Normalfall beschriftete Kante:[53)]



Die Beschriftung beinhaltet folgende Elemente:

- *Trigger*: der Auslöser für die Transition
- *Guard*: eine Bedingung, die wahr sein muss, damit die Transition durchlaufen werden kann
- *Verhalten*: der Name des Verhaltens, das beim Durchlaufen der Transition ausgeführt wird

Eine besondere Form von Zuständen definiert die UML mit den ***Pseudozuständen (pseudostates)***. Damit wird es möglich, komplexe Beziehungen zwischen Zuständen einfach darzustellen.

In der UML-Superstructure wird der Pseudozustand folgendermaßen definiert:
„A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph. […] Pseudostates are typically used to connect multiple transitions into more complex state transitions paths. […]"[54)]

Neben *Entscheidungen (Choices)*, *Gabelung und Vereinigung (Forks and Joins)*, *Terminatoren (Terminate pseudostate)* und *Eintritts- und Austrittspunkten (Entry- / Exit points),* gibt es Pseudozustände mit besonderer Bedeutung: der *Startzustand (Initial pseudostate)* und *Kreuzungen (Junctions)*.[55)]

[52)] OMG (Hrsg.) (2005), S. 553

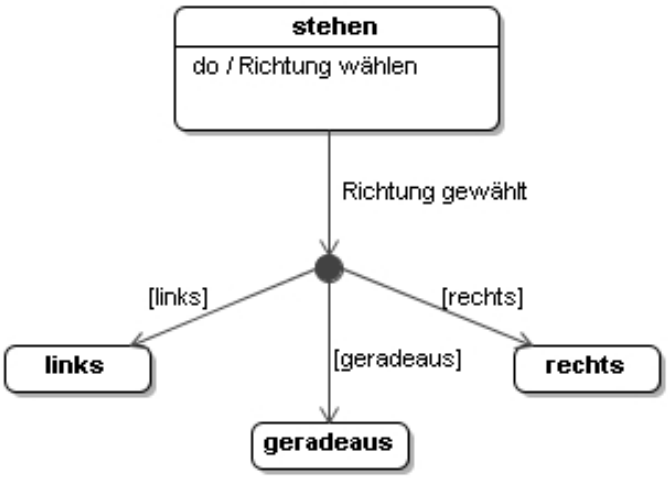
[53)] Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 346

[54)] OMG (Hrsg.) (2005), S. 522

[55)] Vgl. dazu ausführlich : Jeckle, M. / Rupp, C. et al. (2005), S. 355 ff.

Kreuzungen (Junctions) ermöglichen es, verschiedene Transitionen ohne verbindende Zustände hintereinander zu schalten. Dabei muss aber bei jeder Kreuzung mindestens eine eingehende und eine ausgehende Transition angebracht sein. Bei ausgehenden Transitionen können Bedingung hinzugefügt werden.

Die Notation wird im folgenden Beispiel deutlich: [56)]



Der *Startzustand (Initial pseudostate)* bildet den Startpunkt für das Betreten eines Zustandsautomaten. Zu einem Startzustand kann kein Übergang stattfinden, dass heißt er darf keine eingehenden Transitionen besitzen.[57)]

Die Notation des Startzustands ist ein ausgefüllter Kreis:

Ein weiterer besonderer Zustand ist der *Endzustand (Finale state)*. Wird er erreicht, ist die Abarbeitung des Zustandsautomaten beendet. Folglich gibt es bei Endzuständen keine ausgehenden Transitionen.[58)]

Der Endzustand wird als kleiner ausgefüllter Kreis, umgeben von einem unausgefüllten Kreis dargestellt:

[56)] Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 357 ff.

[57)] Vgl. Oesterreich, B. (1998), S. 312 f.

[58)] Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 354

3. Fallstudie „A Drinks Dispensing Machine“

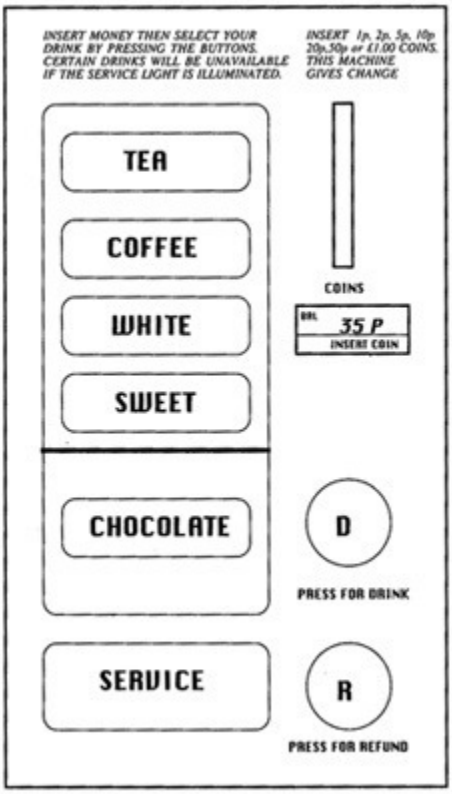
Nachdem die Z-Notation und die besonderen Modellelemente der UML vorgestellt wurden, wird nun eine Fallstudie behandelt. Dabei handelt es sich um die Case Study „A Drinks Dispensing Machine“ aus dem Buch *„An Introduction to Formal Specification with Z and VDM“* von Deri Sheppard.[59)]
Der Buchauszug mit der vollständigen Z-Spezifikation findet sich in Anlage 1 im Anhang dieser Arbeit.

Im ersten Teil dieses Kapitels wird ein kurzer Überblick über die Fallstudie gegeben, bevor dann im zweiten Abschnitt Teile der „Drinks Dispensing Machine“ mit UML modelliert werden.

3.1 Vorstellung der Fallstudie

An dieser Stelle kann nur eine verkürzte informelle Zusammenfassung der Spezifikation wiedergegeben werden, die Einzelheiten sollten zum Verständnis im Buchauszug in Anlage 1 nachgelesen werden.

Die mit der Z-Notation spezifizierte Fallstudie beschreibt einen Getränkeautomaten für Heißgetränke, dessen Bedienfeld die folgende Abbildung[60)] zeigt (eine größere Darstellung findet sich in Anlage 1):



59) Vgl. dazu ausführlich: Sheppard, D. (1995), S. 271-285

60) Entnommen aus: Sheppard, D. (1995), S. 272

Die Struktur des Z-Dokuments entspricht größtenteils dem in Kapitel 1.7 beschriebenen Vorschlag. Die einzelnen Abschnitte werden im Folgenden genauer erläutert:

1.) Textuelle Einleitung

Neben der Darstellung des Bedienfelds (siehe oben) wird beschrieben, dass in den Automaten solange Geld eingeworfen wird, bis der Mindestbetrag erreicht ist. Die Getränkeauswahl findet durch Kombination der Auswahltasten statt. Anschließend erfolgt die Getränkeausgabe durch Drücken der Taste *P*. Falls eine Ausgabe des gewählten Getränks nicht möglich ist, kann über die Taste *R* das eingeworfene Geld zurückgefordert werden.

Die Anzeigen des Automaten sind das *Service-Display*, das anzeigt ob der Automat gewartet werden muss (wenn Zutaten für ein Getränk aufgebraucht oder keine Becher mehr vorhanden sind), das *Coin-Display*, das den aktuellen Wert der eingeworfenen Münzen anzeigt, und das *Report-Display*, das Anleitungen für den Kunden ausgibt.

Des Weiteren ist ein Hinweis zu finden, dass die Fallstudie *Bags* (siehe Kapitel 1.5.2) benutzt.

2.) Einführung von gegebenen Mengen, Abkürzungen und axiomatischen Definitionen

Hier werden alle global benötigten einfachen Typen definiert, zum Beispiel die Auswahltasten auf dem Bedienfeld (*Selection_buttons*) und die Getränkezutaten (*Ingredient*). Der Datentyp für die möglichen Anzeigen des Service-Displays wird deklariert (*Onoff*), und ein Datentyp für alle Münzen (*Coin*) wird eingeführt.

Aus *Coin* ergibt sich schließlich die Menge aller akzeptierten britischen Münzen (*British_Coin*). Die Menge der Getränke (*Drink*) wird durch die Kombination der Auswahltasten (d.h. aus *Selection_buttons*) repräsentiert. Die Zutatenliste (*List_of_ingredients*) für alle definierten *Drinks* ergibt sich in Abhängigkeit von *Ingredient*.

Abschließend werden in dem Abschnitt noch global benötigte Funktionen und Konstanten definiert:[61)] Für die Zuordnung eines bestimmten Getränks (*Drink*) zu einer Menge von benötigten Zutaten (*List_of_ingredients*) wird *Recipe* als totale Injektion eingeführt.

Die Zuordnung einer britischen Münze (*British_Coin*) zu ihrem Wert repräsentiert die Funktion *Worth*. Zusätzlich gibt es eine totale Funktion *HopperMax*, die ermittelt, wie viele Münzen einer Stückelung maximal in einem Münz-Füllschacht des Automaten enthalten sein können. Entsprechend gibt es eine totale Funktion *StockMax*, die jeder Zutat die maximale Anzahl der Einheiten (Teebeutel, Zuckerwürfel, usw.), die der Automat bevorraten kann, zuordnet.

---

61) Vgl. dazu ausführlich: Sheppard, D. (1995), S. 273 ff.

CupMax enthält die maximale Anzahl an Plastikbechern, die der Automat enthalten kann. Alle beschriebenen Funktionen werden in einer (globalen) axiomatischen Definition ausgedrückt:

$$
\begin{array}{l}
Worth : British\_coin \rightarrowtail \mathbb{N} \\
HopperMax : British\_Coin \rightarrow \mathbb{N} \\
StockMax : Ingredients \rightarrow \mathbb{N} \\
CupMax : \mathbb{N} \\
\hline
Worth = \{One\_Penny \mapsto 1, Two\_Pence \mapsto 2, \ldots , One\_Pound \mapsto 100\} \\
HopperMax = \{One\_Penny \mapsto \ldots, \ldots , One\_Pound \mapsto \ldots\} \\
StockMax = \{Tea\_bag \mapsto \ldots, \ldots\} \\
CupMax = \ldots
\end{array}
$$

Als letzte Funktion wird *Value* definiert. Sie berechnet den Wert (in Pence) einer *Bag* von britischen Münzen. *Bags* sind an dieser Stelle sehr sinnvoll, da bei einer Menge von Münzen die Reihenfolge der Münzen unbedeutend ist, die Häufigkeit aber umso wichtiger (vgl. Kapitel 1.5.2).

3.) Beschreibung des abstrakten Zustands des Automaten

Der abstrakte Zustand des Automaten wird mit dem Schema Abs_State_Machine definiert:

Abs_State_Machine

$$
\begin{array}{l}
Balance, Takings : bag\ British\_coin \\
Stock : bag\ Ingredient \\
Cups : \mathbb{N} \\
Prices : Drink \rightarrow \mathbb{N} \\
Service\_light : Onoff \\
Coin\_display : \mathbb{N} \\
Report\_display : Message \\
\hline
\forall c : dom\ Takings \bullet 0 \leq count\ Takings\ c \leq HopperMax \\
\forall i : dom\ Stock \bullet 0 \leq count\ Stock\ i \leq StockMax\ i \\
0 \leq Cups \leq CupMax \\
Coin\_display = Value\ Balance
\end{array}
$$

Dem Schema kann entnommen werden, dass Balance und Takings Bags von *British_coin* sind. Balance zeichnet die Zusammenstellung der vom Kunden eingeworfenen Münzen auf. Die eingeworfenen Münzen werden unverzüglich zu Takings addiert. Somit beinhaltet Takings die momentan im Automaten verfügbare Gesamtsumme an Geld. Stock ist ein Bag mit den aktuell vorrätigen Zutaten und Cups beinhaltet die Anzahl der gegenwärtig verfügbaren Becher.

Prices ist eine totale Funktion, die einem Getränk (*Drink*) einen Preis zuordnet. Service_light, Coin_display und Report_display entsprechen den Anzeigen im Bedienfeld.

Die Prädikate des Schemas besagen:

- für jede Münzstückelung gibt es im Automaten einen Füllschacht mit einer maximalen Anzahl
- für jede Zutat gibt es einen maximalen Wert
- es gibt eine Höchstzahl an Bechern
- das Coin-Display zeigt immer den Wert der aktuellen *Balance*

4.) Beschreibung des initialen Zustands des Automaten

Im Startzustand des Systems sind alle Bags (*Balance*, *Stocks*, *Takings*) leer. Es sind noch keine Becher vorhanden und die Preise für die Getränke wurden noch nicht vergeben. Das *Report-Display* steht auf „*not in use*“ und das *Service-Display* deutet an, dass eine Wartung fällig ist.

5.) Definition der partiellen Operationen

In diesem Abschnitt werden die partiellen Operationen durch Schemata repräsentiert. Zwei dieser Operationen sollen hier kurz vorgestellt werden:

Das erste Schema beschreibt die Operation „Wartung des Automaten“ (*Service_Machine*):

Service_Machine

$$
\begin{array}{l}
\Delta\ \text{Abst\_State\_Machine} \\
\text{new\_stocks?} : \text{bag ingredient} \\
\text{new\_cups?} : \mathbb{N} \\
\text{new\_takings?} : \text{bag British\_coin} \\
\text{new\_prices?} : \text{Drinks} \pfun \mathbb{N} \\
\hline
\text{Balance}' = \text{Balance} \\
\text{Stocks}' = \text{Stocks} \uplus \text{new\_stocks?} \\
\text{Cups}' = \text{Cups}' + \text{new\_cups?} \\
\text{Takings}' = \text{Takings} \uplus \text{new\_takings?} \\
\text{Prices}' = \text{Prices} \uplus \text{new\_prices} \\
\text{Service\_Light}' = \text{off} \\
\text{Report\_display}' = \text{insert coin}
\end{array}
$$

Die Operation wird im Schema Service_Machine so beschrieben, dass Becher (new_cups), Zutaten (new_stocks) und Wechselgeld (new_takings) hinzugefügt sowie Preisänderungen (new_prices) eingegeben werden können.

Balance ändert sich durch die Operation nicht. Das Report_display zeigt nach der Operation "*insert coin*" an und das Service-Display (Service_Light) wird auf *off* gesetzt.
Für die Operation *Service_Machine* gibt es keine Vorbedingungen, folglich kann eine Wartung jederzeit stattfinden.

Die zweite Operation beschreibt die Aktion „Einwerfen einer einzelnen Münze" (*Insert_Coin*):

Insert_Coin

Δ Abst_State_Machine
c? : Coin

---

c ≠ Unnaceptable_coin
count Takings c? < HopperMax c?
Balance' = Balance ⊎ {(c? ↦ 1)}
Takings' = Takings ⊎ {(c? ↦ 1)}
Stocks' = Stocks
Cups' = Cups'
Prices' = Prices
Service_Light' = Service_Light
Report_display' = insert coin

Das Schema Insert_Coin hat zwei Vorbedingungen: die eingeworfene Münze muss britisch sein (c ≠ Unnaceptable_coin) und der Füllschacht der Münzstückelung darf noch nicht voll sein (count Takings c? < HopperMax c?).
Wenn die Münze akzeptiert wird, werden Balance und Takings erhöht und das Report_display fordert zum Einwurf einer weiteren Münze auf, der Rest bleibt unverändert.

Zusätzlich werden noch drei weitere Operationen definiert, die detailliert in Anlage 1 nachzulesen sind:

- *Get_drink*: Wahl eines Getränks durch eine Tastenkombination
- *Refund*: Erstattung der bereits eingeworfenen Münzen
- *Take_profit*: Entnahme des Geldes aus dem Automaten

6.) Erstellen der totalen Operationen

Zu den im vorherigen Abschnitt definierten Operationen werden die Fälle ermittelt, bei denen die Vorbedingungen nicht erfüllt werden. Operationen ohne Vorbedingungen, wie etwa *Service_Machine*, sind bereits totale Operationen und müssen nicht mehr berücksichtigt werden.

Bei der Operation *Insert_Coin* kann es allerdings sein, dass die Münze nicht akzeptiert wird, da sie nicht britisch ist oder der Füllschacht der Münzstückelung bereits voll ist. Für diese Fälle werden eigene Schemata definiert, die die „Umkehrung" der Vorbedingungen realisieren (d.h. *c = Unacceptable_coin* bzw. *count Takings c? = HopperMax c?*).
Mittels *Schema Calculus* werden die Schemata dann zu einer totalen Operation verbunden.

Entsprechend wird auch mit den partiellen Operationen *Get_drink*, und *Refund* verfahren. Da *Take_profit* keine Vorbedingungen hat, handelt es sich bereits um eine totale Operation.

3.2 Modellierung mit der UML

In diesem Kapitel sollen einige Teile der „Drinks Dispensing Machine" aus der im vorherigen Kapitel beschriebenen Z-Spezifikation mit der UML modelliert werden.

Zunächst werden die einfachen Typen aus dem zweiten Abschnitt modelliert. Die beiden Typen *Selection_buttons* (für die Auswahltasten) und *Ingredient* (für die Zutaten) sind im Z-Dokument folgendermaßen deklariert:

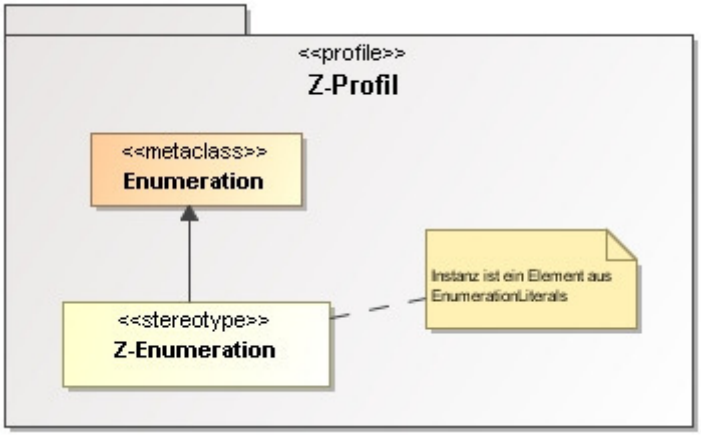
Selection_buttons ::== TEA | COFFEE | WHITE | SWEET | CHOCOLATE

Ingredient ::== Milk_powder | Chocolate_powder| Tea_bag |
Coffee_granulates | Sugar | Water

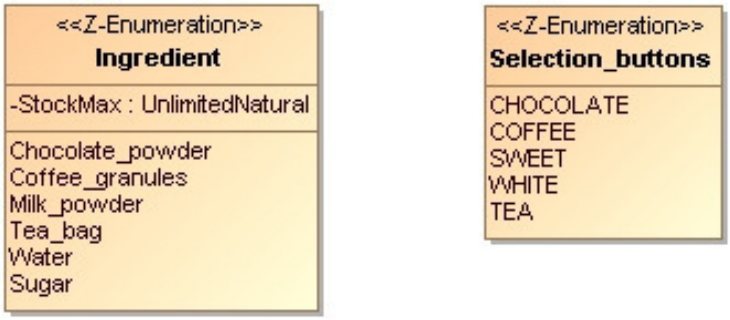
Diese Auflistungen lassen sich in der UML mit *Aufzählungstypen* (*Enumerations)* (siehe Kapitel 2.1) abbilden:

Die Enumeration *Ingredient* beinhaltet bereits *StockMax.* Dieses Attribut steht für die maximale Anzahl der Einheiten jeder Zutat, so wie es in der axiomatischen Definition in Abschnitt 2 des Z-Dokuments beschrieben ist.

Bei dieser Form der Modellierung wird angenommen, dass bei der Instanziierung einer Enumeration die Instanz ein Element aus den *Aufzählungswerten (EnumerationLiterals)* ist. Dies ist aber im UML-Metamodell so nicht eindeutig festgelegt, da dort ein EnumerationLiteral lediglich als *benutzerspezifischer Datenwert für eine Enumeration* [62] definiert ist (siehe Kapitel 2.1).
Deshalb wird es gegebenenfalls notwendig, ein eigenes *UML-Profil* für die Modellierung der Fallstudie anzulegen.[63] Dort kann dann über einen *Stereotyp* die vorhandene Metaklasse *Enumeration* entsprechend erweitert werden.[64] Ein Ausschnitt aus diesem Profil würde sich dann wie folgt gestalten:



Nach dieser Definition des neuen Stereotyps *Z-Enumeration* wird dieser den Aufzählungstypen *Selection_buttons* und *Ingredient* zugewiesen. Das folgende Schaubild zeigt die angepassten Aufzählungstypen. Im weiteren Verlauf des Kapitels wird aus Gründen der Übersichtlichkeit auf die Nutzung des neuen Stereotyps verzichtet (unter der Annahme, dass bereits *Enumeration* wie gewünscht benutzt werden kann).



[62] Vgl. OMG (Hrsg.) (2003), S. 101

[63] Vgl. dazu ausführlich: Jeckle, M. / Rupp, C. et al. (2005), S. 538 ff.

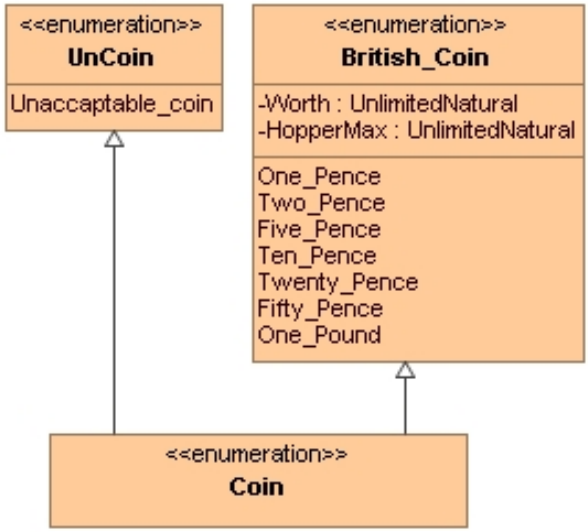
[64] Vgl. dazu ausführlich: OMG (Hrsg.) (2005), S. 649 ff.

Der Datentyp *Coin* für alle Münzen und die daraus abgeleitete Menge aller akzeptierten britischen Münzen (*British_Coin*) werden in der Z-Fallstudie in dieser Form spezifiziert:

$$Coin :: == One\_Penny \mid Two\_Pence \mid Five\_Pence \mid Ten\_Pence \mid Twenty\_Pence \mid Fifty\_Pence \mid One\_Pound \mid Unacceptable\_Coin$$

$$British\_Coin == \{c: Coin \mid c \neq Unacceptable\_Coin \}$$

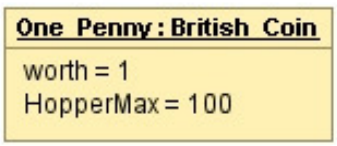
Mittels Aufzählungstypen lässt sich dieser Sachverhalt mit der UML wie folgt modellieren:



Die Enumeration *UnCoin* beinhaltet nur den Aufzählungswert *Unacceptable_Coin*, alle akzeptierten britischen Münzen sind als Aufzählungswerte in der Enumeration *British_Coin* enthalten. Durch die Generalisierungs- bzw. Spezialisierungsbeziehung „erbt“ *Coin* die EnumerationLiterals beider Aufzählungstypen. Diese spezielle Form der „Vererbung“ von EnumerationLiterals muss im angelegten UML-Profil (siehe oben) definiert werden.

British_Coin enthält bereits die Merkmale *Worth* und *HopperMax*. Diese Attribute stehen für den Wert einer Münze bzw. die maximale Anzahl an Münzen einer Stückelung im Füllschacht des Automaten, wie es in Abschnitt 2 der Z-Spezifikation vorgestellt wird.

Auf Basis dieser Modellierung könnte eine Instanz von *British_Coin* dieses Aussehen haben:



Durch Kombination von Auswahltasten wählt der Kunde ein bestimmtes Getränk. Dies ist im Z-Dokument mit der Menge *Drink* spezifiziert:

$$
\begin{aligned}
Drink == \{&\{TEA\}, \{COFFEE\}, \{CHOCOLATE\}, \{TEA, WHITE\}, \{TEA\ SWEET\},\\
&\{TEA, WHITE, SWEET\}, \{COFFEE, WHITE\},\{COFFEE, SWEET\},\\
&\{COFFEE, WHITE, SWEET\}\}
\end{aligned}
$$

Jedes dieser Getränke hat eine Liste von Zutaten, was die Menge *List_of_ingredients* repräsentiert:

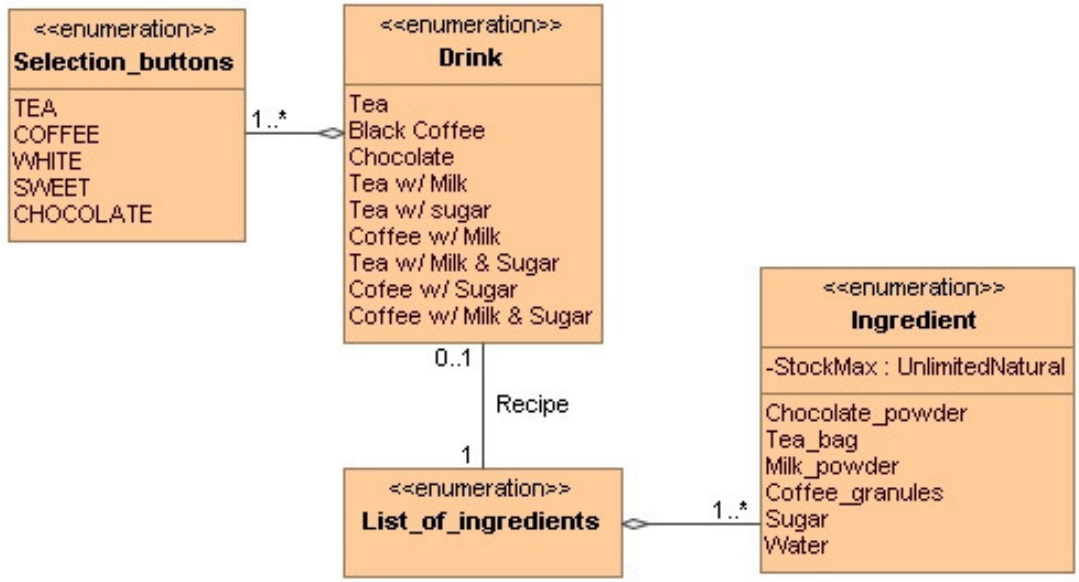
$$
\begin{aligned}
List\_of\_ingredients == \{&\{Tea\_bag, Water\}, \{Coffee\_granules, Water\},\\
&\{Chocolate\_powder, Water\}, \{Tea\_bag, Water,\\
&\ Milk\_powder\}, \{Tea\_bag, Water, Sugar\},\\
&\{Tea\_bag, Water, Sugar, Milk\_powder\},\\
&\{Coffee\_granules, Water, Milk\_powder\},\\
&\{Coffee\_granules, Water, Sugar\},\\
&\{Coffee\_granules, Water, Sugar, Milk\_powder\}\}
\end{aligned}
$$

Die Zuordnung eines Getränks zu einer Zutatenliste erfolgt mit der totalen Injektion *Recipe*:

$$
\begin{aligned}
&Recipe: Drink \rightarrowtail List\_of\_ingedients\\
&\hline
&Recipe = \{\{TEA\} \mapsto \{Tea\_bag, Water\},\\
&\qquad \{COFFEE\} \mapsto \{Coffee\_granules, Water\},\\
&\qquad \{CHOCOLATE\} \mapsto \{Chocolate\_powder, Water\},\\
&\qquad \{TEA, WHITE\} \mapsto \{Tea\_bag, Water,\ Milk\_powder\},\\
&\qquad \{TEA\ SWEET\} \mapsto \{Tea\_bag, Water, Sugar\},\\
&\qquad \ldots\\
&\qquad \}
\end{aligned}
$$

Genau genommen wird *Recipe* dazu benutzt, um eine Abbildung zwischen einer bestimmten Tastenkombination und der zugehörigen Zutatenliste zu schaffen.

Die entsprechende UML-Modellierung gestaltet sich folgendermaßen:

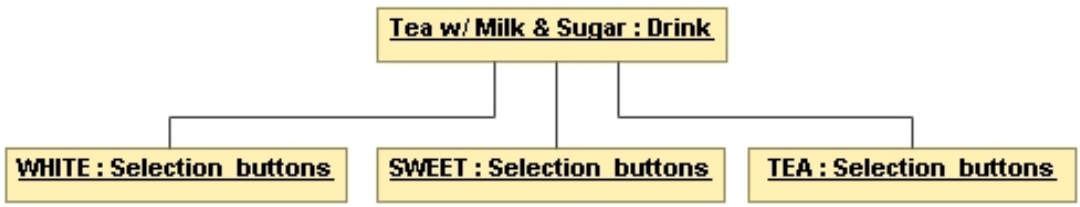


Die Aggregation zwischen *Selection_buttons* und *Drink* besagt, dass ein Getränk durch eine oder mehrere Auswahltasten charakterisiert wird.

Mit der Assoziation *Recipe* zwischen *Drink* und *List_of_ingredients* wird ausgedrückt, dass ein Getränk genau eine Zutatenliste hat und eine Zutatenliste maximal zu einem Getränk gehört. Dadurch wird die in der Z-Spezifkation formulierte *totale Injektion* beschrieben.[65] Die verschiedenen Funktionstypen und deren UML- Modellierungen werden ausführlich in Kapitel 4 erörtert.

Die Aggregation zwischen *List_of_ingredients* und *Ingredient* besagt, dass eine Zutatenliste aus mindestens einer Zutat besteht.

Eine Instanz von Drink würde dann diesen Aufbau haben (immer unter der Voraussetzung, Enumeration kann wie oben beschrieben instanziiert werden bzw. dies wurde entsprechend im Z-spezifischen UML-Profil hinterlegt):

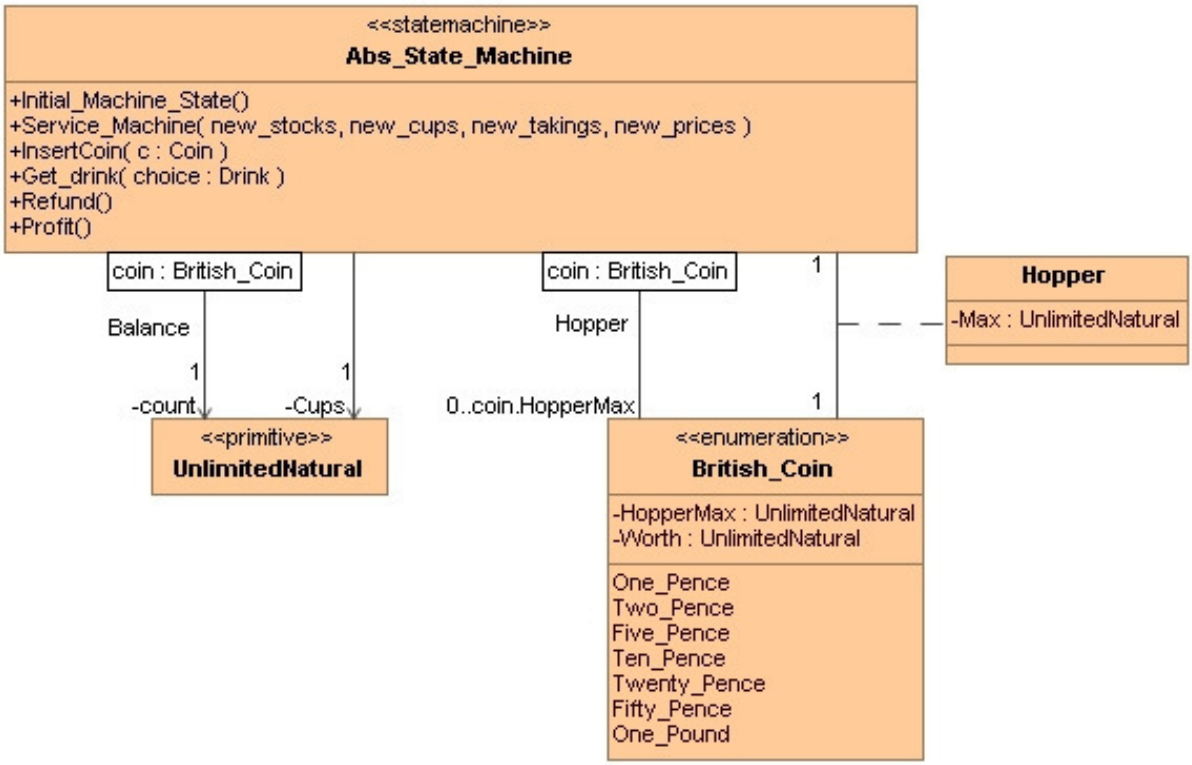


[65] Vgl. Sheppard, D. (1995), S. 120 ff.

Nachdem nun die Typen, Mengen und Relationen der Fallstudie „A Drinks Dispensing Machine“ modelliert wurden, werden im nächsten Schritt die Zustände und Operationen des Getränkeautomaten untersucht.

Wie in Kapitel 1 beschrieben wurde, wird in einer Z-Spezifikation ein System über seinen Zustand und Änderungen dieses Zustands modelliert. Es ist also nahe liegend, bei einer Adaption mir der UML die in Kapitel 2 beschriebenen *Zustandsautomaten (State machines)* zu verwenden.

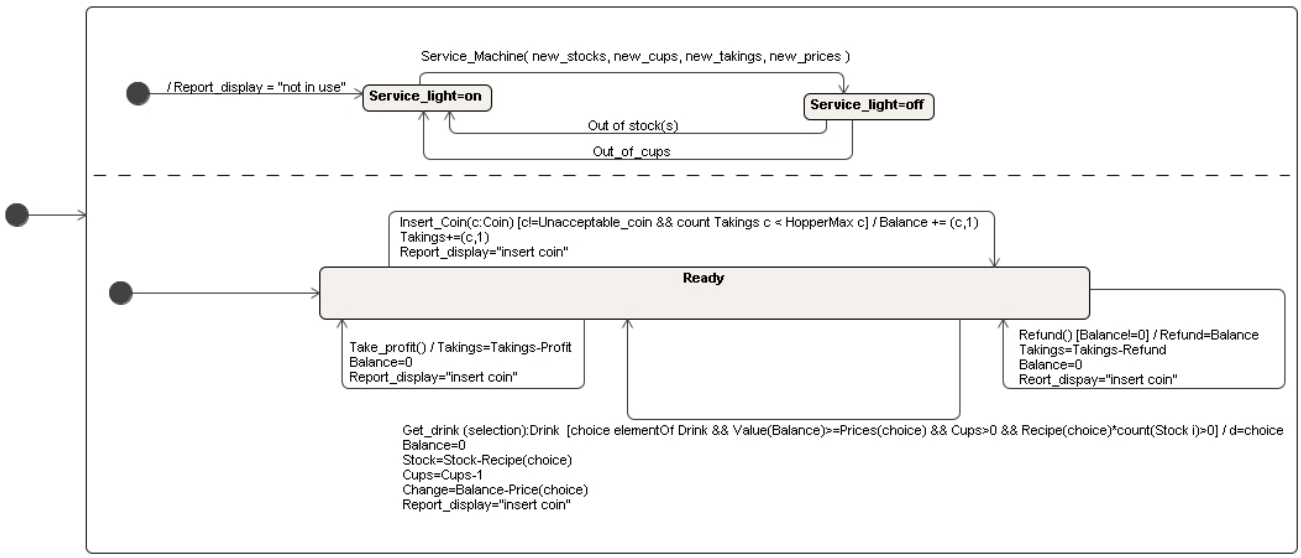
Für den Getränkeautomaten sieht eine solche State machine mit ihren Operationen und Attributen im Klassendiagramm dargestellt wie folgt aus:



Der Zustandsautomat *Abs_State_Machine* besitzt fünf Operationen: *Initial_Machine_State* zur Initialisierung des Automaten, *Service_Machine* für die Wartung, *Insert_Coin* für das Einwerfen einer einzelnen Münze, *Get_drink* für die Wahl eines Getränks, *Refund* für die Erstattung bereits eingeworfener Münzen und *Take_profit* für die Entnahme des Geldes aus dem Automaten.

Über die Assoziation *Balance* wird ausgedrückt, dass das Attribut *count* vom primitiven Datentyp *UnlimitedNatural* ist. Ebenso verhält es sich mit dem Attribut *Cups*.
Die Assoziation *Hopper* und die Assoziationsklasse beschreiben, dass der Automat zu jeder akzeptierten britischen Münzstückelung einen Füllschacht mit einer bestimmten Größe hat.

Die eigentliche State machine, in der die Zustände und Transitionen modelliert werden, ist folgendermaßen aufgebaut (eine vergrößerte Abbildung findet sich in Anlage 2 im Anhang):



Als Erstes ist zu erwähnen, dass die State machine zwei Regionen beinhaltet. Dadurch lassen sich zueinander parallele Abläufe beschreiben.[66)]

Im oberen Bereich wird die Funktion für die Wartung des Automaten (*Service_machine*) modelliert. Nach der Initialisierung hat das System den Zustand, dass das Service-Display angeschaltet ist (*Service_light=on*). Durch die Wartung können Zutaten, Becher und der Geldbestand aufgefüllt sowie die Preise gesetzt werden. Anschließend ist der Zustand *Service_light=off* aktiv. Dieser wird nur verlassen, wenn Becher (*Out_of_cups*) oder Zutaten (*Out_of_stock(s)*) fehlen.

Die anderen Operationen werden im unteren Bereich modelliert.
Ist der Automat im Bereitschaftszustand (*Ready*) kann der Kunde Geld einwerfen. Die zugehörige Transition hat den *CallTrigger InsertCoin()*, die Bedingungen (*Guards*) entsprechen den Vorbedingungen aus dem Schema in der Z-Spezifikation: die eingeworfene Münze *c* muss akzeptiert werden und der Füllschacht der Stückelung darf noch nicht voll sein. Das definierte *Verhalten* besagt, dass *Balance* und *Takings* um den Wert von *c* erhöht werden und das Report-Display „*insert coin*“ anzeigt. Somit entspricht das Verhalten den Nachbedingungen des Z-Schemas. Nach Ausführung der Transition befindet sich das System wieder im Zustand *Ready*.

---

66) Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 371

Die Operation *Refund* wird mit der nächsten Transition im Schaubild ausgedrückt. Der CallTrigger *Refund()* hat einen Guard, der besagt, dass die Summe des eingeworfenen Geldes nicht 0 sein darf (*[Balance!=0]*). Wird diese (Vor-)Bedingung erfüllt, wird das angegebene Verhalten ausgeführt, das heißt *Refund* erhält den Wert von *Balance*, *Balance* wird von *Takings* abgezogen, *Balance* wird auf 0 gesetzt und das Report-Display zeigt „*insert coin*“ an. Anschließend ist der Zustand *Ready* wieder aktiv.

Die dritte Transition in der unteren Region steht für die Operation *Get_drink*. Auch hier entsprechen die Bedingungen, die in den Guards definiert sind, den Vorbedingungen aus dem Z-Dokument: die Getränkewahl, die sich durch die Kombination der Auswahltasten ergibt, muss einem verfügbaren Getränk entsprechen (*choice elementOf Drink*), der eingeworfene Geldbetrag muss mindestens so hoch sein wie der Preis des gewählten Getränks (*Value(Balance)>=Prices(choice)*), es müssen noch Becher verfügbar sein (*Cups>0*) und alle für das Getränk benötigten Zutaten müssen noch vorrätig sein (*Recipe(choice)*count(Stock i)>0*).
Sind diese Bedingungen erfüllt, wird Balance auf 0 gesetzt (*Balance=0*), der Zutatenvorrat und die Becher werden reduziert (*Stock=Stock-Recipe(choice)* bzw. *Cups=Cups-1*) und das Rückgeld wird ermittelt (*Change=Balance-Price(choice)*). Nach Ausführen der Transition befindet sich das System wieder im Zustand *Ready*.

Die letzte Operation ist *Take_profit*. Da es hierzu keine Vorbedingung gibt, enthält die Transition auch keinen Guard. Das Verhalten besagt, dass sich der Geldbestand des Automaten um den entnommenen Betrag verringert (*Takings=Takings-Profit*), dass *Balance* auf 0 gesetzt wird (*Balance=0*) und dass das Report-Display „*insert coin*“ anzeigt.

Mit dieser Modellierung konnten somit alle *partiellen Operationen* abgedeckt werden. Wie aber im vorherigen Kapitel ausführlich beschrieben wurde, reicht es nicht aus nur den „Gutfall“ zu betrachten. Vielmehr müssen auch die Fälle abgedeckt werden, in denen die Vorbedingungen (bzw. die Bedingungen der Guards) nicht erfüllt werden. Erst dann sind die *totalen Operationen* vollständig spezifiziert. Wie man hierzu vorgehen kann soll abschließend exemplarisch bei der Operation *Insert_Coin* gezeigt werden.

In der Z-Spezifikation der Fallstudie sind für die Umstände, die nicht mit den Vorbedingungen des Schemas *Insert_Coin* abgedeckt sind, zwei weitere Schemata aufgeführt, *Reject_Coin* und *Hopper_full*:

Reject_Coin

Δ Abst_State_Machine
c?, c! : Coin

---

c = Unnaceptable_coin
Balance' = Balance
Takings' = Takings
Stocks' = Stocks
Cups' = Cups'
Prices' = Prices
Service_Light' = Service_Light
Report_display' = try another coin
c! = c?

Hopper_full

Δ Abst_State_Machine
c?, c! : Coin

---

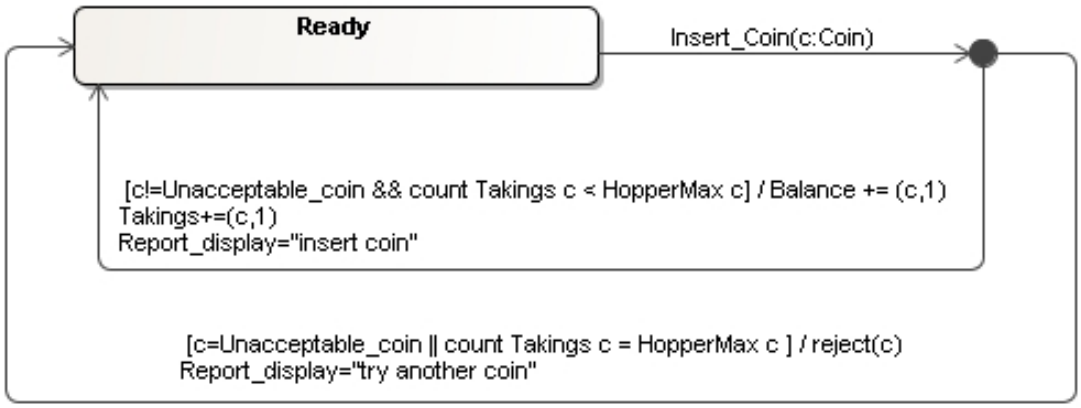
count Takings c? = HopperMax c?
Balance' = Balance
Takings' = Takings
Stocks' = Stocks
Cups' = Cups'
Prices' = Prices
Service_Light' = Service_Light
Report_display' = try another coin
c! = c?

Für jede der Vorbedingungen aus *Insert_Coin* ist der „Schlechtfall“ in einem eigenen Schema definiert: wird eine nicht akzeptable Münze eingeworfen ($c = Unnaceptable\_coin$) oder ist der Füllschacht für die Münzstückelung bereits voll ($count\ Takings\ c? = HopperMax\ c?$).
Die Auswirkungen sind in beiden Fällen dieselben: das Report-Display zeigt die Meldung *„try another coin“* an und die eingeworfene Münze *c* wird wieder ausgegeben (*c! = c?*).

Anschließend werden die Schemata mit dem Schema Calculus zu einer totalen Operation vereint:

$$Total\_Insert\_coin \mathrel{\widehat{=}} Insert\_coin \vee Reject\_coin \vee Hopper\_full$$

Im UML-Zustandsautomaten kann dies über die in Kapitel 2.2 vorgestellten *Kreuzungen (Junctions)* modelliert werden:



Eine aus der Kreuzung ausgehende Transition beschreibt den „Gutfall“, wie er oben bereits in der State machine enthalten ist. Die andere ausgehende Transition beschreibt die Ausnahmen.
Da die beiden Ausnahmefälle dieselbe Auswirkung haben, können sie in dem Beispiel zu einer Transition mit dem Guard *[c=Unacceptable_coin || count Takings c = HopperMax c]* zusammengefasst werden. Das Verhalten ist in beiden Fällen die Rückgabe der eingeworfenen Münze (*reject(c)*) und das Anzeigen der Fehlermeldung (*Report_display="try another coin"*).

Entsprechend muss auch für die anderen partiellen Operationen *Get_drink* und *Refund* vorgegangen werden.

Somit konnten alle wesentlichen Teile der Z-Spezifikation mit der UML abgebildet werden: angefangen bei der Deklaration von Typen, Mengen und Relationen über die Modellierung von Zuständen und Zustandsänderungen durch partielle Operationen bis hin zur Erstellung von totalen Operationen.

4. Die Funktionen der Z-Notation in der UML

Nachdem im ersten Kapitel bereits die Funktionen als spezielle Ausprägung der Relationen vorgestellt wurden, werden in diesem Kapitel die verschiedenen Formen der Funktionen in der Z-Notation behandelt. Anschließend wird ihre Modellierung mit der UML vorgenommen.

4.1 Domain und Range

Bisher wurde im Zusammenhang mit Funktionen immer von einer „Grundmenge“ und einer „Wertemenge“ gesprochen.

Es kann aber notwendig sein, dass man nur den Teil der Grundmenge, in der alle ersten Elemente eines geordneten Paares liegen, und den Teil der Wertemenge, in der alle zweiten Elemente eines geordneten Paares liegen, betrachtet. Die Z-Notation hat hierfür eigene Begrifflichkeiten: ***Domain*** und ***Range***. [67)]

In dem Beispiel mit dem Durchwahl-Verzeichnis aus Kapitel 1.5.1 wurde eine partielle Funktion $\mathrm{telefon}_F$ definiert:

$$
\begin{array}{|l}
\mathrm{telefon}_F : \mathrm{NAME} \pfun \mathrm{DURCHWAHL} \\
\hline
\mathrm{telefon}_F = \{ \\
\quad \mathrm{Klaus\ Meier} \mapsto 133, \\
\quad \mathrm{Elke\ Schmid} \mapsto 181, \\
\quad \mathrm{Herbert\ Knebel} \mapsto 133, \\
\quad \}
\end{array}
$$

In diesem Beispiel entspricht die Grundmenge der deklarierten Menge NAME und die Wertemenge der deklarierten Menge $\mathrm{DURCHWAHL}$. Die *Domain* von $\mathrm{telefon}_F$ beinhaltet dagegen alle Mitarbeiter, die per Telefon erreicht werden können. Die *Range* von $\mathrm{telefon}_F$ beinhaltet alle Nummern, die Telefonen zugeordnet wurden.

Es ist durchaus denkbar, dass es in der Firma weitere Namen gibt, die nicht in der Domain sind, da ihnen kein Telefonanschluß zugeordnet ist. Ebenso gibt es sicherlich zahlreiche Nummern in $\mathrm{DURCHWAHL}$, die keinem Telefon zugewiesen sind. Man kann also vereinfachend sagen, dass Domain und Range Teilmengen der Grund- und Wertemenge sind.

[67)] Vgl. Jacky, J. (1997), S. 82 f.

Die Abkürzungen *dom* (Domain) und *ran* (Range) sind Operatoren, deren Argument eine binäre Relation und deren Wert eine Menge ist.

Für das obere Beispiel gilt folglich:

$$\text{dom telefon}_F = \{\text{Klaus Meier, Elke Schmid, Herbert Knebel}\}$$

$$\text{ran telefon}_F = \{133, 181\}$$

4.2 Funktionstypen der Z-Notation

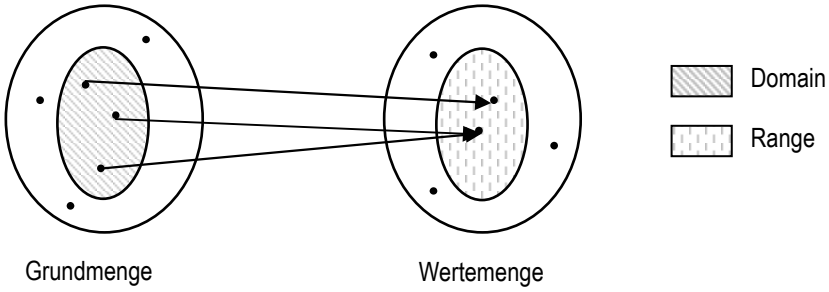
Im diesem Kapitel werden die verschiedenen Typen von Funktionen der Z-Notation[68] vorgestellt, um im nächsten Abschnitt deren Modellierung mit der UML darzustellen.

Im Folgenden bezeichnen *X* und *Y* Mengen und *func* eine Funktion: $[X, Y]$

Die ***partielle Funktion*** ist die allgemeinste Form der Funktion. Sie ist folgendermaßen definiert:

$$X \pfun Y == \{ \text{func}: X \rel Y \mid (\forall x : X; y1, Y2 : Y \bullet$$
$$(x \mapsto y1) \in \text{func} \land (x \mapsto y2) \in \text{func} \Rightarrow y1 = y2\}$$

Das bedeutet, dass jedem Element der Grundmenge höchstens ein Element in der Wertemenge zugeordnet wird. Das heißt aber auch, dass die Grundmenge Werte enthalten kann, die nicht zwangsläufig Elemente der Domain sein müssen.[69] Dies drückt auch das folgende Schaubild aus:[70]



Bei der ***totalen Funktion*** dagegen entspricht die Grundmenge der Domain. In diesem Fall ist jedes Element der Grundmenge (immer!) einem Element der Wertemenge zugeordnet.[71] Man kann auch sagen, die totale Funktion ist eine Teilmenge der partiellen Funktion.

Die Definition lautet:

$$X \fun Y == \{ \text{func}: X \pfun Y \mid \text{dom func} = X\}$$

---

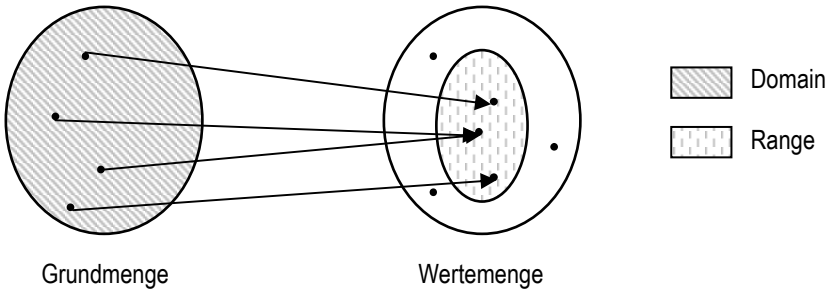
[68] Vgl. dazu ausführlich: Spivey, J.M. (1992), S. 105 f.

[69] Vgl. Sheppard, D. (1995), S. 118

[70] Alle Schaubilder mit Änderungen entnommen aus: Sheppard, D. (1995), S. 118 ff.

[71] Vgl. Sheppard, D. (1995), S. 119

Als Schaubild gestaltet sich dieser Sachverhalt folgendermaßen:

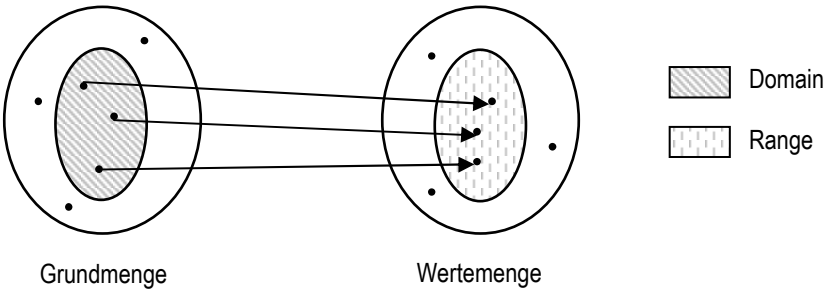


Eine ***Injektion*** zeichnet sich dadurch aus, dass alle Elemente der Wertemenge höchstens einmal in der Range auftauchen. Dementsprechend muss die Umkehrabbildung auch wieder eine Funktion sein.[72)]

Wenn dabei jedem Element der Grundmenge höchstens ein Element der Wertemenge zugeordnet wird, spricht man von einer ***partiellen Injektion***. Die Definition hierfür lautet:

$$X \rightarrowtail Y == \{ func: X \nrightarrow Y \mid (\forall x1, x2 : dom\ func \bullet func(x1) = func(x2) \Rightarrow x1 = x2 \}$$

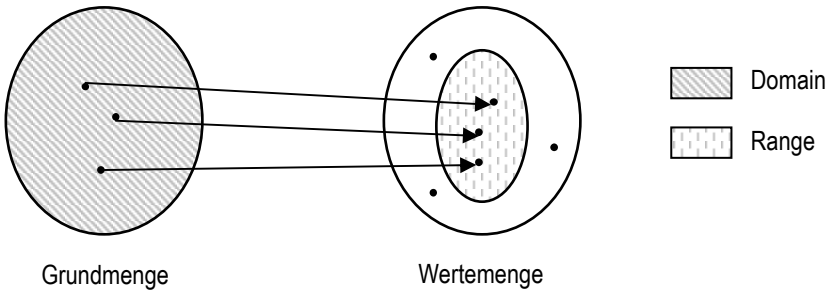
Das zugehörige Schaubild hat dieses Aussehen:



Von einer ***totalen Injektion*** spricht man dagegen, wenn die Domain gleich der Grundmenge ist:[73)]

$$X \rightarrowtail Y == \{ func: X \nrightarrow Y \mid dom\ func = X \} = (X \rightarrowtail Y) \cap (X \rightarrow Y)$$

Abgebildet werden kann die totale Injektion wie folgt:



[72)] Vgl. Ficker, H.-J. / Jarré, S. et al. (1998), S.6

[73)] Vgl. Ficker, H.-J. / Jarré, S. et al. (1998), S.7

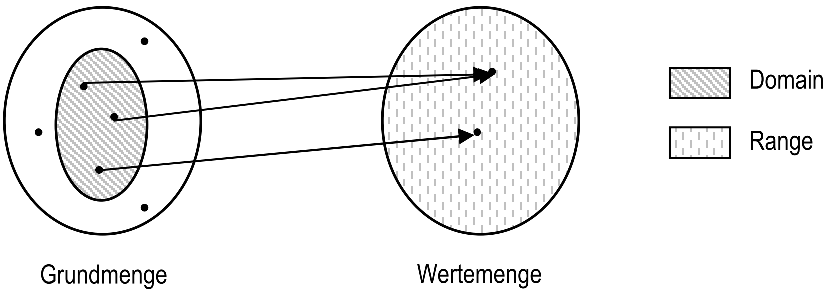
[73)] Vgl. Ficker, H.-J. / Jarré, S. et al. (1998), S.7

Bei einer ***Surjektion*** taucht jedes Element der Wertemenge in den geordneten Paaren der Funktion mindestens einmal auf. Das heißt die Range der Funktion entspricht der Wertemenge.[73)]

Bei der ***partiellen Surjektion*** wird zusätzlich zur Surjektion jedem Element in der Grundmenge höchstens ein Element in der Wertemenge zugeordnet:

$$X \twoheadrightarrow\!\!\!\!\!\!+\; Y == \{\ func: X \nrightarrow Y \mid ran\ f = Y\ \}$$

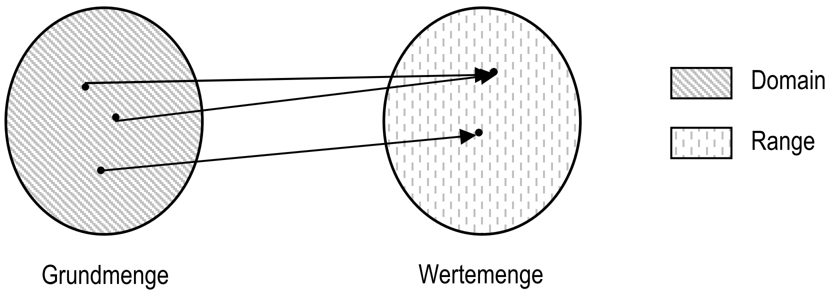
In einem Mengenschaubild kann die partielle Surjektion folgendermaßen dargestellt werden:



Bei der ***totalen Surjektion*** entspricht zusätzlich zur Surjektion die Domain der Grundmenge:

$$X \twoheadrightarrow Y == (X \twoheadrightarrow\!\!\!\!\!\!+\; Y) \cap (X \rightarrow Y)$$

Dieser Umstand sieht in der Darstellung wie folgt aus:



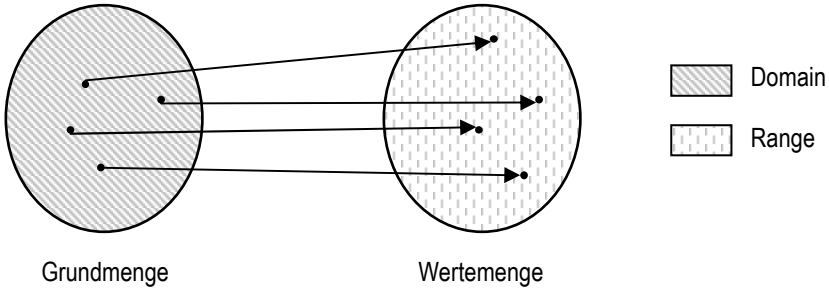
Eine ***Bijektion*** ist eine Funktion mit einer totalen 1:1-Beziehung zwischen der Grundmenge und der Wertemenge. Sie ist folglich beides, eine totale Injektion als auch eine totale Surjektion.[74)]
Jedem Element wird durch die bijektive Funktion genau ein Element der Wertemenge zugeordnet, so dass auch beide Mengen gleich groß sind.[75)] Dementsprechend ist die Bijektion definiert:

$$X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y == (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)$$

[74)] Vgl. Sheppard, D. (1995), S. 121

[75)] Vgl. Ficker, H.-J. / Jarré, S. et al. (1998), S.7

In einem Mengeschaubild kann die Bijektion auf diese Weise abgebildet werden:



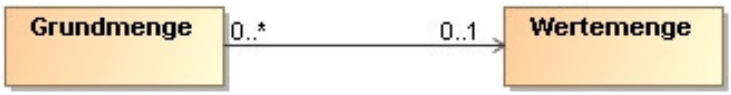
4.3 Modellierung der Funktionen mit der UML

Nachdem das grundsätzliche Verständnis für die Funktionstypen geschaffen wurde, soll nun eine Modellierung der verschiedenen Zuordnungen mit der UML erfolgen.

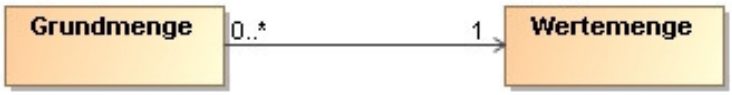
Für die Darstellung von gleichartigen Beziehungen verwendet die UML *Assoziationen (associations)*.[76] Assoziationen eignen sich daher für die Visualisierung der Z-Funktionen. Vor allem die Möglichkeit eine *Multiplizität* für die *Assoziationsenden* festzulegen, kann dazu genutzt werden.

In Kapitel 3 wurde bereits grundsätzlich formuliert, dass Funktionen im Gegensatz zu Relationen keine n:n- oder 1:n-Beziehungen sein können. Vielmehr handelt es sich um n:1- oder 1:1-Beziehungen. Diese Aussage beschränkte sich auf *Domain* und *Range* einer Funktion. Eine differenzierte Betrachtung soll nun diese Aussage ergänzen.

Die ***partielle Funktion*** lässt sich durch die folgende *unidirektional navigierbare* Assoziation ausdrücken:

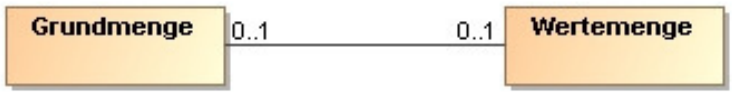


Eine ***totale Funktion*** hingegen lässt sich wie folgt darstellen:

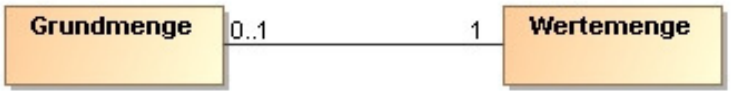


[76] Vgl. Jeckle, M. / Rupp, C. et al. (2005), S. 134 f.

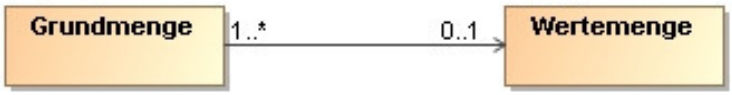
Laut der Definition ist die Umkehrabbildung der *Injektion* auch wieder eine Funktion (siehe Kapitel 4.2). Dies lässt sich durch die *bidirektionale Navigierbarkeit* der Assoziation ausdrücken. Folglich sieht eine ***partiellen Injektion*** folgendermaßen aus:



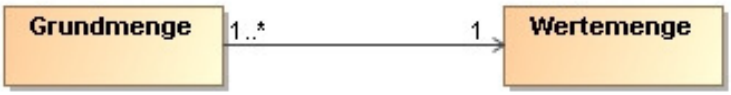
Die ***totale Injektion*** wurde bereits bei der Fallstudie in Kapitel 3 eingesetzt. Sie entspricht dieser Darstellung:



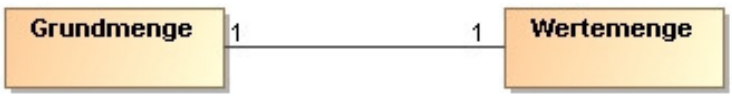
Bei der *Surjektion* entspricht die Wertemenge der Range. Entsprechend wird eine ***partielle Surjektion*** wie folgt visualisiert:



Bei der totalen Surjektion entspricht zusätzlich die Domain der Grundmenge, was sich so darstellen lässt:



Eine ***Bijektion*** entspricht wie oben beschrieben einer totalen 1:1-Beziehung zwischen Grund- und Wertemenge. Sie ist sowohl eine totale Injektion als auch eine totale Surjektion. Daher ergibt sich folgende Visualisierung:



Zusammenfassung und Ausblick

Abschließend lässt sich sagen, dass die beiden vorgestellten Vorgehensweisen zur Software-Spezifikation ihre eigenen Stärken aber auch viele Gemeinsamkeiten haben. Vor allem im letzten Kapitel wurde deutlich, dass die UML einfache und ausreichende Möglichkeiten bietet auch komplexere mathematische Abbildungen zu modellieren.

Eine weitergehende Untersuchung könnte sich mit den objektorientierten Erweiterungen von Z beschäftigen. Zwar bietet Z mit Zustands- und Operationsschemata bereits die Möglichkeit Klassen mit Attributen und Methoden zu definieren, wesentliche Elemente der Objektorientierung wie Vererbung oder Polymorphismus sind aber nur recht umständlich abbildbar.
Hier setzen die objektorientierten Erweiterungen und Dialekte von Z an. Als wichtigste Vertreter sind hier Object-Z [77)] und Z++ [78)] zu nennen.

[77)] Vgl. dazu ausführlich: Smith, G. (1999)

[78)] Vgl. dazu ausführlich: Lano, K. (1990)

Anhang:

Anlagenverzeichnis

19

A DRINKS DISPENSING MACHINE

19.1 INTRODUCTION

This final case study examines the specification of a drinks dispensing machine, the control panel for which is shown in Fig 19.1. Coins are inserted to (at least) the value of the drink and the drink chosen by pressing an appropriate combination of the selection buttons. The drink is then served by pressing button D. Unlike tea and coffee, chocolate is served only white *and* sweet (commercial chocolate powder contains whitener and sweetener). If for some reason the drink cannot be dispensed the customer can obtain a full refund by pressing button R. The machine accepts any coin that is legal tender in the United Kingdom and the correct change is issued (if necessary) once a drink has been dispensed.

The machine has a number of displays which are useful to the customer and to those who maintain it. The service light is illuminated if the machine needs to be serviced. Servicing will be required when ingredients for certain drinks are exhausted. However, the machine is fully functional for drinks that do not need those ingredients. The only occasion when the machine is effectively out of use (apart from initialization) is when the ingredients for all the drinks have been exhausted. The two other displays are the coin display and the report display. The coin display shows the current value (in pence) of the coins inserted into the machine. This is a simple LED display with the word BAL (for balance) and the letter P permanently part of the display. Immediately below this is the report display which is used to give instructions to the customers. On the real machine this would be designed in some eye-catching way.

Machines like this are quite commonplace and frequently driven by microchips which contain the logic for deducing the outcome of a particular machine situation. This case study is therefore an exercise in the specification of the ROM programs. The study follows the now familiar pattern but omits the various proofs, precondition calculations and indexes in an attempt to make it more manageable and readable. These are left as exercises. This study concentrates instead on the techniques of Z and, in particular, the use of bags.

19.2 THE GIVEN SETS, ABBREVIATION DEFINITIONS AND AXIOMATIC DESCRIPTIONS

There are a number of sets and functions to be defined in this section. We begin by providing simple data types for the selection buttons on the control panel and the ingredients of the various drinks:

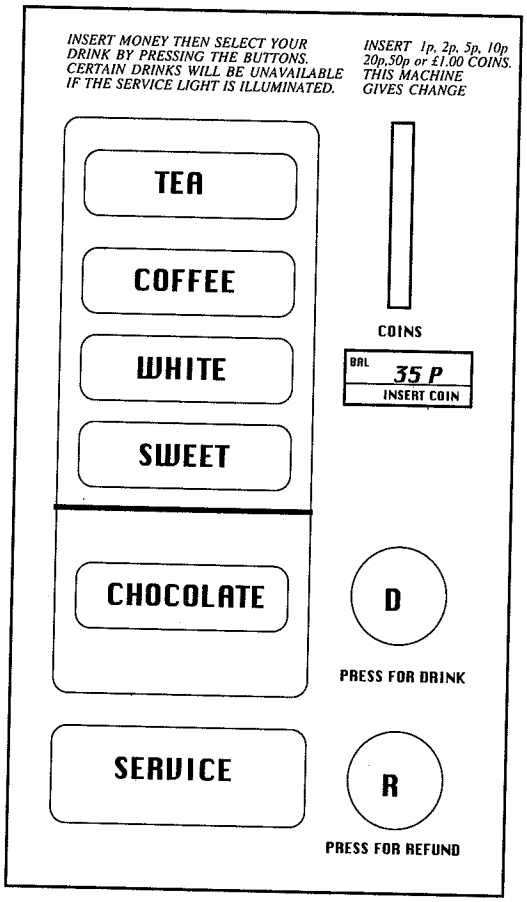


Fig 19.1 Control panel for the simple drinks dispenser

Selection_buttons ::= *TEA* | *COFFEE* | *WHITE* | *SWEET* | *CHOCOLATE*
Ingredient ::= *Milk_powder* | *Chocolate_powder* | *Tea_bag* | *Coffee_granules* | *Sugar* | *Water*

The information displayed in the report display is provided by the data type *Message*:

Message ::= *this drink is unavailable* | *insert more money and select drink again* | *correct change unavailable* | *no cups in machine* | *try another coin* | *not in use* | *insert coin*

while the condition of the service light is indicated using the data type *Onoff*:

$$Onoff ::= on \mid off$$

We will further assume that there is a 'universe' of coins which contains the accepted British coin set. All other coins are regarded as 'foreign' and are not differentiated. A foreign coin is unacceptable to the machine. The data type *Coin* summarizes this universe from a British perspective:

$$Coin ::= One\_Penny \mid Two\_Pence \mid Five\_Pence \mid Ten\_Pence \mid Twenty\_Pence \mid Fifty\_Pence \mid One\_Pound \mid Unacceptable\_coin$$

The set of British coins is therefore easily defined through the following abbreviation definition:

$$British\_Coin == \{c : Coin \mid c \neq Unacceptable\_coin\}$$

Having introduced the set of selection buttons we are now in a position to define those combinations of button presses that represent a recognized drink. This information is provided by simple enumeration in the abbreviation definition for *Drink*. Notice that the choice of drink is represented by a *set of buttons* which means that the order in which the buttons are pressed is irrelevant. Notice also that odd combinations such as $\{TEA, COFFEE, CHOCOLATE\}$ are not members of this set and therefore represent unavailable drinks (thank goodness!):

$$Drink == \{\{TEA\}, \{COFFEE\}, \{CHOCOLATE\}, \{TEA, WHITE\}, \{TEA, SWEET\}, \{TEA, WHITE, SWEET\}, \{COFFEE, WHITE\}, \{COFFEE, SWEET\}, \{COFFEE, WHITE\ SWEET\}\}$$

The final abbreviation definition provides the list of ingredients for each one of the drinks:

$$List\_of\_ingredients == \{\{Tea\_bag, Water\}, \{Coffee\_granules, Water\}, \{Chocolate\_powder, Water\}, \{Tea\_bag, Water, Milk\_powder\}, \{Tea\_bag, Water, Sugar\}, \{Tea\_bag, Water, Sugar, Milk\_powder\}, \{Coffee\_granules, Water, Milk\_powder\}, \{Coffee\_granules, Water, Sugar\}, \{Coffee\_granules, Water, Sugar, Milk\_powder\}\}$$

SAQ 19.1 Write definitions for these two sets by set comprehension.

We now turn our attention to the various constants and functions that will be required. First, we have to be able to associate a particular drink with the set of ingredients required to make it. The ingredient set is unique to a drink and each drink must have a set of ingredients so we define *Recipe* as a total injection from *Drinks* to *List_of_ingredients*. Clearly, *Recipe* represents a fixed mapping which is simply used to look up the ingredients associated with a particular combination of selection buttons. We therefore declare *Recipe* as a global constant through an axiomatic description:

$$Recipe : Drink \rightarrowtail List\_of\_ingredients$$

$$\begin{aligned} Recipe = \{&\{TEA\} \mapsto \{Tea\_bag, Water\} \\ &\{COFFEE\} \mapsto \{Coffee\_granules, Water\} \\ &\{CHOCOLATE\} \mapsto \{Chocolate\_powder, Water\} \\ &\{TEA, WHITE\} \mapsto \{Tea\_bag, Water, Milk\_powder\} \\ &\{TEA, SWEET\} \mapsto \{Tea\_bag, Water, Sugar\} \\ &\{TEA, WHITE, SWEET\} \mapsto \{Tea\_bag, Water, Sugar, Milk\_powder\} \\ &\{COFFEE, WHITE\} \mapsto \{Coffee\_granules, Water, Milk\_powder\} \\ &\{COFFEE, SWEET\} \mapsto \{Coffee\_granules, Water, Sugar\} \\ &\{COFFEE, WHITE, SWEET\} \mapsto \{Coffee\_granules, Water, Sugar, Milk\_powder\}\} \end{aligned}$$

A similar argument applies to *Worth*, the function that associates a coin with its value. This must be a total injection from *British_Coin* to its value in pence because all coins have a unique value. We will therefore again define the function as a global constant.

The machine is assumed to have a number of coin hoppers or tubes where each denomination is separately stored. The machine can store only a certain number of coins in each hopper and so we define a total function called *HopperMax* which determines the maximum number of coins of each denomination that can be held. Similarly, *StockMax* is a total function that associates each ingredient with the maximum number of units (tea bags, sugar lumps, etc.) that the machine can store, while *CupMax* represents the maximum number of plastic cups the machine can hold. Water is assumed to be inexhaustible because all machines are connected to the main supply. The individual limits for coins, cups and ingredients will be supplied later by the manufacturer of the machine. All these points are reflected in the following axiomatic description:

$$
\begin{array}{l}
Worth : British\_Coin \rightarrowtail \mathbb{N} \\
HopperMax : British\_Coin \rightarrow \mathbb{N}_1 \\
StockMax : Ingredient \rightarrow \mathbb{N}_1 \\
CupMax : \mathbb{N} \\
\hline
Worth = \{One\_Penny \mapsto 1, Two\_Pence \mapsto 2, Five\_Pence \mapsto 5, Ten\_Pence \mapsto 10, \\
\qquad Twenty\_Pence \mapsto 20, Fifty\_Pence \mapsto 50, One\_Pound \mapsto 100\} \\
HopperMax = \{One\_Penny \mapsto .., \ldots, One\_Pound \mapsto ..\} \\
StockMax = \{Tea\_bag \mapsto .., \ldots, Water \mapsto \infty\} \\
CupMax = \ldots
\end{array}
$$

Value is a total function that computes the value (in pence) of a bag of British coins. Bags are used extensively in this specification because they conveniently associate each coin in a collection with the number of times it occurs. A recursive definition of this total function is:

$$
\begin{array}{l}
Value : \text{bag } British\_Coin \rightarrow \mathbb{N} \\
\hline
Value\ [\![\ ]\!] = 0 \\
\forall c : British\_Coin ; n : \mathbb{N} \bullet Value\ [\![c \mapsto n]\!] = Worth\ c * n \\
\forall b_1, b_2 : \text{bag } British\_Coin \bullet Value\ (b_1 \uplus b_2) = Value\ b_1 + Value\ b_2
\end{array}
$$

The value of an empty bag of coins is 0. The value of a bag containing coins of just one denomination is the worth of the coin times the number that we have in the bag. Finally, we make the observation that a bag containing coins of more than one denomination can be regarded as formed by the union of two other bags; one containing coins of only one denomination, the other containing the rest of the coins. We find the value of this first bag and continually apply the observation to the other bag, keeping a running total as we proceed. Recursive calls are terminated by the fact that the value of an empty bag is zero.

We also need to decide whether one bag is a subset of another. This is important

because we can issue the correct change only if the coins needed are currently part of the collection of coins that make up the machine's takings. In the following example the first bag is a subset of the second because the second bag contains the first:

$$\{One\_Penny \mapsto 6\} \sqsubseteq \{One\_Penny \mapsto 7, Five\_Pence \mapsto 6, One\_Pound \mapsto 3\}$$

while in the next example the first bag is *not* contained in the second bag and is therefore not a subset:

$$\neg(\{One\_Penny \mapsto 8\} \sqsubseteq \{One\_Penny \mapsto 7, Five\_Pence \mapsto 6, One\_Pound \mapsto 3\})$$

Bag subset is simply a relation like 'less than' or 'equal to' over the numbers. We define this relation as a generic constant so that it can be used on any bag not just bags of *British_coins*, thereby permitting easy migration of the specification to another country. The following definition says that one bag is a subset of another if for each element, the *count* in the first bag is less than or equal to the *count* in the second. This of course ensures that all elements in the first bag are also in the second.

$[X]$

$$\_ \sqsubseteq \_ : \text{bag } X \leftrightarrow \text{bag } X$$

$$b_1 \sqsubseteq b_2 \Leftrightarrow (\forall x : X \bullet count\ b_1\ x \leq count\ b_2\ x)$$

19.3 THE ABSTRACT STATE OF THE MACHINE

The abstract state of the machine is described by the following schema:

Abs_State_Machine

$$Balance, Takings : \text{bag } British\_coin$$
$$Stock : \text{bag } Ingredient$$
$$Cups : \mathbb{N}$$
$$Prices : Drink \rightarrow \mathbb{N}$$
$$Service\_light : Onoff$$
$$Coin\_display : \mathbb{N}$$
$$Report\_display : Message$$

$$\forall c : \text{dom } Takings \bullet 0 \leq count\ Takings\ c \leq HopperMax\ c$$
$$\forall i : \text{dom } Stock \bullet 0 \leq count\ Stock\ i \leq StockMax\ i$$
$$0 \leq Cups \leq CupMax$$
$$Coin\_display = Value\ Balance$$

Balance and *Takings* are both bags of *British_coin* enforcing the point that no other coins are permitted in the machine. *Balance* simply records the combination of coins that has been inserted by the current customer. Inserted coins are immediately added to *Takings*, which always represents the total amount of money in the machine.

Stock is a bag that records the current stock levels of each of the ingredients, while *Cups* records the number of plastic cups in the machine. *Prices* is a total function from *Drink* to $\mathbb{N}$ showing that all drinks have an associated price. This function can be updated if required to reprice the drinks. *Service_light* is a state variable which is either *on* or *off*, while *Coin_display* and *Report_display* correspond to the displays on the front of the machine (see Fig 19.1). Both *Coin_display* and *Report_display* are taken as state variables to convey the notion of 'persistence' associated with an LED.

In this design we need separate state variables for *Balance* and *Coin_display* even though they always represent the same amount. *Balance* is kept as a separate variable to permit the machine to keep track of the combination of coins the customer inserted. This bag is returned intact when the refund button is pressed. Customers would not be happy if after inserting, say, a 20 pence coin for a drink and then asking for a refund they received twenty 1 pence coins!

The invariants record the following facts:

- For each British coin there will be a maximum number that the machine can store in its coin tube.
- For each ingredient there will be some maximum stock value that can be stored in the machine.
- There is a maximum number of cups which the machine can hold.
- The coin display always shows the value of the current balance.

19.4 THE DELTA AND xi NOTATIONS

$\Delta Machine\_State$

$$Abs\_State\_Machine$$
$$Abs\_State\_Machine'$$

$\Xi Machine\_State$

$$Abs\_State\_Machine$$
$$Abs\_State\_Machine'$$

$$Balance' = Balance$$
$$Stocks' = Stocks$$
$$Takings' = Takings$$
$$Cups' = Cups$$
$$Prices' = Prices$$
$$Service\_light' = Service\_light$$
$$Coin\_display' = Coin\_display$$
$$Report\_display' = Report\_display$$

Once again these are the conventional Z definitions but in this specification there is little use for the xi schema because almost all operations write to the report display LED and thereby change it (but see *No_refund* later).

19.5 THE INITIAL STATE OF THE MACHINE

Initial_Machine_State

$$Abs\_State\_Machine'$$

$$Balance' = [\![\,]\!]$$
$$Stocks' = [\![\,]\!]$$
$$Takings' = [\![\,]\!]$$
$$Cups' = 0$$
$$Prices' = \{d : Drink \bullet (d, 0)\}$$
$$Service\_Light' = on$$
$$Report\_display' = not\ in\ use$$

In the initial state all the bags are empty, the machine has no cups, the drinks have not been priced, the report display LED says *not in use* and the service light is *on* indicating that a service is required.

19.6 PARTIAL OPERATIONS FOR THE DRINKS MACHINE

The partial operations are:

- *Service_Machine*
- *Insert_coin*
- *Get_drink*
- *Refund*
- *Take_ profit*

In all of these the condition of *Coin_display* is always determined by the state invariant:

$$Coin\_display = Value\ Balance$$

This predicate migrates to all operations through schema inclusion and therefore no explicit reference is made to the display in any of the schemas that follow.

19.6.1 Servicing the machine

Service_Machine

$$\Delta Abs\_State\_Machine$$
$$new\_stocks? : \text{bag}\ ingredient$$
$$new\_cups? : \mathbb{N}$$
$$new\_takings? : \text{bag}\ British\_coin$$
$$new\_ prices? : Drinks \pfun \mathbb{N}_1$$

$$Balance' = Balance$$
$$Stocks' = Stocks \uplus new\_stocks?$$
$$Cups' = Cups + new\_cups?$$
$$Takings' = Takings \uplus new\_takings?$$
$$Prices' = Prices \oplus new\_ prices?$$
$$Service\_Light' = off$$
$$Report\_display' = insert\ coin$$

There are no preconditions to a service; the machine can be serviced at any time. The service light does not have to be illuminated nor do we have to run out of ingredients. The role of the service light is simply to indicate when the machine *must* be serviced. Servicing is therefore an operation that always succeeds.

When we service the machine we can add to the number of cups, the stocks of ingredients and to the takings already in the machine. Adding to the takings means that the machine is more likely to be able to give correct change so the service improves its functionality. In the schema bag union is used to deal with these aspects. It should also be pointed out that during a service the levels of any of these items could be left alone. This would be effected by using inputs such as $new\_cups? = 0$ or $new\_stocks? = [\![\,]\!]$. The prices of the drinks can also be changed during a service if required by overriding the old *Prices* function with *new_prices?* Leaving prices unchanged on a service is indicated by using $new\_prices? = \{\}$.

Servicing does not alter the *Balance* in the machine. If someone puts in money without pressing for a drink or a refund then the next person gets some good fortune. Because servicing can only *add* to the machine, servicing is distinguished from profit taking, which is regarded as a separate operation (see Sec. 19.6.5). Finally, once the machine has been serviced the service light is *off* and the report display LED reads *insert coin* indicating its readiness for use.

19.6.2 Inserting coins

Insert_Coin

Δ *Abs_State_Machine*
$c? : Coin$

$c? \neq Unacceptable\_coin$
$count\ Takings\ c? < HopperMax\ c?$
$Balance' = Balance \uplus \{(c? \mapsto 1)\}$
$Takings' = Takings \uplus \{(c? \mapsto 1)\}$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = Service\_light$
$Report\_display' = insert\ coin$

The schema *Insert_coin* describes the process of inserting a single coin into the machine. The process of building a balance is therefore seen as the repeated application of the insert coin operation. An alternative (and more abstract) approach would be to model this process in terms of the machine accepting a bag of coins in a single operation but this is a somewhat less intuitive solution. The precondition(s) for this operation demand(s) that:

- The coin is British.
- The coin tube (hopper) for the coin is not full.

If the coin is accepted both the balance and the takings are increased, while the report

display LED prompts for another coin. All other aspects of the system state remain the same. In this specification therefore, all inserted (acceptable) coins go straight to *Takings*, while *Balance* is simply a variable used to keep track of the combination of coins tendered so far. Consequences of this specification are that *Takings* alone records how much money is in the machine at any point in time, *Value Balance* records how much the current customer is in credit, while all change has to be taken from *Takings*.

19.6.3 Getting a drink

Get_Drink

$$
\begin{array}{l}
\Delta Abs\_State\_Machine \\
choice? : \mathbb{P} Selection\_buttons \\
d! : Drink \\
Change! : \text{bag } British\_coin \\
\hline
choice? \in Drink \\
Value\ Balance \geq Prices\ choice? \\
\forall i : Recipe\ choice? \bullet count\ Stock\ i > 0 \\
Cups > 0 \\
\exists b : bag\ British\_coins \bullet (b \sqsubseteq Takings \land Value\ Balance = Value\ b + Prices\ choice?) \\
Balance' = [\![\,]\!] \\
Stock' \uplus \{i : Recipe\ choice? \bullet i \mapsto 1\} = Stock \\
Cups' = Cups - 1 \\
Change! \sqsubseteq Takings \land Value\ Balance = Value\ Change! + Prices\ choice? \\
Takings' \uplus Change! = Takings \\
Prices' = Prices \\
Service\_light' = Service\_light \\
Report\_display' = insert\ coin \\
d! = choice?
\end{array}
$$

Drinks are chosen by pressing some combination of the selection buttons. The input *choice?* is therefore a set of button presses indicating that the order in which they are pressed is irrelevant. Once a drink has been dispensed the machine issues correct change (if necessary). The preconditions demand that:

- The combination of button presses corresponds to a known drink.
- The balance in the machine is at least enough to pay for the chosen drink.
- All the ingredients are available to make the drink.
- A cup is available.
- The machine can issue correct change if necessary.

This last precondition is interesting and again illustrates a useful technique when handling bags of money. The predicate that has to be satisfied if the correct change is to be issued is:

$$\exists b : bag\ British\_coins \bullet (b \sqsubseteq Takings \land Value\ Balance = Value\ b + Prices\ choice?)$$

which demands that all the coins necessary to make the change are already part of the machine's takings and that the value of the balance in the machine includes the price of

the drink and the value of the change. Although this is a funny way of putting it, it avoids our having to make any commitment to the composition of the bag of coins returned to the customer. This is determined by the machine alone, dependent upon the mix of coins currently in the coin tubes, but at this level of abstraction we do not have to address how this is achieved.

Dispensing a drink is, of course, a physical act and the nearest we can describe it in the specification is by making the output variable *d!* equal to the input *choice?* Once the drink has been dispensed the balance becomes empty, the stock levels of each of the ingredients are reduced by one, the machine has one less cup, the change is issued, the takings have been reduced to reflect the change issued and the report display LED prompts for another coin. The service light and the drink prices are unaffected.

Notice how the specification deals with the reductions in the *Stock* and *Takings* bags. The predicate:

$$Stock' \uplus \{i : Recipe\ choice? \bullet i \mapsto 1\} = Stock$$

is a postcondition that describes the relationship between the input and the stock levels before and after the operation. The condition expressed is that bag union between the stock after and a bag where each of the ingredients of the chosen drink occurs exactly once, is equal to the stock before. This is the same as saying that the stock level of every ingredient of the chosen drink has been reduced by one after the operation. The predicate:

$$Takings' \uplus Change! = Takings$$

describes the reduction in takings in similar fashion.

19.6.4 The refund

Refund

$\Delta Abs\_State\_Machine$
$Refund!$: bag $British\_Coin$

$Balance \neq [\![\ ]\!]$
$Refund! = Balance$
$Refund! \sqsubseteq Takings$
$Takings' \uplus Refund! = Takings$
$Balance' = [\![\ ]\!]$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = Service\_light$
$Report\_display' = insert\ coin$

Customers can demand a refund from the machine at any time. The operation has just one precondition:

- The balance is not empty.

which really corresponds to customers inserting coins and then changing their minds. It is important from a user-friendly point of view that the machine returns the same bag of coins that the customer input and this is reflected by the predicate:

$$Refund! = Balance$$

The refund is issued from the takings in the machine which, of course, contains all the coins the customer inserted (see *Insert_coin* earlier). The predicate:

$$Refund! \sqsubseteq Takings$$

is therefore really superfluous but remains for the sake of clarity. Once a refund has been issued, the balance is reset and the report display LED prompts for another coin. Stocks, cups, prices and the service light remain unaffected, while the *Takings* are reduced to reflect the refund given.

19.6.5 Taking profit

Take_profit

$\Delta Abs\_State\_Machine$
$Profit!$: bag *British_Coin*

$Takings' \uplus Profit! = Takings$
$Balance' = [\![\ ]\!]$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = Service\_light$
$Report\_display' = insert\ coin$

Taking profit, like servicing, has no preconditions and will always succeed. However, unlike servicing, this operation 'profiteers' in the sense that any balance owing to the customer is removed by including it in the takings, i.e. $Balance' = [\![\ ]\!]$. We cannot remove money that is not in the machine for our bag operators are not defined on negative occurrences (however see Hayes and Jones (1989)) and a profit of zero is possible if no one has used the machine in between the profit-taking operations.

When taking profit we can remove all the money in the machine but this will prevent the machine from being used if the next customer does not tender the exact money.

SAQ 19.2 How would you express this property as a theorem?

Taking profit leaves cups, stocks, prices and the service light unchanged but writes the message *insert coin* to the report LED. Notice again how the first predicate defines the takings afterwards by demanding that the bag union for $Takings'$ and *Profit!* equals the original *Takings*. Once again this technique does not dictate any particular coin composition for the profit.

19.7 TOTAL OPERATIONS FOR THE DRINKS MACHINE

As usual the total operations are formed from the partial operations by considering what happens when their preconditions are offended. Previous case studies used the schema *Success* as part of the definition of the total operations. This study departs from this practice because the report display LED really achieves the same thing in that the *insert coin* prompt shows that the previous operation has been successful. Similarly, the usual role for the output variable *Report!* has been assumed by the LED.

There are no preconditions for *Service* or *Take_profit* and so these operations are already total. The preconditions for the *Insert_coin* operation, however, show that it can fail in a number of ways:

- The coin inserted is unacceptable (i.e. it is not British).
- The coin tube (hopper) for the coin is full.

The following schemas describe what happens in these circumstances:

Reject_coin

$\Delta Abs\_State\_Machine$
$c?, c! : Coin$

$c? = Unacceptable\_coin$
$Takings' = Takings$
$Balance' = Balance$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = Service\_light$
$Report\_display' = try\ another\ coin$
$c! = c?$

Hopper_full

$\Delta Abs\_State\_Machine$
$c?, c! : Coin$

$count\ Takings\ c? = HopperMax\ c?$
$Takings' = Takings$
$Balance' = Balance$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = Service\_light$
$Report\_display' = try\ another\ coin$
$c! = c?$

and so the total operation of inserting a coin is defined by:

$$Total\_Insert\_coin \triangleq Insert\_coin \lor Reject\_coin \lor Hopper\_full$$

Notice that the *Hopper_ full* operation does not signal a service. Because the machine gives change, it is capable of reducing the contents of popular coin tubes by being biased towards certain coinage mixes. In the total operation coins are rejected ($c! = c?$) and returned to the customer if the preconditions fail.

By examining the preconditions for the *Get_drink* operation we see that it too can fail in a number of ways:

- The combination of selection buttons pressed by the customer does not correspond to a known drink.
- The customer has not inserted enough money.
- The machine is out of stock(s) for the chosen drink.
- The machine has run out of cups.
- The exact change is unavailable.

The schemas that deal with each of these events are as follows:

Drink_not_known

$\Delta Abs\_State\_Machine$
$choice? : \mathbb{P} Selection\_buttons$

$\neg (choice? \in Drink)$
$Takings' = Takings$
$Balance' = Balance$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = Service\_light$
$Report\_display' = this\ drink\ is\ unavailable$

Not_enough_money

$\Delta Abs\_State\_Machine$
$choice? : \mathbb{P} Selection\_buttons$

$Value\ Balance < Prices\ choice?$
$Takings' = Takings$
$Balance' = Balance$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = Service\_light$
$Report\_display' = insert\ more\ money\ and\ select\ drink\ again$

Out_of_stock(s)

$\Delta Abs\_State\_Machine$
$choice? : Selection\_buttons$

$\exists i : Receipe\ choice? \bullet count\ Stock\ i = 0$
$Takings' = Takings$
$Balance' = Balance$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = on$
$Report\_display' = this\ drink\ is\ unavailable$

Out_of_cups

$\Delta Abs\_State\_Machine$
$choice? : \mathbb{P} Selection\_buttons$

$Cups = 0$
$Takings' = Takings$
$Balance' = Balance$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = on$
$Report\_display' = no\ cups\ in\ machine$

No_change_available

$\Delta Abs\_State\_Machine$
$choice? : Selection\_buttons$

$(Value\ Balance \geq Prices\ choice? \wedge$
$\neg \exists b : British\_coins \bullet (b \sqsubseteq Takings \wedge Value\ Balance = Value\ b + Prices\ choice?))$
$Takings' = Takings$
$Balance' = Balance$
$Stocks' = Stocks$
$Cups' = Cups$
$Prices' = Prices$
$Service\_light' = Service\_light$
$Report\_display' = correct\ change\ unavailable$

Most of the schemas are fairly straightforward but some comment may be appropriate for the preconditions of *Out_of_stock(s)* and *No_change_available*. The predicate:

$$\exists i : Recipe\ choice? \bullet count\ Stock\ i = 0$$

says that there exists at least one ingredient of the drink which is out of stock, while:

$$Value\ Balance \geq Prices\ choice? \wedge$$
$$\neg \exists b : British\_coins \bullet (b \sqsubseteq Takings \wedge Value\ Balance = Value\ b + Prices\ choice?)$$

says that although the balance is sufficient to pay for the drink there is no bag of coins which is currently part of the machine's takings and is of value equal to the change required.

Finally, the total operation can be related to the process of pressing button D on the control panel such that the non-deterministic definition of this operation becomes:

$$\begin{aligned} Press\_button\_D \triangleq\ & Get\_drink \vee Drink\_not\_known \\ & \vee Not\_enough\_money \\ & \vee Out\_of\_stock(s) \\ & \vee Out\_of\_cups \\ & \vee No\_change\_available \end{aligned}$$

All of the total operations so far leave the balance in the machine if the preconditions fail. To recover money the customer must press button R for a refund. However, the precondition for *Refund* requires that the balance is *not* empty. If a customer requests a refund when the balance is empty the status quo is preserved. Thus:

No_refund

$\Xi Machine\_State$

$Balance = [\![\ ]\!]$

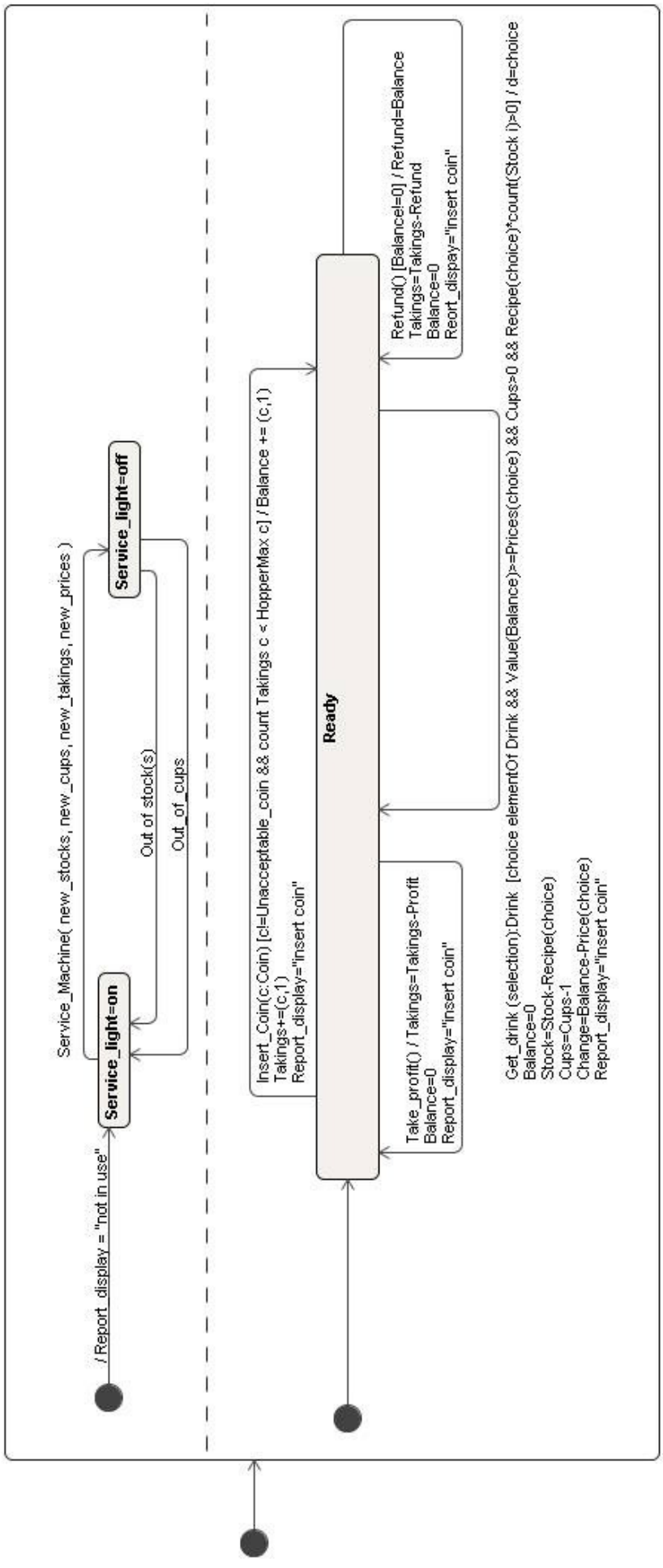
The total operation for a refund can therefore related to pressing button R:

$$Press\_button\_R \triangleq Refund \vee No\_Refund$$

19.8 SUMMARY

In this case study the emphasis has been centred largely on the use of bags to specify the properties of a familiar real-world object. Bags are rather neglected members of the mathematical toolkit but in many circumstances—especially where monetary amounts or stock levels are concerned—they present an ideal model. The real drinks dispensing machine would have to have quite sophisticated coin recognition mechanisms and elaborate algorithms to determine optimum coin compositions for change given current machine takings. Using bags we are able to capture the essential behaviour of these aspects without committing ourselves to particular implementations. A number of these are well known within the automatic catering industry and the manufacturer is free to employ the most appropriate given the intended operating characteristics of the machine. The next part of the book reviews the complementary method VDM. The drinks dispenser will be visited again so that direct comparison can be made between the two approaches.

Anlage 2: UML- Zustandsautomat zu Kapitel 3.2



Literaturverzeichnis:

Barwise J. / Etchemendy J. (2005) — Sprache, Beweis und Logik. Band I. Aussagen- und Prädikatenlogik. Mentis-Verlag 2005

Ebbinghaus, H.-D. (2003) — Einführung in die Mengenlehre. Mit Aufgaben und Lösungshinweisen. 4. Auflage. Spektrum Akademischer Verlag 2003

Ficker, H.-J. / Jarré, S. et al. (1998) — Z-Beispiel Nr.2. http://www.informatik.uni-bremen.de/agbs/live/home/soenke/referat.ps. 22.02.2008

Jacky, J. (1997) — The Way of Z. Practical Programming with Formal Methods. Cambridge University Press 1997

Jeckle, M. / Rupp, C. et al. (2005) — UML 2 glasklar. Praxiswissen für die UML-Modellierung und -Zertifizierung. 2. Auflage. Carl Hanser Verlag München Wien 2005

Lano, K. (1990) — Z++, an Object-Oriented Extension to Z. Z User Workshop. Springer Science+Business Media (1990)

Miles, R. / Hamilton, K. (2006) — Learning UML 2.0. A pragmatic Introduction to UML. O'Reilly Media Verlag 2006

Oesterreich, B. (1998) — Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language. 4. aktualisierte Auflage. Oldenbourg Verlag 1998

OMG (Hrsg.) (2003)
UML 2.0 Infrastructure Specification
Version 2.0
The Object Management Group
http://www.omg.org/docs/ptc/03-09-15.pdf
10.02.2008

OMG (Hrsg.) (2005)
Unified Modeling Language: Superstructure
Version 2.0
The Object Management Group
http://www.omg.org/docs/formal/05-07-04.pdf
10.02.2008

Smith, G. (1999)
The Object-Z Specification Language
(Advances in Formal Methods)
2. Auflage
Springer-Verlag 1999

Sheppard, D. (1995)
An Introduction to Formal Specification with Z and VDM
McGraw-Hill Publishing Co. 1995

Spivey, J.M. (1988)
Understanding Z: A Specification Language and its
Formal Semantics
Cambridge University Press 1988

Spivey, J.M. (1992)
The Z Notation: A Reference Manual
Second Edition
http://spivey.oriel.ox.ac.uk/mike/zrm/zrm.pdf
01.02.2008