Grundlagen der Systementwicklung[1]

WS 2000/01

M. Wirsing, S. Merz

Diese Vorabversion des Skripts ist noch nicht korrigiert.
Die Leser werden um Kommentare und Verbesserungsvorschläge gebeten.

[1] Skript zur Vorlesung ausgearbeitet von Johanna Martin auf der Basis der Vorlesungsfolien.

Inhaltsverzeichnis

I Systementwicklung und Formale Spezifikationen 6

1 Einführung 6

1.1 Der Software-Lebenszyklus . . . 7
1.2 Probleme bei der System-Entwicklung . . . 10
1.3 Formale Methoden in der Informatik . . . 11
1.4 Zusammenfassung . . . 14

II Datenorientierte Systementwicklung 16

2 Signaturen zur Schnittstellenbeschreibung 16

3 Funktionale Modelle: Schnittstelleninterpretation durch Strukturen 19

3.1 Algebren und Strukturen . . . 19
3.2 Σ-Formeln . . . 22
3.2.1 Syntax von Formeln . . . 23
3.2.2 Semantik von Formeln . . . 24
3.2.3 Wahrheitstabellen . . . 24
3.2.4 Aussagenlogische und prädikatenlogische Umformungen . . . 25
3.2.5 Beweissysteme für die Prädikatenlogik . . . 26
3.2.6 Allgemeingültigkeit von Gleichungen: Der Kalkül von Birkhoff . . . 28
3.3 Eigenschaften von Algebren: Homomorphismen, initiale und erreichbare Strukturen . . . 29
3.4 Eine Verallgemeinerung von Initialität: Freie Erweiterung . . . 37
3.5 Zusammenfassung . . . 38

4 Algebraische Spezifikation mit CASL 39

4.1 Einfache algebraische Spezifikationen . . . 39
4.2 Spezifikationen mit Konstruktoren . . . 40
4.3 Spezifikation initialer und freier Strukturen . . . 46
4.4 Spezifikation grundlegender Rechenstrukturen . . . 47
4.4.1 Keller . . . 47
4.4.2 Binäre Bäume mit Knotenmarkierungen . . . 49
4.4.3 Lose endliche Mengen . . . 49
4.5 Strukturierte Spezifikationen . . . 51
4.5.1 Summe zweier Spezifikationen . . . 51
4.5.2 Erweiterung . . . 52

4.5.3 Kapselung durch „Verstecken“von Symbolen 52
4.5.4 Umbenennung 53
4.6 Parametrisierung von Spezifikationen 57
4.7 Zusammenfassung 61

5 Verfeinerungstechniken 62
5.1 Der Verfeinerungsbegriff: Verfeinerung durch Modellklassen-Inklusionen 62
5.2 Wechsel der Datenstruktur 67
5.3 Exkurs: Kurze Einführung in SML 74
5.3.1 Typen 75
5.3.2 Werte und Funktionen 78
5.3.3 Ausnahmen 79
5.3.4 Signaturen 79
5.3.5 Strukturen 80
5.4 Übergang zu funktionalen Programmen 81
5.5 Zusammenfassung 83

III Spezifikation zustandsbasierter Systeme 85

6 Zustände und Transitionssysteme 85

7 Modellorientierte Spezifikationen am Beispiel von Z 88
7.1 Die Spezifikationssprache Z 88
7.2 Grundrechenstrukturen und Logik von Z 89
7.2.1 Logik 89
7.2.2 Mengenlehre 90
7.2.3 Relationen 92
7.2.4 Funktionen 92
7.2.5 Sequenzen 93
7.3 Grundschemata 93
7.4 Schemakombination 95
7.4.1 Schemainklusion 96
7.4.2 Logische Kombination von Schemata 96
7.4.3 Dekorationen 99
7.5 Zusammenfassung 103

8 Spezifikationsentwicklung in Z **104**
8.1 Verfeinerung von Operationen . . . 104
8.2 Wechsel der Datenstruktur . . . 108
8.3 Verifikationsbedingungen für Verfeinerungen . . . 110
8.4 Übergang zu imperativen Programmen . . . 115
8.5 Zusammenfassung . . . 122

IV Reaktive Systeme **124**

9 Abläufe von Transitionssystemen **124**
9.1 Begriff, Beispiele . . . 124
9.2 Fairnessbedingungen . . . 129
9.3 Eigenschaften von Abläufen . . . 132
9.4 Zusammenfassung . . . 138

10 Spezifikation von Transitionssystemen mit TLA **138**
10.1 Die Logik TLA . . . 139
10.1.1 Zustandsformeln (state predicates) . . . 140
10.1.2 Aktionsformeln (actions, transition predicates) . . . 141
10.1.3 Temporallogische Formeln (temporal formulas) . . . 142
10.2 Spezifikationsstile in TLA . . . 144
10.2.1 Einfache Interleaving-Spezifikationen . . . 145
10.2.2 Interleaving-Spezifikationen mit synchroner Kommunikation . . . 145
10.2.3 Interleaving-Spezifikationen mit asynchroner Kommunikation . . . 146
10.2.4 Noninterleaving-Spezifikationen . . . 147
10.2.5 Zusammenfassung . . . 148
10.3 Verifikationsregeln . . . 148
10.3.1 Beweise von Invarianten . . . 148
10.3.2 Lebendigkeit 1: Ausnutzen von Fairnessbedingungen . . . 150
10.3.3 Lebendigkeit 2: fundierte Ordnungen . . . 154
10.3.4 Temporallogische Gesetze . . . 155
10.4 Zusammenfassung . . . 158

11 Verfeinerung und Strukturierung **158**
11.1 Ablaufverfeinerung . . . 159
11.2 Komposition von Spezifikationen . . . 162
11.3 Kapselung . . . 163
11.3.1 Erweiterung von TLA um Quantoren . . . 164

11.3.2 Stotterinvarianz von TLA-Formeln 165
11.3.3 Semantik von Quantoren über flexible Variablen 166
11.3.4 Verfeinerungsabbildungen (refinement mappings) 167
11.4 Zusammenfassung 169

12 Analyse von Transitionssystemen durch Modelchecking **169**
12.1 Überprüfung von Invarianten 169
12.2 Büchi-Automaten 175
12.3 Automatenbasiertes Modelchecking 179
12.4 Abstraktionstechniken 183
12.5 Zusammenfassung 186

Teil I

Systementwicklung und Formale Spezifikationen

Ziele

- Phasenmodell der Systementwicklung und die Rolle von Validierung und Verifikation verstehen
- (Technische) Probleme bei der Systementwicklung aufzeigen
- Historische Übersicht über formale Methoden zur Systementwicklung geben.

1 Einführung

Thema dieser Vorlesung sind Grundlagen der Entwicklung von sicherheitskritischen Softwaresystemen und von technischen Systemen mit einem hohen Anteil an Software. Bei solchen Systemen ist es wichtig, dass die eingesetzte Software von hoher Qualität ist und möglichst fehlerfrei arbeitet. Dies kann nur durch den systematischen Einsatz rigoroser und möglichst formaler Methoden erreicht werden, da nur mit auf mathematisch präzisen Notationen basierenden Beschreibungen von Anforderungen und Entwürfen Programme als fehlerfrei nachgewiesen werden können. Wir werden uns in dieser Vorlesung auf die grundlegenden formalen Methoden und Techniken konzentrieren, die heute die Basis für die Beschreibung und Untersuchung von Datenstrukturen, zustandsbasierten Systemen und dynamischen reaktiven Systemen sind. Mehr pragmatische Aspekte der Software-Entwicklung werden in der parallel stattfindenen Vorlesung über Objekt-Orientierte Software-Entwicklung untersucht. In der Vorlesung Formale Objekt-Orientierte Software-Entwicklung wird gezeigt, wie die formalen Techniken dieser Vorlesung mit pragmatischen Engineering-Techniken wie UML integriert werden können. Diese pragmatischen Techniken werden geübt und vertieft im Software-Engineering-Praktikum. Eine weitergehende Untersuchung des dynamischen Verhaltens von Systemen bietet die Vorlesung Temporale Logik, die durch ein Praktikum über Modelchecking ergänzt wird.

Beispiel 1: Bedeutung korrekter Software

- Am 22. Juli 1962 wurde eine Trägerrakete mit dem unbemannten Raumschiff Mariner I 290 Sekunden nach dem Start zerstört. Der Verlust betrug 18–20 Millionen US-Dollar. Das Programm im Bodencomputer hätte folgendes Fragment enthalten sollen:

  if *not* in radar contact with the rocket
  then
  do not correct its flight path

  Aus Versehen war das „not“ vergessen worden, deshalb konnte der Bordcomputer die Flugbahn der Rakete nicht mehr korrigieren. Sie kam ins Trudeln und musste zerstört werden, bevor sie Menschen gefährdete. Das Programm war vorher getestet und in vier Mondraketen ohne Fehler benutzt worden. Der Vorsitzende des Untersuchungsausschusses fragte: „Wer war verantwortlich?“ Er erhielt folgende Antwort: „Das Programm war in 300 Testläufen erprobt werden. Es ist nicht möglich, die Arbeit jedes Programmierers nachprüfen zu lassen!“

- Ähnliche Softwarefehler führten im Frühjahr 1996 zur Zerstörung der Trägerrakete Ariane 5, die Satelliten im Wert von mehr als 1 Milliarde Dollar an Bord hatten.

- Die deutsche **Telekom** verlor sehr viel Geld bei der Einführung neuer Telefontarife am 1. Januar 1996. Das Programm hatte den ersten Januar **nicht** als Feiertag, sondern als Werktag gerechnet. Nach Kundenprotesten gab die Telekom jedem Kunden eine Gutschrift.
- Der **Windows-Taschenrechner** (Version 3.1–8.3) wies lange Zeit folgenden Fehler auf:

$$\begin{array}{r} 2.01 \\ -2.00 \\ \hline 0.00 \end{array}$$

- Bei Therac 25, einem Strahlengerät zur Krebsbehandlung, führte fehlerhafte Programmierung zu Verbrennungen und Todesfällen bei Patienten. ◀

System Engineering beschäftigt sich mit Techniken der Systementwicklung von der ersten informellen Beschreibung eines Problems bis hin zu seiner Lösung als Programm. Der Begriff Software Engineering wurde in den Jahren 1967–1968 geprägt, als sich herausstellte, dass zur Konstruktion qualitativ guter Programme systematische Techniken eingesetzt werden müssen. Heute spricht man allgemeiner von System Engineering, da im Allgemeinen nicht nur die Software, sondern das gesamte System einschließlich seiner technischen und menschlichen Aktoren und seiner Arbeitsabläufe und -prozesse berücksichtigt werden müssen.

Eine der früheren Forderungen des Software Engineering war es, den gesamten Programmentwurfsprozess mit formalen Methoden durchzuführen oder zumindest zu unterstützen. Dazu gehört die formale Spezifikation von Programmen und ihre Verifikation bezüglich dieser Spezifikationen. Zumindest die sicherheitskritischen Teile eines Programmes sollten so verifiziert sein, dass solche Katastrophen wie die oben genannten nicht auftreten können.

1.1 Der Software-Lebenszyklus

Kommerzielle Programme werden in Teams von mehreren Programmierern erstellt. Es ist nicht üblich, dass eine Person am gesamten Entwicklungsprozess teilnimmt. Deswegen müssen Methoden gefunden werden, um Aufgaben zu koordinieren und zu unterteilen.

Aus diesem Grunde unterteilt man den System-Entwicklungsprozess in Phasen wie in Abbildung 1.

Die einzelnen Begriffe des Diagramms lassen sich folgendermaßen kurz charakterisieren:

Anforderungsanalyse und -spezifikation. Erstellung eines Dokuments, das eine Beschreibung enthält, *was* das System tun soll, aber nicht, *wie* es das tun soll.

Systementwurf und -spezifikation. Spezifikation der Architektur des Systems und der Aufgaben seiner Komponenten.

Detaillierter Entwurf. Verfeinerung des Entwurfs bis zum einem Detaillierungsgrad, der es ermöglicht, den Code zu erstellen.

Implementierung. Codierung der Komponenten des Entwurfs in einer konkreten Programmierspache.

Integration. Zusammenführung der einzelnen Systemkomponenten zum Gesamtsystem.

Wartung. Fehlerkorrektur und Änderung des fertigen Systems während des Praxiseinsatzes.

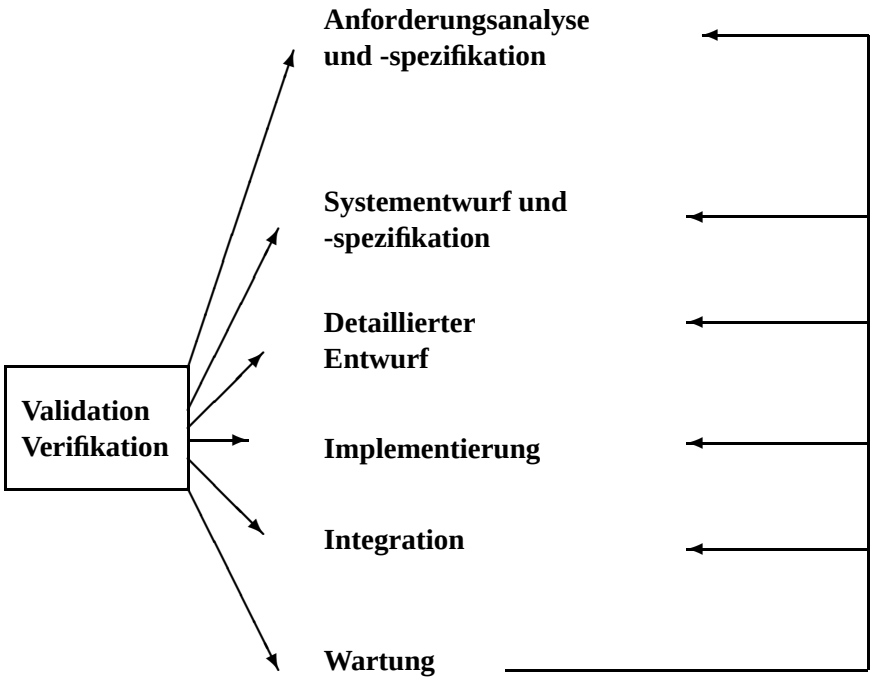


Abbildung 1: System-Entwicklungsprozess

Validierung. Prüfung, ob ein Produkt die Benutzeranforderungen erfüllt. („Bauen wir das richtige Produkt?“)

Verifikation. Prüfung, ob jede Phase der System-Entwicklung die Intentionen der vorhergehenden Phase erfüllt. („Bauen wir das Produkt richtig?“)

Testen. Prozess des Ausführens eines vollständigen Systems zur Überprüfung, dass es die Anforderungen erfüllt; Teil der Validierung.

Das Testen erfolgt häufig erst sehr spät im Entwicklungsprozess, da eine Implementierung dafür vorliegen muss. Ein Ziel formaler Methoden ist es, Validierung auch in früheren Phasen der Entwicklung durchführen zu können. Techniken dazu sind etwa: Simulation, Prototyping, Symbolische Ausführung, statische und dynamische Analyse.

Die Vorlesung beschäftigt sich vor allem mit Anforderungs- und Entwurfsspezifikation! Eine detailliertere Behandlung der pragmatischen Konzepte der Software-Entwicklung findet sich in der Vorlesung „Objekt-Orientierte Software-Entwicklung“.

Beispiel 2: Analogie zum Bau einer Autobahnbrücke

- Anforderungsanalyse:
  - Größe, Länge, Höhe,
  - Kapazität,
  - Bodenbeschaffenheit,
  - Anforderungsspezifikation: Technisches Dokument.
- Systemspezifikation:

  - Entwurf der Struktur der Brücke,
  - bei einfacheren Lösungen unter Benutzung von Standard-Komponenten (zur Kosten- und Zeiteinsparung).

- Wartung:
  - kleine Entwurfs- und Baufehler werden während der Benutzung beseitigt,
  - bei größeren Problemen: z.B. Neukonstruktion der Brücke, oder wie kürzlich in London die Sperrung einer neuen Brücke. ◄

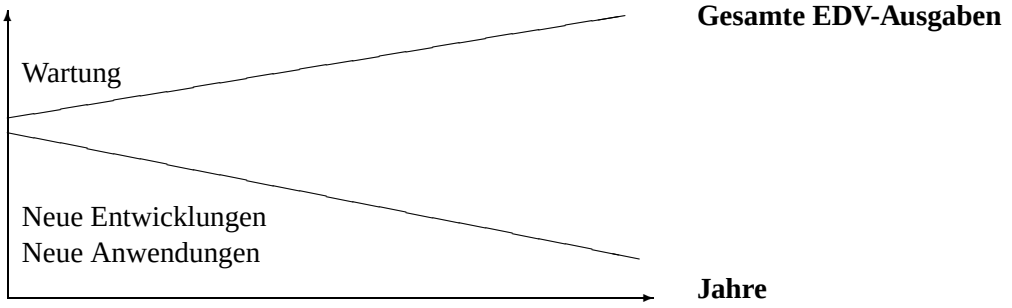
Beispiel 3: Kosten von Software im Lebenszyklus

Während Hardware immer billiger wird, nehmen die Kosten für Software zu. Wir unterscheiden zwei Arten von Kosten:

a) Kosten der Software-Konstruktion,

b) Kosten der Wartung sowohl für Fehlerkorrektur als auch Änderung der Anforderungen.

Die Kosten von a) betragen 20% oder weniger, die Kosten der Wartung b) mindestens 80%.



◄

Wiederverwendbarkeit

Vorteile:

- spart Zeit und Kosten,
- weniger fehleranfällig,
- länger verwendete Software ist robuster,
- Wiederverwendbarkeit auf allen Ebenen insbesondere von Spezifikationen.

Probleme:

- „not invented here“ Syndrom, d.h. man traut der fremden Software nicht,
- Wiederverwendung ist schwierig, wenn der Code an das neue Problem angepasst werden muss,
- die Entwicklung generischer Software ist aufwendig.

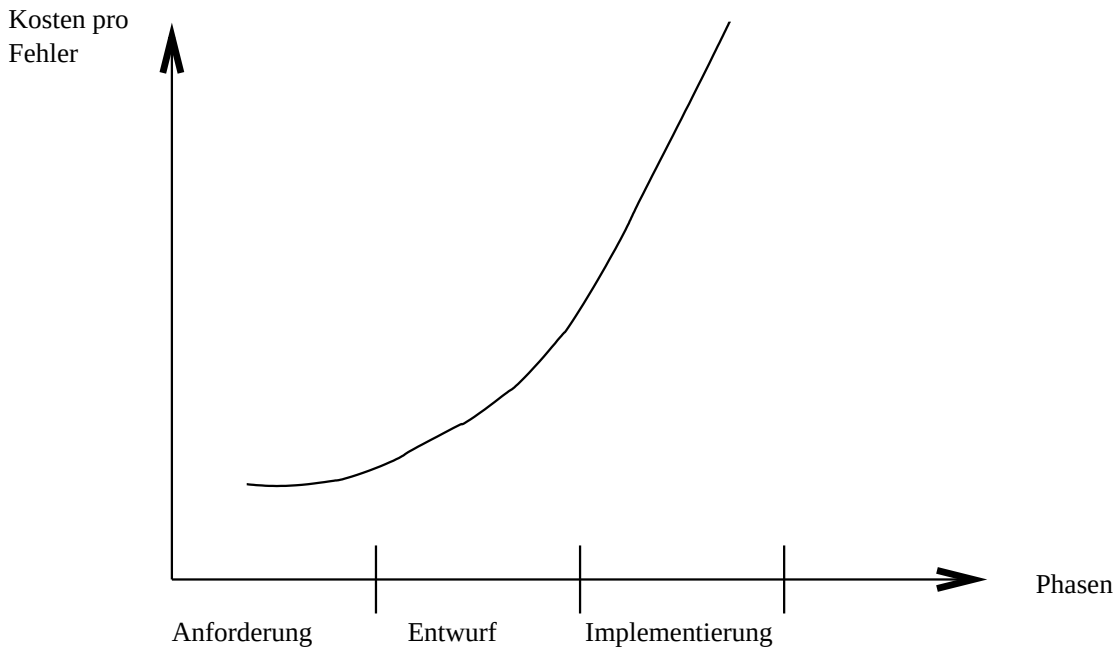
In den letzten Jahren hat sich die Verwendung von Programmbibliotheken (API) durchgesetzt, durch bessere Notationen (z.B. Java beans) nimmt auch der Einsatz von Komponenten zu.

1.2 Probleme bei der System-Entwicklung

Entwicklungsschritt	Notation
Anforderungsspezifikation	in natürlicher Sprache mit Bildern und Tabellen
Entwurfsspezifikation	in graphischer Notation und in natürlicher Sprache; formale Spez. bei sicherheitskritischen Systemen
Detaillierte Spezifikation	in Programmnotation vermischt mit natürlicher Sprache
Implementierung	in Programmiersprache

Tabelle 1: typischer Entwicklungsprozess

1. Natürliche Sprache ist nicht präzise genug, zu viele Details
2. Zu viele verschiedene Notationen, häufig in jeder Phase andere Notation. Fehler treten insbesondere auf an den Schnittstellen zwischen den Phasen.
3. Fehler in der Anforderungsdefinition oder im Entwurf sind meist schwerer zu beheben und damit teuerer als Fehler in späteren Phasen.



Beispiel 4:

Mangel an Präzision.

- „Die Schnittstelle zum System, das von Radaroperatoren benutzt wird, soll benutzerfreundlich sein. Die virtuelle Schnittstelle soll auf wenigen einfachen allgemeinen Konzepten beruhen, die eine klare Benutzerführung bieten und klar zu verstehen sind.“
- „Die Voltzahl der Lampe ist immer eine ganze Zahl zwischen 3 und 6 Volt.“
  Heißt dies „Voltzahl ≥ 3 und Voltzahl ≤ 6“ oder „Voltzahl > 3 und Voltzahl < 6“?

Unerfüllbarkeit. „Der Wasserstand der letzten drei Monate soll auf einem *magnetischen Band* gespeichert werden.“ (dies kann ein Teil der Hardwareanforderungen eines zukünftigen Systen sein) und der Satz „Das Kommando `PRINT_LEVEL` druckt den durchschnittlichen Wasserstand für einen spezifizierten Tag der letzten drei Monate aus. Die Antwortzeit soll nicht länger als *drei Sekunden* sein.“

Unvollständigkeit. „Das System soll die stündlichen Temperaturen überwachen, die von Sensoren an den laufenden Reaktoren gemessen werden. Diese Werte sollen für die letzten drei Monate gespeichert werden. Die Funktion des Kommandos `AVERAGE` ist es, auf einer Benutzerkonsole die durchschnittliche Temperatur eines Reaktors für einen spezifizierten Tag anzuzeigen.“

Was wird angezeigt, wenn um 14 Uhr das Mittel der heutigen Temperatur verlangt wird: *error* oder Mittel der Temperatur bis 14 Uhr?

Zweideutigkeit. „Das Operatorkennzeichen besteht aus dem Operatornamen und seinem Passwort; das Passwort besteht aus sechs Ziffern. Es soll auf der Sicherheitskonsole angezeigt werden und in der Logindatei gespeichert werden, wenn ein Operator sich einloggt. Wenn ein Fehler bei einer Reaktorüberlastung entdeckt wird, soll der Fehlerbildschirm 1 auf der Masterkonsole angezeigt werden und der Fehlerbildschirm 2 soll auf der Verbindungskonsole angezeigt werden mit einer blinkenden Kopfzeile.“

Bezieht sich „es“ auf Passwort oder Operatornamen? Sollen beide Konsolen blinkende Kopfzeilen haben? ◄

1.3 Formale Methoden in der Informatik

Der Begriff „Formale Methoden“ bezeichnet die Verwendung von mathematischen Notationen wie Logik, Algebra und Mengenlehre zur Beschreibung von Anforderungs- und Entwurfsspezifikationen zusammen mit Validierungs- und Verifikationstechniken basierend auf Mathematik. Geschichtlich:

ab ca. 1960: formale Sprachen und Automaten

Untersuchung der Syntax von formalen Sprachen. Operationelles Verhalten von Programmen: Chomsky-Hierarchie, endliche Automaten, reguläre Ausdrücke und Turing-Maschinen.

1970: Semantik von Programmiersprachen (Scott, Strachey)

Zuordnung einer Bedeutung zur Syntax durch Abbildung auf bekannte mathematische Strukturen: Semantik des λ-Kalküls, denotationelle Semantik.

1969–1975: Beweise von Programmen, Zusicherungskalküle (Dijkstra, Hoare, Floyd)

Zusammenhang von Programmen mit logischen Eigenschaften vor und nach der Ausführung des Programms.

ab ca. 1975: Formale Spezifikation (Guttag, ADJ)

abstrakte Beschreibung von Programmen/Problemeigenschaften durch Axiome („algebraische bzw. axiomatische Spezifikation“), bzw. basierend auf Modellen und Mengenlehre („modellorientierte Spezifikation“).

ab ca. 1970: Formale Programmentwicklung (CIP, Burstall/Darlington, Hoare)

Studium der Beziehungen zwischen Abstraktionsebenen der Beschreibung, Programmtransformationen, Wechsel der Datenstruktur.

ab ca.1980: Einsatz temporaler Logik zur Beschreibung des dynamischen Verhaltens von Systemen.

ab ca.1990: Integration pragmatischer graphischer Notationen mit formalen Techniken, semantische Fundierung graphischer Notationen, neue Analysetechniken.

ab ca.1995: große Erfolge der Temporallogik durch Einsatz von Modelchecking (endlicher Automaten) zur Verifikation/Falsifizierung von Sicherheits- und Lebendigkeitseigenschaften von Kontrollsystemen.

Bei Spezifikationen unterscheidet man vor allem modellorientierte Spezifikationen (auch als zustandsorientierte Spezifikationen bezeichnet) und eigenschaftsorientierte, insbesondere algebraische Spezifikationen. Zur Beschreibung dynamischer Systeme benötigt man noch zusätzliche Ausrucksmöglichkeiten, die durch temporale Logik bereitgestellt werden. Von den modellorientierten Methoden kennt und verwendet man heute vor allem VDM [Jones 90], B und Z. Bei den algebraischen Ansätzen handelt es sich um Larch, LOTOS, Maude und eine neue Sprache CASL.Bei den temporallogischen Ansätzen verwendet man u.a. LTL, CTL, TLA. Die Formalismen CASL, Z und TLA werden in dieser Vorlesung noch detaillierter behandelt weden.

Modell-orientierte Spezifikationen zielen auf die Beschreibung imperativer Konzepte durch Zusicherungen der Form

$$\{\Phi\}\ P\ \{\Psi\}$$

Beispiel 5: Z.B. ist die Zusicherung

$$\{n \geq 0\}\ s := n;\ Q\ \{s = fac(n)\}$$

eine Spezifikation eines Programms Q, das die Fakultätsfunktion berechnet. In dieser Spezifikation ist s eine Programmvariable, n eine „mathematische“ oder „logische“ Variable, deren Wert während der Ausführung von Q nicht verändert werden darf. *fac* bezeichnet die Fakultätsfunktion, deren Bedeutung „bekannt“ ist. Beachten Sie, dass die Spezifikation (bewusst) unvollständig ist, da Q nur auf nichtnegative Werte angewendet werden soll. ◄

Axiomatische Spezifikationen zielen auf das funktionale Verhalten von Rechenstrukturen. Eine Datenstruktur wird beschrieben durch ihre Schnittstelle oder Signatur, d.h. durch Angabe (der Namen) der involvierten Datentypen und ihrer charakteristischen Operationen sowie der Angabe der charakteristischen Eigenschaften durch (Gleichungs-) Axiome.

Beispiel 6: Die Rechenstruktur der Keller ist gegeben durch Namen für den Datentyp Stack der Keller und den Datentyp Data der Kellerelemente. Charakteristische Funktionen sind

$$\begin{aligned}
&\text{empty} : \text{Stack}\\
&\text{push} : \text{Data} \times \text{Stack} \rightarrow \text{Stack}\\
&\text{top} : \text{Stack} \rightarrow \text{Data}\\
&\text{pop} : \text{Stack} \rightarrow \text{Stack}
\end{aligned}$$

Charakteristische Eigenschaften sind

$$\begin{aligned}
&\text{top}(\text{push}(d,s)) = d\\
&\text{pop}(\text{push}(d,s)) = s
\end{aligned}$$

Eine imperative Version der Kellerstruktur enthält die folgenden **Prozeduren**:

New(s) weist der Variablen s den leeren Keller zu,

Push(s,n) fügt das Symbol n zum Keller hinzu,

Pop(s) entfernt das oberste Kellersymbol,

Top(s,m) weist den obersten Eintrag des Kellers s der Variablen m zu,

Isempty(s,b) weist der Variablen b den Wert *true* zu, falls s leer ist, andernfalls erhält b den Wert *false*.

Im modell-orientierten Ansatz legt man eine Struktur zur Realisierung von Kellern fest, z.B. endliche Sequenzen. Eine Spezifikation kann dann folgendermaßen gegeben werden:

$$\begin{array}{rcl} \{\mathbf{true}\} & \mathrm{New}(s) & \{s = \varepsilon\} \\ \{s = t \wedge m = n\} & \mathrm{Push}(s,m) & \{s = \mathrm{cons}(t,n)\} \\ \{s = \mathrm{cons}(t,n)\} & \mathrm{Pop}(s) & \{s = t\} \\ \{s = \mathrm{cons}(t,n)\} & \mathrm{Top}(s,m) & \{m = n\} \end{array}$$

Hier bezeichnet ε die leere Sequenz, $\mathrm{cons}(t,n)$ das Anhängen von n an t. Wie vorher ist s eine Programmvariable, ebenso m; dagegen sind t, n logische Variablen, die im Programm nicht vorkommen und deshalb durch Ausführung des Programms nicht verändert werden können. Neuere Spezifikationstechniken wie RAISE und auch Larch verwenden anstelle der konkreten Sequenzenstruktur axiomatisch definierte Datentypen (wie die obige Spezifikation von Kellern) und integrieren so den modellorientierten Ansatz mit dem axiomatischen. ◀

Der Einsatz formaler Methoden hängt essentiell vom Vorhandensein geeigneter Unterstützungswerkzeuge ab. Ziel ist es, automatische Beweissysteme bereitzustellen. Diese sind aber nur erfolgreich bei sehr kleinen Programmen bzw. bei eingeschränkten Formalismen wie endlichen Automaten. Andererseits ist das manuelle Beweisen fehleranfällig und komplex.

Daher konzentriert sich die Entwicklung von Werkzeugen auf halbautomatische Beweissysteme und Beweischecker.

Das Ziel ist heute die Integration formaler Methoden mit informellen Methoden. Formale Methoden sollen mit ihren Analyse- und Validierungstechniken vor allem helfen, Fehler in den frühen Phasen der Software-Entwicklung zu vermeiden.

Was können formale Methoden? Im folgenden sind einige Vorurteile gegenüber formalen Methoden aufgelistet.

1. Formale Methoden machen Testen überflüssig.

   Falsch: Der Übergang „reale Welt – System“ bzw. „informelle Beschreibung – formale Beschreibung“ kann nicht bewiesen, sondern nur getestet werden; Akzeptanztest für Benutzer; im Entwicklungsprozess Prüfung auf Fehler.

2. Formale Methoden eliminieren Notwendigkeit natürlicher Sprachen.

   Falsch: Die Anforderungsanalyse startet immer in natürlicher Sprache, erweitert um Fachausdrücke der Anwendung. Die natürliche Sprache ist Kommentar für Spezifikationen (und erste Erklärung, zur Klärung von Details verwendet man die formale Spezifikation).

System	Entwickler	Bemerkungen
LCF	R. Milner	Logic for Computable Functions
PVS	John Rushby, N. Shankar (SRI)	Interaktiver Beweiser für Logik höherer Ordnung, seit 1980
Isabelle	L. Paulson, T. Nipkow	Weiterentwicklung von LCF Cambridge, seit 1985
Larch	DEC Palo Alto / MIT Guttag, Horning	seit 1989
NUPRL	Constable, Cornell U.	Weiterentwicklung von LCF, seit 1986
Modelchecker	verschiedene Systeme zum Beweis von Eigenschaften endlicher Automaten (auch bei Echtzeit)	ca. seit 1990

Tabelle 2: Beispiele für beweisunterstützende Systeme

3. Ein Doktortitel ist notwendig, um formale Methoden zu verstehen.

   Falsch: Es ist notwendig, den Status Quo zu kritisieren und Ideale voranzustellen, nach dem Motto: „Ideale von heute sind die Praxis von morgen“ (z.B. formale Sprachen $\rightsquigarrow$ Compiler, endliche Automaten + Temporale Logik $\rightsquigarrow$ Model Checker). Formale Spezifikation ist eine formale Notation wie jede Programmiersprache.

1.4 Zusammenfassung

- Systementwicklung kann in Phasen eingeteilt werden, beginnend mit der Anforderungsbeschreibung und endend mit der Wartung. In allen Phasen benötigt man Validierung und Verifikation zur Überprüfung der Resultate.
- Mangel an Präzision, Unvollständigkeit und Zweideutigkeit sind häufige Fehlerquellen, die durch formale (möglichst automatische) Analysen besser entdeckt werden können.
- Grundlegende formale Techniken sind
  - axiomatische Spezifikationen zur Beschreibung von Daten und des funktionalen Verhaltens,
  - modell-orientierte Spezifikationen zur Beschreibung von zustandsbasiertem Verhalten,
  - temporal-logische Spezifikationen zur Beschreibung von dynamischem und reaktivem Verhalten.

Literaturhinweise

E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner (Hrsg.): *Algebraic Foundation of Systems Specification*. Berlin, Springer, 1999.

CASL: *Common language for formal specification of functional requirements and modular software design*, http://www.brics.dk/Projects/CoFI/, 2000.

J. Jacky: *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

L. Lamport: *TLA—The Temporal Logic of Actions*, http://www.research.digital.com/SRC/personal/lamport/tla/tla.html.

J. Loeckx, H. Ehrich, M. Wolf : *Specification of Abstract Data Types*. Wiley & Teubner, 1996.

Z. Manna, A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems—Specification*. Springer-Verlag, New York, 1992.

C. Morgan: *Programming from Specifications*. Prentice Hall, 3. Auflage, 1998.

H. Partsch: *Specification and Transformation of Programs*. Springer-Verlag, 1990.

M. Wirsing: *Algebraic Specification*. In: J. van Leeuwen (Hrsg.): Handbook of Theoretical Computer Science, North-Holland, Amsterdam, 1990, 675-788.

The Z Notation. http://archive.comlab.ox.ac.uk/z.html

Teil II

Datenorientierte Systementwicklung

Historisch:

1975 Guttag, Einsatz von Gleichungsspezifikationen in der Programmierung;

ADJ (Goguen, Thatcher, Wagner, Wright), erste formale Fundierung, sogenannte initiale Semantik;

1976 Gimona, Giarratana, Montanari, Wand: terminale Semantik;

1978 CIP-Gruppe (Bauer, Broy, Dosch, Partsch, Pepper, Wirsing): lose Semantik aller termerzeugten Modelle

2 Signaturen zur Schnittstellenbeschreibung

Ziele

- (die Syntax von) Schnittstellen durch Signaturen beschreiben lernen
- die Konstruktion der Terme einer Signatur verstehen

Zur Angabe der Schnittstelle eines Systems benötigt man die Angabe der

- Namen der nach außen sichtbaren Datentypen („Sorten“, „Typen“)
- Namen und Typ der nach außen sichtbaren Funktionen („Funktionssymbole“)
- Namen und Typ der nach außen sichtbaren Relationen („Prädikatensymbole“)

Man nennt dies eine Signatur — vgl. auch Schnittstellendefinitionen in Programmiersprachen wie Java oder Modula-2.

Definition 1 *Eine (algebraische)* Signatur Σ *ist ein Tripel* (S, F, P) *mit*

S : *eine Menge von Sorten, d.h Namen für Mengen,*

F : *eine* $S^* \times S$*-sortierte Familie von Funktionssymbolen, wobei*

- $\langle s_1, \ldots s_n \rangle \in S^*$ *den Definitionsbereich,*
- $s \in S$ *den Wertebereich der Funktionen aus* $F_{\langle \langle s_1, \ldots s_n \rangle, s \rangle}$ *bezeichnen, und*

P : *eine* S^**-sortierte Familie von Prädikatensymbolen.*

Bemerkung:

1. Für $f_{\langle\langle s_1,\ldots,s_n\rangle,s\rangle}$ schreiben wir

   $$f : s_1 \times \ldots \times s_n \to s$$

   und bezeichnen damit eine totale Funktion.

   In CASL hat man zusätzlich eine Notation für partielle Funktionen:

   $$f : s_1 \times \ldots s_n \to ? \; s$$

   bezeichnet eine (möglicherweise) partielle Funktion.

2. Wenn sich die Parametertypen zweier Funktions- oder Prädikatensymbole an mindestens einer Stelle unterscheiden, ist nach obiger Definition Overloading möglich. Overloading bedeutet, dass der gleiche Name für unterschiedliche Funktionen verwendet wird, z.B.

   $$+ : \text{Nat} \times \text{Nat} \to \text{Nat} \in F_{\langle\langle Nat,Nat\rangle,Nat\rangle}$$
   $$+ : \text{Vector} \times \text{Vector} \to \text{Vector} \in F_{\langle\langle Vector,Vector\rangle,Vector\rangle}$$

   Im Folgenden schränken wir uns für die theoretischen Überlegungen auf totale Funktionen ein, für die praktischen Beispiele lassen wir auch partielle Funktionen zu.

Beispiel 7: Boolesche Werte

sig BOOL0 =
sorts Bool
ops true : Bool;
false : Bool;
not _ : Bool → Bool;
_ and _ , _ or _ , _ implies _ : Bool × Bool → Bool
end

beschreibt die Signatur mit den folgenden Komponenten:

$$S = \{\text{Bool}\}$$
$$F_{\langle \varepsilon,\text{Bool}\rangle} = \{\text{true}, \text{false}\}$$
$$F_{\langle\langle \text{Bool}\rangle,\text{Bool}\rangle} = \{\text{not}\}$$
$$F_{\langle\langle \text{Bool,Bool}\rangle,\text{Bool}\rangle} = \{\text{and}, \text{or}, \text{implies}\}$$

◀

Beispiel 8: Natürliche Zahlen

sig NAT0 =
sorts Nat
ops zero: Nat;
succ _ : Nat → Nat
end

◀

Beispiel 9: Mengen

sig SET0 =
sorts Set, Elem
ops empty : Set; %%{äquivalent zu empty: → Set}%
{_} : Elem → Set
_ ∪ _ : Set × Set → Set
preds _ ∈ _ : Elem × Set
end

◄

Terme

Aus den Elementen einer Signatur Σ können (korrekt typisierte) Ausdrücke gebildet werden, sogenannte Σ-Terme. Solche Terme werden aus freien Variablen und den Funktionssymbolen von Σ aufgebaut. Genauer:

Definition 2 *(induktive Definition von Termen)*

1. *Seien $\Sigma = (S,F,P)$ und die Familie $X = (X_s)_{s\in S}$ von Identifikatoren gegeben. Dann ist $T(\Sigma,X)_s$ (für alle $s \in S$) definiert als die kleinste Menge, welche die folgenden Eigenschaften erfüllt:*
   - *(i) Für alle $x \in X_s$ ist $x \in T(\Sigma,X)_s$, d.h. jede Variable ist ein Σ-Term.*
   - *(ii) Ist $f \in F_{\langle\varepsilon,s\rangle}$, so ist $f \in T(\Sigma,X)_s$, d.h. f ist Konstante und somit ist jede Kostante ein Σ-Term.*
   - *(iii) Ist $f \in F_{\langle\langle s_1,\ldots,s_n\rangle,s\rangle}$ und $t_i \in T(\Sigma,X)_{s_i} (i = 1,\ldots,n)$, so ist $f(t_1,\ldots,t_n) \in T(\Sigma,X)_s$*
2. *Σ-Terme ohne Elemente von X heißen* Grundterme, *d.h. jedes $t \in T(\Sigma,\emptyset)_s$ ist Grundterm (ohne Variablen).*
3. *Eine Signatur heißt* bewohnt, *wenn für jedes $s \in S$ ein Grundterm $t \in T(\Sigma,\emptyset)_s$ existiert, d.h. wenn $T(\Sigma,\emptyset)_s$ verschieden von $\emptyset$ ist für alle $s \in S$.*
4. *Ein Term $t \in T(\Sigma,X)_s$ heißt* von der Sorte s.
5. $T(\Sigma,X) =_{\text{def}} T(\Sigma,X)_s \in S$

Beispiel 10: bewohnte Signaturen

1. NAT0 ist bewohnt, denn *zero* $\in T(\text{NAT0},\emptyset)_{\text{Nat}}$.
2. SET0 (mit Elem) ist nicht bewohnt, da $T(SET0,\emptyset)_{Elem} = \emptyset$.
3. Die Signatur

   sig SETNAT0 =
   sorts Nat, Set
   ops zero : Nat
   empty : Set
   ...
   end

   ist dagegen bewohnt.

◄

Beispiel 11: Terme

Sei x eine Variable der Sorte Nat, s eine Variable der Sorte Set. Dann sind

$\text{succ}(\text{succ}(\text{zero})), \text{succ}(x)$	Terme der Sorte Nat
$\text{empty}, \{\text{succ}(x)\}, \{\text{succ}(\text{succ}(\text{zero}))\}, s \cup \text{empty}$	Terme der Sorte Set
	(wobei Elem = Nat gewählt sei). ◄

Formal bezeichnet man mit $T(\Sigma, X)$ die Menge aller Terme über der Signatur Σ mit Variablen höchstens aus X.

Zusammenfassung

- Eine Signatur dient zur syntaktischen Beschreibung von Schnittstellen.
- Durch die Signatur (und die Angabe einer Familie von Variablen) wird die Menge der syntaktisch korrekten Terme bestimmt.

3 Funktionale Modelle: Schnittstelleninterpretation durch Strukturen

Ziele

- den Zusammenhang zwischen Signatur, Term und mathematischer Struktur verstehen lernen
- Formeln und ihre Interpretation in Strukturen verstehen
- ähnliche Algebren durch Σ-Homomorphismen in Beziehung setzen
- abstrakte Datentypen, initiale und erreichbare Algebren kennenlernen.

Signaturen sind syntaktische Schnittstellenbeschreibungen. Realisiert werden sie in einem funktionalen Ansatz durch Strukturen bzw. Algebren, die jedem Symbol aus einer Signatur eine typkorrekte Bedeutung zuordnen. Im Folgenden untersuchen wir auch Beziehungen zwischen Algebren und daraus sich ergebende ausgezeichnete Modelle, die in der Informatik an vielen Stellen eine Rolle spielen.

3.1 Algebren und Strukturen

Eine Σ-Algebra besitzt zu

- jeder Sorte eine Trägermenge,
- jedem Funktionssymbol eine Funktion.

Enthält die Signatur Σ auch Prädikatensymbole, so spricht man von Σ-Strukturen.

Definition 3 (Σ-Algebra) *Sei $\Sigma = (S, F, P)$ eine Signatur.*

1. *Eine* Σ-Algebra *A besteht aus*
   (a) *einer Familie $(A_s)_{s \in S}$ von nichtleeren Trägermengen und*
   (b) *(totalen) Funktionen $f^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$, für alle $f \in F_{\langle\langle s_1,\ldots,s_n\rangle,s\rangle}$.*
2. *Eine Σ-Struktur besitzt zusätzlich Relationen $p^A \subseteq A_{s_1} \times \ldots \times A_{s_n}$, für alle $p \in P_{\langle s_1,\ldots,s_n\rangle}$*
3. *Die Klasse aller Σ-Algebren wird mit* $\text{Alg}(\Sigma)$ *bezeichnet, die Klasse aller Σ-Strukturen mit* $\text{Struct}(\Sigma)$.

Beispiel 12: sig(BOOL0)-Algebren ohne „and“, „or“, „implies“

1. Das Standardmodell $\mathcal{B}$ der Warheitswerte

$$\begin{aligned}&\mathcal{B}_{Bool} = \{T, F\},\\ &true^{\mathcal{B}} = T, \qquad false^{\mathcal{B}} = F,\\ &not^{\mathcal{B}}(T) = F, \qquad not^{\mathcal{B}}(F) = T.\end{aligned}$$

2. Die Struktur *NB* über den natürlichen Zahlen mit

$$\begin{aligned}&NB_{Bool} = \mathbb{N},\\ &true^{NB} = 1, \qquad false^{NB} = 0,\\ &not^{NB}(2i) = 2i+1, \quad not^{NB}(2i+1) = 2i\end{aligned}$$

3. Die Struktur *ZB* über den ganzen Zahlen

$$\begin{aligned}&ZB_{Bool} = \mathbb{Z},\\ &true^{ZB} = 1, \qquad false^{ZB} = 0,\\ &not^{ZB}(1) = 0, \quad not^{ZB}(0) = 1,\\ &not^{ZB}(i) = i, \quad \text{für } i \text{ verschieden von } 0, 1.\end{aligned}$$

4. Die Struktur *ZB*1 über den ganzen Zahlen mit

$$\begin{aligned}&ZB1_{Bool} = \mathbb{Z},\\ &true^{ZB1} = 1, \qquad false^{ZB1} = 0,\\ &not^{ZB}(1) = 0, \qquad not^{ZB}(0) = 1,\\ &not^{ZB1}(i) = i+1, \quad \text{für } i \text{ verschieden von } 0, 1.\end{aligned}$$

5. Die triviale Struktur *UB* mit

$$\begin{aligned}&UB_{Bool} = 1,\\ &true^{UB} = false^{UB} = 1,\\ &not^{UB}(1) = 1,\end{aligned}$$

◀

Beispiel 13: Algebren für SET0 (Mengensignatur)

1. Endliche Mengen über $\mathbb{Z}$: $\mathcal{P}^{\text{fin}}(\mathbb{Z})$

$$\begin{aligned}&\text{Elem}^{\mathcal{P}^{\text{fin}}(\mathbb{Z})} =_{\text{def}} \mathbb{Z}\\ &\text{Set}^{\mathcal{P}^{\text{fin}}(\mathbb{Z})} =_{\text{def}} \{M \mid M \subseteq \mathbb{Z}, M \text{ endlich}\} \qquad \text{empty}^{\mathcal{P}^{\text{fin}}(\mathbb{Z})} =_{\text{def}} \emptyset\\ &\{z\}^{\mathcal{P}^{\text{fin}}(\mathbb{Z})} =_{\text{def}} \{z\} \qquad M_1 \cup^{\mathcal{P}^{\text{fin}}(\mathbb{Z})} M_2 =_{\text{def}} M_1 \cup M_2\end{aligned}$$

2. Endliche oder unendliche Mengen von ganzen Zahlen: $\mathcal{P}(\mathbb{Z})$

$$\begin{aligned}&\text{Elem}^{\mathcal{P}(\mathbb{Z})} =_{\text{def}} \mathbb{Z}\\ &\text{Set}^{\mathcal{P}(\mathbb{Z})} =_{\text{def}} \{M \mid M \subseteq \mathbb{Z}\}\\ &\text{Operationen wie vorher}\end{aligned}$$

3. AVL-Bäume über natürlichen Zahlen

$\text{Elem}^{AVL} =_{\text{def}} \mathbb{N}$
$\text{Set}^{AVL} =_{\text{def}}$ „alle AVL-Bäume“ $\{z\}^{AVL} =_{\text{def}}$ „einelementige AVL-Bäume“
$\text{empty}^{AVL} =_{\text{def}}$ „leerer AVL-Baum“ $AV_1 \cup^{AVL} AV_2 =_{\text{def}}$ „Vereinigung von AVL-Bäumen“

4. Die einelementige Struktur U für Set

$\text{Elem}^{U} =_{\text{def}} \mathbb{Z}$
$\text{Set}^{U} =_{\text{def}} \{\bullet\}$ $\text{empty}^{U} =_{\text{def}} \bullet$
$\{z\}^{U} =_{\text{def}} \bullet$ $M_1 \cup^{U} M_2 =_{\text{def}} \bullet$

5. Zahlen für Set

$\text{Elem}^{ZZ} =_{\text{def}} \mathbb{Z}$
$\text{Set}^{ZZ} =_{\text{def}} \mathbb{Z}$ $\text{empty}^{ZZ} =_{\text{def}} 0$
$\{z\}^{ZZ} =_{\text{def}} z$ $z_1 \cup^{ZZ} z_2 =_{\text{def}} z_1 + z_2$

◄

Termalgebra

Sei $\Sigma = (S,F)$ eine bewohnte Signatur.

Es lässt sich auf einfache Weise eine Struktur definieren, die sogenannte *Termalgebra*, die ebenso wie die Menge der Terme mit $T(\Sigma,X)$ bezeichnet wird. Wir schreiben im folgenden kurz T für $T(\Sigma,X)$.

Die Trägermengen von T sind die Mengen $T(\Sigma,X)_s$ für $s \in S$. Die Interpretation der *Funktionssymbole* ist definiert durch

$f^T(a_1,\ldots,a_n) =_{\text{def}} f(a_1,\ldots,a_n)$ für $a_i \in T_{s_i}$

Bemerkung:

1. $T(\Sigma) =_{\text{def}} T(\Sigma,\emptyset)$ heißt *Grundtermalgebra*.

2. Enthält die Signatur auch Prädikatensymbole neben der Gleichheit, so lassen sich die Interpretationen dieser Symbole frei wählen. Solche Strukturen werden „Herbrand-Strukturen“ genannt und zum Beispiel zur Definition der Semantik von Logikprogrammen benutzt.

Beispiel 14: Grundtermalgebra $T = T(\text{BOOL0})$ mit

$T_{\text{Bool}} = T(\text{BOOL0})_{Bool}$
$\text{true}^T = \text{true}$ $\text{false}^T = \text{false}$
$\text{not}^T(t) = \text{not}(t)$ usw.

◄

Interpretation von Termen

Definition 4 *Gegeben Signatur $\Sigma = (S,F,P)$, Variablenmenge X, Σ-Algebra A.*

1. *Eine* Belegung *von X nach A ist eine Familie von Abbildungen $(\nu_s : X_s \to A_s)_{s\in S}$, geschrieben $\nu : X \to A$.*

2. *Die* Interpretation $I_\nu : T(\Sigma,X) \to A$ *von Termen in A bzgl. ν ist die folgendermaßen definierte Familie von Abbildungen:*

   (a) $(I_{\nu_s})(x) = \nu_s(x)$ *für $s \in S$ und $x \in X_s$,*

   (b) $I_{\nu_s}(f(t_1,\ldots,t_n)) = f^A(I_{\nu_{s_1}}(t_1),\ldots,I_{\nu_{s_n}}(t_n))$
   für alle $f \in F_{\langle\langle s_1,\ldots,s_n\rangle,s\rangle}$, $t_1 \in T(\Sigma,X)_{s_1}$, …, $t_n \in T(\Sigma,X)_{s_n}$.

   Anstelle von I_{ν_s} schreiben wir meist I_ν.

   Für Grundterme $t \in T(\Sigma,\emptyset)$ ist die Definition unabhängig von der Belegung ν, wir schreiben daher t^A anstelle von $I_\nu(t)$.

Beispiel 15:

1. In der Struktur *NB* aus Beispiel 12 gilt für $\nu(x) = 2i$ (mit beliebigem $i \in \mathbb{N}$):

$$\begin{aligned} I_\nu(\text{not}(\text{not}(x))) &= \text{not}^{NB}(\text{not}^{NB}(\nu(x))) = \text{not}^{NB}(\text{not}^{NB}(2i)) \\ &= \text{not}^{NB}(2i+1) = 2i = \nu(x) = I_\nu(x) \end{aligned}$$

   Analog gilt für $\nu(x) = 2i+1$:

$$I_\nu(\text{not}(\text{not}(x))) = 2i+1 = \nu(x) = I_\nu(x)$$

   d.h. insgesamt gilt für alle Belegungen $\nu : \{x\} \to \text{Bool}_{NB}$:

$$I_\nu(\text{not}(\text{not}(x))) = \nu(x)$$

2. Betrachte die Signatur SET0 mit Variablenmenge $X_{Nat} = \{x\}$, $X_{Set} = \{s\}$ sowie die Algebra $\mathcal{P}^{\text{fin}}(\mathbb{Z})$ (vgl. Beispiel 13).

   Sei $\nu : X \to \mathcal{P}^{\text{fin}}(\mathbb{Z})$ die Belegung mit

$$\nu_{\text{Nat}}(x) = 17, \quad \nu_{\text{Set}}(s) = \emptyset$$

   Dann gilt:

$$\begin{aligned} I_\nu(\text{empty} \cup \{x\}) &= I_\nu(\text{empty}) \cup^{\mathcal{P}^{\text{fin}}(\mathbb{Z})} I_\nu(\{x\}) \\ &= \text{empty}^{\mathcal{P}^{\text{fin}}(\mathbb{Z})} \cup^{\mathcal{P}^{\text{fin}}(\mathbb{Z})} \{I_\nu(x)\}^{\mathcal{P}^{\text{fin}}(\mathbb{Z})} = \emptyset \cup \{17\} = \{17\} \end{aligned}$$

◀

3.2 Σ-Formeln

Σ - Formeln sind Formeln der mehrsortigen Prädikatenlogik 1. Stufe mit Funktionssymbolen und Gleichheit, sie sind aufgebaut aus atomaren Formeln, Booleschen Verknüpfungen und Quantoren ($\forall, \exists$).

Im Folgenden werden die für die Vorlesung wichtigen Definitionen, Beweiskalküle und Umformungsregeln kurz vorgestellt.

3.2.1 Syntax von Formeln

Definition 5 *Sei $\Sigma = (S, F, P)$ eine Signatur und X eine S-sortierte Famile von Variablen.*

1. *Eine* atomare Σ*-Formel hat die Gestalt*

   $$p(t_1, \ldots, t_n) \quad \text{für } p \in P_{\langle s_1,\ldots,s_n\rangle}, t_1 \in T(\Sigma, X)_{s_1}, \ldots, t_n \in T(\Sigma, X)_{s_n}$$

   Außerdem ist für jede Sorte $s \in S$ und alle Terme $t_1, t_2 \in T(\Sigma, X)_s$ auch $t_1 =_s t_2$ eine atomare Formel. Der (Sorten-)Index wird in der Regel weggelassen.

2. *Die Menge der* Σ-Formeln $WFF(\Sigma)$ *„well-formed formula“ ist die kleinste Menge, die folgende Eigenschaften erfüllt (induktive Definition):*

   (a) jede atomare Σ-Formel ist in $WFF(\Sigma)$,

   (b) sind G_1, G_2 in $WFF(\Sigma)$, so auch $(\neg\, G_1)$ und $(G_1 \wedge G_2)$,

   (c) ist G in $WFF(\Sigma)$, $x \in X_s$, so ist $(\forall x : s.\ G)$ in $WFF(\Sigma)$.

3. *Weitere Operatoren werden als Abkürzungen definiert:*

   $$\begin{array}{lll} (G_1 \vee G_2) & =_{\text{def}} & \neg\,((\neg\, G_1) \wedge (\neg\, G_2)) \\ (G_1 \Rightarrow G_2) & =_{\text{def}} & ((\neg\, G_1) \vee G_2) \\ (G_1 \equiv G_2) & =_{\text{def}} & ((G_1 \Rightarrow G_2) \wedge (G_2 \Rightarrow G_1) \\ (\exists x : s.\ G) & =_{\text{def}} & (\neg\,(\forall x : s.\ (\neg\, G))) \end{array}$$

   Außerdem gelten die üblichen Klammerersparnisregeln.

   Mit $FV(G)$ bezeichnen wir die Menge der freien Variablen von G. Eine Formel G heißt geschlossen, *wenn $FV(G) = \emptyset$ ist.*

4. *Eine* aussagenlogische *Formel ist eine Σ-Formel ohne Quantoren.*

5. *Eine* Hornformel *hat die Gestalt*

   $$\forall x_1 : s_1, \ldots, \forall x_m : s_m.\ G_1 \wedge \ldots \wedge G_n \Rightarrow G$$

   für atomare Formeln G_i und G. Wenn alle atomaren Formeln hier Gleichungen sind, sprechen wir auch von bedingten Gleichungen.

Beispiel 16:

1. $\forall x, y, z : \text{Elem}.\ (x \circ y) \circ z = x \circ (y \circ z)$

   ist eine allquantifizierte Gleichung zur Beschreibung der Assoziativität.

2. $\forall x, y : \text{Nat}.\ \text{succ}(x) = \text{succ}(y) \Rightarrow x = y$

   ist eine bedingte Gleichung zur Beschreibung der Injektivität von succ.

3. $\forall x : \text{Bool}.\ x = \text{true} \vee x = \text{false}$

   sagt aus, dass die Sorte Bool höchstens zwei Elemente besitzt.

4. $\forall x : \text{Bool}.\ \text{not}(\text{not}(x)) = x$

   drückt die Idempotenz von not aus. ◄

3.2.2 Semantik von Formeln

Die *Gültigkeit* von Σ-Formeln in einer Σ-Struktur A ist folgendermaßen definiert:

Definition 6 *Sei $X = (X_s)_{s\in S}$ und A eine Σ-Struktur.*
Dann ist die Gültigkeit *einer Formel G in der Struktur A bzgl. Belegung $\nu : X \to A$, in Zeichen $A,\nu \models G$, wie folgt induktiv definiert:*

1. $A,\nu \models p(t_1,\ldots,t_n)$ *gdw.* $(I_\nu(t_1),\ldots,I_\nu(t_n)) \in p^A$
   $A,\nu \models t_1 = t_2$ *gdw.* $I_\nu(t_1) = I_\nu(t_2)$
   $A,\nu \models \neg\, G$ *gdw.* $A,\nu \not\models G$
   $A,\nu \models G_1 \wedge G_2$ *gdw.* $A,\nu \models G_1$ *und* $A,\nu \models G_2$
   $A,\nu \models \forall x : s.\ G$ *gdw.* $A,\nu_x \models G$ *für alle Belegungen* $\nu_x : X \to A$ *mit* $\nu_x(z) = \nu(z)$ *für z verschieden von x*
2. *Die Formel G* gilt in der Struktur A *($A \models G$) gdw. $A,\nu \models G$ für alle Belegungen $\nu : X \to A$.*
3. *Eine Menge E von Formel* gilt in der Struktur A *($A \models E$) gdw. $A \models G$ für alle $G \in E$.*
4. *G ist* allgemeingültig *($\models G$) gdw. $A \models G$ für alle Σ-Strukturen A.*
5. *G* folgt prädikatenlogisch aus *E ($E \models G$) gdw. $A \models G$ für alle Strukturen A mit $A \models E$.*

Beispiel 17: $N = \langle \mathbb{N}, 0, \_ + 1\rangle$ bezeichne das Standardmodell der natürlichen Zahlen.

1. Sei $\nu(x) = 12$. Es gilt

$$\begin{array}{lll} \mathbb{N},\nu \models \text{zero} = \text{succ}(x) & \text{gdw.} & I_\nu(\text{zero}) = I_\nu(\text{succ(x)}) \\ & \text{gdw.} & \text{zero}^{\mathbb{N}} = \text{succ}^{\mathbb{N}}(\nu(x)) \\ & \text{gdw.} & 0 = 12+1 \\ & \text{gdw.} & 0 = 13 \end{array}$$

2. Sei $\nu(y) = 0$. Wir zeigen: $\mathbb{N},\nu \models \forall x : \text{Nat.}\ \text{succ}(x) > y$.
   Es reicht zu zeigen: $\mathbb{N},\nu_x \models \text{succ}(x) > y$ gilt für jede Belegung ν_x mit $\nu_x(y) = 0$ und $\nu_x(x) \in \mathbb{N}$.
   Sei $\nu_x(x) = i \in \mathbb{N}$, dann ist

$$I_{\nu_x}(\text{succ}(x)) = \text{succ}^{\mathbb{N}}(\nu_x(x)) = i+1 \qquad \text{und} \qquad I_{\nu_x}(y) = \nu_x(y) = 0$$

   und es gilt $(i+1,0) \in >^{\mathbb{N}}$. ◀

Beispiel 18: In der Struktur NB aus Beispiel 12 gilt $NB \models \forall x : \text{Bool.}\ \text{not}(\text{not}(x)) = x$
da für alle $\nu : \{x\} \to NB_{\text{Bool}}$ gilt (vgl. Bsp. 15): $I_\nu(\text{not}(\text{not}(x))) = \nu(x)$. ◀

3.2.3 Wahrheitstabellen

Tautologien kann man mit Hilfe von Wahrheitstabellen nachweisen. In einer Wahrheitstabelle werden alle möglichen Belegungen von atomaren Teilformeln einer aussagenlogischen Formel mit Wahrheitswerten betrachtet. Z.B. zeigt Tabelle 3 die Wahrheitstabelle für die Negation

Tabelle 4 fasst die Wahrheitstabellen von Konjunktion, Disjunktion und Implikation zusammen

Eine aussagenlogische Formel ist eine Tautologie, wenn ihr Wert für alle Belegungen in der Wahrheitstabelle T ergibt.

G	$\neg\, G$
T	F
F	T

Tabelle 3: Wahrheitstabelle für die Negation.

G	H	$G \wedge H$	$G \vee H$	$G \Rightarrow H$
T	T	T	T	T
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Tabelle 4: Wahrheitstabellen für Konjunktion, Disjunktion und Implikation.

Beispiel 19: Wahrheitstabellen zum Nachweis von Tautologien

Die folgende Wahrheitstabelle zeigt, dass $G \vee \neg\, G$ und $G \Rightarrow (H \Rightarrow G)$ Tautologien sind.

G	$\neg\, G$	$G \vee \neg\, G$
T	F	T
F	T	T

G	H	$H \Rightarrow G$	$G \Rightarrow (H \Rightarrow G)$
T	T	T	T
T	F	T	T
F	T	F	T
F	F	T	T

Dagegen ist $H \Rightarrow (H \Rightarrow G)$ keine Tautologie, wie die dritte Zeile der folgenden Wahrheitstabelle zeigt.

G	H	$H \Rightarrow G$	$H \Rightarrow (H \Rightarrow G)$
T	T	T	T
T	F	T	T
F	T	F	F
F	F	T	T

◀

3.2.4 Aussagenlogische und prädikatenlogische Umformungen

Mit aussagenlogischen Formeln kann man ähnlich wie in der Arithmetik einfache algebraische Umformungen durchführen. Die wichtigsten Gesetze sind:

Assoziativität und Kommutativität von Konjunktion und Disjunktion

$$G \wedge (H \wedge K) \equiv (G \wedge H) \wedge K \qquad G \vee (H \vee K) \equiv (G \vee H) \vee K$$
$$G \wedge H \equiv H \wedge G \qquad G \vee H \equiv H \vee G$$

Distributivität von Konjunktion und Disjunktion

$$G \wedge (H \vee K) \equiv (G \wedge H) \vee (G \wedge K) \qquad G \vee (H \wedge K) \equiv (G \vee H) \wedge (G \vee K)$$

Absorption

$$G \wedge (G \vee H) \equiv G \qquad G \vee (G \wedge H) \equiv G$$

Involution

$$\neg \neg G \equiv G$$

Regeln von De Morgan

$$\neg (G \wedge H) \equiv (\neg G \vee \neg H) \qquad \neg (G \vee H) \equiv (\neg G \wedge \neg H)$$

Neutralitätsgesetze

$$G \wedge (H \vee \neg H) \equiv G \qquad G \vee (H \wedge \neg H) \equiv G$$

Fanggesetze

$$G \vee (H \vee \neg H) \equiv (H \vee \neg H) \qquad G \wedge (H \wedge \neg H) \equiv (H \wedge \neg H)$$

Für Formeln mit Quantoren gibt es ähnliche Umformungsregeln, z.B.

$$\begin{aligned}(\forall x : s.\ G) \wedge (\forall x : s.\ H) &\equiv (\forall x : s.\ G \wedge H)\\ (\exists x : s.\ G) \vee (\exists x : s.\ H) &\equiv (\exists x : s.\ G \vee H)\end{aligned}$$

Vertauschung gleichartiger Quantoren

$$\begin{aligned}\forall x : s_1.\ \forall y : s_2.\ G &\equiv \forall y : s_2.\ \forall x : s_1.\ G\\ \exists x : s_1.\ \exists y : s_2.\ G &\equiv \exists y : s_2.\ \exists x : s_1.\ G\end{aligned}$$

3.2.5 Beweissysteme für die Prädikatenlogik

Um die Gültigkeit von prädikatenlogischen Formeln zeigen zu können, reichen Wahrheitstabellen nicht aus. Man verwendet Beweissysteme. Wir schreiben $\vdash G$, falls G eine beweisbare Formel ist und $E \vdash G$, falls G beweisbar ist unter Annahme der Formeln in E als zusätzlichen Axiomen.

Definition 7

1. *Ein Beweissystem $\vdash$ der Logik 1. Stufe heißt* korrekt, *wenn für jede Formel G gilt:*

   $$\vdash G \quad \Rightarrow \quad \models G$$

2. *Ein Beweissystem $\vdash$ der Logik 1. Stufe heißt* vollständig, *wenn für jede Formel G gilt:*

   $$\models G \quad \Rightarrow \quad \vdash G$$

Sei $\vdash$ ein korrektes und vollständiges Beweissystem der mehrsortigen Logik 1. Stufe. E sei eine Menge von geschlossenen Σ-Formeln 1. Stufe.

Dann gilt aufgrund der Korrektheit und Vollständigkeit für jede geschlossene Σ-Formel G:

$$E \vdash G \qquad \text{gdw.} \qquad E \models G$$

Hierfür gibt es einen Standardbeweis, siehe z. B. Gödel, 1930, oder Shoenfield bzw. Handbook for Mathematical Logic (Gödelscher Vollständigkeitssatz).

Im folgenden benötigen wir häufig den Begriff der Substitution:

Definition 8 *(Substitution) Sei $\Sigma = (S,F)$ eine Signatur und X eine S-sortierte Menge von Variablen. Eine* Substitution *ist eine endliche sortenerhaltende Abbildung $\sigma : X \to T(\Sigma,X)$, die jeder Variablen aus X einen Term aus $T(\Sigma,X)$ zuordnet. Endlich heißt hier, dass $\sigma(x) \neq x$ nur für endlich viele $x \in X$ gilt. (Sortenerhaltend heißt, dass für alle $s \in S$ und alle $x \in X_s$ auch $\sigma(x)$ von der Sorte s ist.)*

Bemerkung:

1. Wir schreiben $[t/x]$ für die Substitution σ mit $\sigma(x) = t$ und $\sigma(z) = z$ für $z \neq x$. Analog bezeichnet $[t_1/x_1, \ldots, t_n/x_n]$ die simultane Substitution von $t_1, \ldots, t_n$ für $x_1, \ldots, x_n$.

2. Jede Substitution wird folgendermaßen auf Σ-Terme und Σ-Formeln fortgesetzt:

$$\begin{array}{rcll} x\sigma & =_{\text{def}} & \sigma(x) & \\ f(t_1,\ldots,t_n)\sigma & =_{\text{def}} & f(t_1\sigma,\ldots,t_n\sigma) & \text{für Funktions- und Prädikatensymbole } f \\ (t_1 = t_2)\sigma & =_{\text{def}} & (t_1\sigma = t_2\sigma) & \\ (\neg\, G)\sigma & =_{\text{def}} & (\neg\, (G\sigma)) & \\ (G \wedge G')\sigma & =_{\text{def}} & G\sigma \wedge G'\sigma, & \\ (\forall x : s.\ G)\sigma & =_{\text{def}} & \forall y : s.\ G[y/x]\sigma & \text{wobei y eine neue Variable (mit } \sigma(y) = y) \text{ ist} \end{array}$$

Ein Beispiel für ein korrektes und vollständiges Beweissystem ist der *Hilbert-Kalkül* für mehrsortige Logik 1. Stufe: Sei $\Sigma = (S, F, P)$ eine bewohnte Signatur.

Axiome: (für jede Σ-Formel G)

1. $\vdash \neg\, G \vee G$ (Tertium non datur)
2. $\vdash G[t/x] \Rightarrow \exists x : s.\ G$ für $t \in T(\Sigma, X)_s, x \in X_s$

Regeln:

1. ($\vee$I) $\dfrac{G}{G' \vee G}$

2. (a) (Kontraktion) $\dfrac{G \vee G}{G}$

   (b) (Assoziativität) $\dfrac{G \vee (G' \vee G'')}{(G \vee G') \vee G''}$

3. (Modus ponens) $\dfrac{G \vee G' \quad \neg\, G \vee G''}{G' \vee G''}$

4. ($\exists$E) $\dfrac{G \Rightarrow G'}{(\exists x : s.\ G) \Rightarrow G'}$ falls nicht $x \in FV(G)$

Beispiel 20:

1. $\{G \vee G'\} \vdash G' \vee G$:

   (1) $G \vee G'$ (Annahme)
   (2) $\neg\, G \vee G$ (tertium non datur)
   (3) $G' \vee G$ (modus ponens)(1)(2)

2. $\{G, G \Rightarrow G'\} \vdash G'$:

(1)	G	(Annahme)
(2)	$G' \vee G$	($\vee$ I)(1)
(3)	$G \vee G'$	(aus 2 wie oben)
(4)	$\neg\, G \vee G'$	(Annahme, Def. von $\Rightarrow$)
(5)	$G' \vee G'$	(modus ponens)(3)(4)
(6)	G'	(Kontraktion)(5) ◄

3.2.6 Allgemeingültigkeit von Gleichungen: Der Kalkül von Birkhoff

Im folgenden schränken wir uns auf Gleichungen ein und untersuchen deren Allgemeingültigkeit. Ein Beispiel für eine allgemeingültige Formel ist die Reflexivität $t = t$. Andere allgemeingültige Gleichungen gibt es nicht:

Satz 9 *Eine Gleichung t = t' ist genau dann allgemeingültig, wenn t und t' syntaktisch gleich sind.*

Beweis. "$\Rightarrow$": Reflexivität von „=".

"$\Leftarrow$": Beweis durch Kontraposition: Seien t, t' *nicht* syntaktisch gleich (für $t, t' \in T(\Sigma, X)$).

Dann sind t und t' verschiedene Terme, d.h. es gilt *nicht* $T(\Sigma, X) \models t = t'$. Also ist $t = t'$ nicht allgemeingültig. Q.E.D.

Wir benötigen Axiome, um nichttriviale Gleichungen herleiten zu können. Deshalb betrachten wir im Falle von Gleichungen immer eine Menge von Axiomen.

Bemerkung: Für jede Σ-Struktur A gilt: $A \models t = t'$ gdw. $A \models \forall x : s.\ t = t'$, falls $x \in X_s$. Deshalb ist die Gültigkeit von Gleichungen und sogar jeder beliebigen Formel G 1. Stufe äquivalent zur Gültigkeit ihres „All-Abschlusses" $\forall x_1 : s_1 \ldots \forall x_n : s_n.\ G$, wobei $\{x_1, \ldots, x_n\}$ die Menge der freien Variablen von G ist.

Das folgende klassische Beweissystem stammt von Birkhoff (1931):

Sei E eine Menge von Gleichungen über Σ, X.

Definition 10 (Birkhoff´s reiner Gleichungskalkül)

$\vdash_B$ über Σ, X ist die kleinste zweistellige Relation von Gleichungen E (über Σ, X) und einer Gleichung $t_1 = t_2$ mit $t_1, t_2 \in T(\Sigma, X)$ mit

1. *$E \vdash_B t_1 = t_2$ für $t_1 = t_2 \in E$,*
2. *für alle Terme $t, t_1, t_2, t_3 \in T(\Sigma, X)_s$ und alle $x \in X_s$:*

Reflexivität	$E \vdash_B t_1 = t_1$
Symmetrie	*wenn* $E \vdash_B t_1 = t_2$, *dann* $E \vdash_B t_2 = t_1$
Transitivität	*wenn* $E \vdash_B t_1 = t_2$ *und* $E \vdash_B t_2 = t_3$, *dann* $E \vdash_B t_1 = t_3$
Verträglichkeit	*wenn* $E \vdash_B t_1 = t_2$, *dann* $E \vdash_B t[t_1/x] = t[t_2/x]$
Substitution	*wenn* $E \vdash_B t_1 = t_2$, *dann* $E \vdash_B t_1[t/x] = t_2[t/x]$

Beispiel 21: Sei $\Sigma = (\{s\}, \{\vee \in s \times s \to s;\ \wedge \in s \times s \to s\})$.

$$E = \{\ (\vee\text{-}\wedge\text{-Absorption}) \quad x \vee (x \wedge y) = x,$$
$$(\wedge\text{-}\vee\text{-Absorption}) \quad x \wedge (x \vee y) = x\ \}$$

Wir zeigen: $E \vdash_B x \vee x = x$.

(1)	$x \vee (x \wedge y) = x$	($\vee$-$\wedge$-Absorption)
(2)	$x \vee (x \wedge (x \vee y)) = x$	(Substitution)(1)
(3)	$x \wedge (x \vee y) = x$	($\wedge$-$\vee$-Absorption)
(4)	$x = x \wedge (x \vee y)$	(Symmetrie)(3)
(5)	$x \vee x = x \vee (x \wedge (x \vee y))$	(Kongruenz)
(6)	$x \vee x = x$	(Transitivität)(5)(2) ◀

Satz 11 *Der Birkhoff'sche Gleichungskalkül ist korrekt und vollständig, d.h. für Σ, X, E gilt für alle $t_1, t_2 \in T(\Sigma, X)_s$, $s \in S$:*

$$E \vdash_B t_1 = t_2 \quad \textit{gdw.} \quad \mathrm{Mod}(\langle \Sigma, E \rangle) \models t_1 = t_2 \quad \textit{d.h. für alle } A \in \mathrm{Mod}(\langle \Sigma, E \rangle) \textit{ gilt } A \models t_1 = t_2$$

Elegantere und effizientere Kalküle für die Ableitungen für Gleichungen werden in der Termersetzung behandelt.

3.3 Eigenschaften von Algebren: Homomorphismen, initiale und erreichbare Strukturen

Signaturen können durch ganz unterschiedliche Algebren interpretiert werden. Zur Spezifikation von Softwaresystemen will man aber häufig spezielle Algebren auszeichnen, die sich zur Darstellung von Programmen eignen. Zum Beispiel ist man besonders interessiert an Strukturen, die genau die angegebene Menge von Gleichungen erfüllen (sogenannte initiale Strukturen) oder an Strukturen, deren Trägermengen sich durch Grundterme darstellen lassen. Außerdem soll die Darstellung eines Softwaresystems für den Benutzer unabhängig von der gewählten Interpretation sein, d.h sie soll abstrakt sein. In diesem Abschnitt wird gezeigt, wie sich diese Eigenschaften mathematisch fassen lassen. Das wichtigste Strukturierungsmittel zum Vergleich von Algebren sind Homomorphismen.

Definition 12 *Seien $\Sigma = (S, F)$ Signatur und A, B Σ-Algebren*

1. *Ein* Σ-Homomorphismus $(\rho : A \to B)$ *ist eine Familie von Abbildungen*

   $$(\rho_s : A_s \to B_s)_{s \in S}$$

   mit folgenden Eigenschaften:

   Für alle Funktionssymbole $f \in F_{\langle s_1, \ldots, s_n \rangle, s \rangle}$, und alle $a_i \in A_{s_i}$, $i = 1, \ldots, n$:

   $$\rho_s(f^A(a_1, \ldots, a_n)) = f^B(\rho_{s_1}(a_1), \ldots, \rho_{s_n}(a_n))$$

   (vgl. Abbildung 2).

2. *Ein bijektiver Σ-Homomorphismus heißt* Σ-Isomorphismus. *Zwei Σ-Algebren A, B heißen* isomorph, *wenn es einen Σ-Isomorphismus von A nach B gibt.*

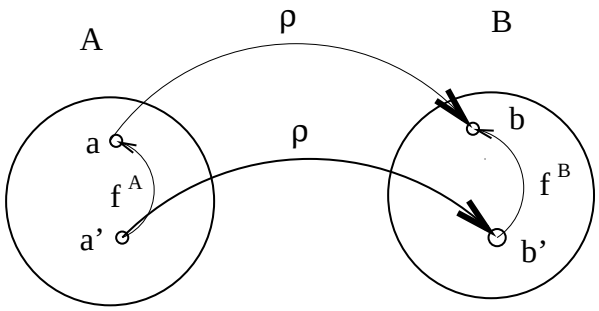


Abbildung 2: Homomorphiebedingung für einstellige Funktion f.

Bemerkung:

1. Die Gleichheit wird durch Σ-Homomorphismen erhalten, aber nicht die Ungleichheit, d.h. es gilt natürlich

   $$a_1 =^A a_2 \quad \Rightarrow \quad \rho_s(a_1) =^B \rho_s(a_2)$$

   Dagegen gilt i.a. *nicht*:

   $$a_1 \neq^A a_2 \quad \Rightarrow \quad \rho_s(a_1) \neq^B \rho_s(a_2)$$

2. Für Strukturen fordert man als zusätzliche Homomorphie-Eigenschaft die Verträglichkeit der Prädikatensymbole, d.h. für alle Prädikatensymbole $p \in P_{\langle s_1,\ldots,s_n\rangle}$ und $a_i \in A_{s_i}$, $i = 1,\ldots,n$:

   $$(a_1,\ldots,a_n) \in p^A \quad \Rightarrow \quad (\rho_{s_1}(a_1),\ldots,\rho_{s_n}(a_n)) \in p^B$$

Beispiel 22: Wir betrachten folgende NAT0-Algebren:

- $N = \langle \mathbb{N}, 0, \_ + 1\rangle$ das Standardmodell der natürlichen Zahlen
- $Z = \langle \mathbb{Z}, 0, \_ + 1\rangle$ das Standardmodell der ganzen Zahlen
- $N_2 = \langle \{0,1\}, 0, \_ + 1 (\mathrm{mod}\, 2)\rangle$
- $N_1 = \langle \{0\}, 0, id\rangle$.

Die Beziehungen zwischen den Modellen sind in Abbildung 3 dargestellt. Dabei sind:

- $in : N \to Z$ mit $in(x) = x$ „Einbettungshomomorphismus“
- $\rho_2 : N \to N_2$ bzw. $\rho_2^Z : Z \to N_2$ mit $\rho_2(x) = \rho_2^Z(x) = x \bmod 2$
- $\rho_1 : N_2 \to N_1$ mit $\rho_1(x) = 0$

Nachweis der Homomorphiebedingung für $\rho_2 : N \to N_2$:

$\rho_2(\mathrm{zero}^N) = 0 = \mathrm{zero}^{N2}$
$\rho_2(\mathrm{succ}^N(x)) = \rho_2(x+1) = (x+1) \bmod 2$
$\mathrm{succ}^{N2}(\rho_2(x)) = \mathrm{succ}^{N2}(x \bmod 2) = ((x \bmod 2)+1) \bmod 2 = (x+1) \bmod 2$ ◄

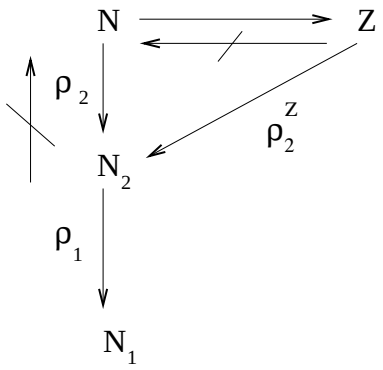


Abbildung 3: Homomorphismen zwischen NAT0-Algebren.

Aus der Homomorphiebedingung folgt, dass die Anwendung von Homomorphismen mit der Auswertung von Grundtermen verträglich ist.

Lemma 13 *Sei* $\rho : A \to B$ *ein* Σ*-Homomorphismus. Dann gilt für alle Grundterme* $t \in T(\Sigma)$*:*

$$\rho(t^A) = t^B$$

Beweis. durch strukturelle Induktion. Q.E.D.

Beispiel 23: Nicht-Existenz von Homomorphismen

1. Von N_2 nach N gibt es keinen Nat0-Homomorphismus ρ. Ansonsten müsste gemäß Lemma 13 gelten:

   $$\rho(\text{succ}(\text{succ}(\text{zero}))^{N_2}) = \text{succ}(\text{succ}(\text{zero}))^N = 2 \qquad \rho(\text{zero}^{N_2}) = \text{zero}^N = 0$$

   Andererseits gilt

   $$\text{succ}(\text{succ}(\text{zero}))^{N_2} = \text{zero}^{N_2}$$

   und daher müsste auch

   $$2 = \rho(\text{succ}(\text{succ}(\text{zero}))^{N_2}) = \rho(\text{zero}^{N_2}) = 0$$

   gelten. Es kann daher keinen Homomorphismus $\rho : N_2 \to N$ geben.

2. Es gibt keinen NAT0-Homomorphismus von Z nach N. Angenommen, $\rho : Z \to N$ ist NAT0-Homomorphismus. Wieder muss nach Lemma 13 gelten:

   $$\rho(\text{zero}^Z) = \text{zero}^N = 0$$

   Es sei $\rho(-1) = a \in \mathbb{N}$, dann folgt weiter

   $$0 = \rho(\text{succ}^Z(-1)) = \text{succ}^N(\rho(-1)) = \text{succ}^N(a) = a+1$$

   im Widerspruch zu $N \models \forall x : \text{Nat}.\ \neg\,(\text{zero} = \text{succ}(x))$. ◀

Durch Homomorphismen wird in manchen Teilklassen $K \subseteq \mathrm{Alg}(\Sigma)$ eine spezielle Klasse von Algebren ausgezeichnet:

Definition 14 *Eine Σ-Algebra I heißt* initial *in K, wenn gilt:*

1. $I \in K$
2. *für alle $B \in K$ gibt es genau einen Σ-Homomorphismus $\rho : I \to B$.*

Beispiel 24: Sei Σ eine bewohnte Signatur.

1. Die Grundtermalgebra $T(\Sigma)$ ist initial in $\mathrm{Alg}(\Sigma)$:
   Sei $A \in \mathrm{Alg}(\Sigma)$. Die Familie von Abbildungen $eval_s : T(\Sigma)_s \to A_s$ mit $eval(t) = t^A$ ist ein Σ-Homomorphismus. Denn es gilt $eval(f(t_1,\ldots,t_n)) = f^A(eval(t_1),\ldots,eval(t_n))$, und das ist genau die Homomorphismus-Eigenschaft.
2. Das Standardmodell $N = \langle \mathbb{N}, 0, \_ + 1 \rangle$ der natürlichen Zahlen ist initial in $\mathrm{Alg}(NAT0)$.

$$\begin{array}{lcccc} \mathbb{N} = & 0 & 1 & 2 & \ldots \\ & \updownarrow & \updownarrow & \updownarrow & \\ & \mathrm{zero} & \mathrm{succ}(\mathrm{zero}) & \mathrm{succ}(\mathrm{succ}(\mathrm{zero})) & \end{array}$$

3. Andere Interpretationen von $sig(NAT0)$ wie z.B. Z, N_2, N_1 sind *nicht* initial in $\mathrm{Alg}(NAT0)$. ◀

Beispiel 25: Sei LISTNATI folgende Signatur:

sig LISTNATI =
sorts List, Nat
ops nil : List
cons : Nat × List → List
zero : Nat
succ : Nat → Nat
end

und $K = \mathrm{Alg}(\mathrm{LISTNATI})$. Dann ist die Algebra $\mathbb{N}^*$ der endlichen Sequenzen natürlicher Zahlen initial.

Dagegen sind die Algebra $\mathbb{Z}^*$ der endlichen Sequenzen ganzer Zahlen, die Algebra der *unendlichen* Sequenzen natürlicher Zahlen und die Algebra der endlichen Mengen (wobei cons das Einfügen einer Zahl in eine endliche Menge bezeichnet) *nicht initial* in K.

Begründung: Gemäß Beispiel 24 ist die Grundtermalgebra $T(LISTNATI)$ initial. $\mathbb{N}^*$ ist isomorph zu $T(LISTNATI)$:

$$\begin{array}{llll} \mathrm{nil} & \mathrm{cons}(n,\mathrm{nil}) & \mathrm{cons}(n,\mathrm{cons}(m,\mathrm{nil})) & \ldots \\ \updownarrow & \updownarrow & \updownarrow & \\ \varepsilon & \langle n \rangle & \langle n,m \rangle & \end{array}$$

◀

Initiale Algebren erfüllen genau die Gleichungen zwischen Grundtermen, die in allen Modellen einer Klasse von Algebren gelten.

Satz 15 *Sei $\Sigma = (S,F)$ eine bewohnte Signatur, K eine Klasse von Σ-Algebren und $I \in K$ initial in K. Dann gilt für alle Grundterme $t_1, t_2 \in T(\Sigma)_s$, $s \in S$:*

$$I \models t_1 = t_2 \quad \textit{gdw.} \quad K \models t_1 = t_2$$

Beweis.

"⇒" Gelte $I \models t_1 = t_2$ und sei $A \in K$. Da I initial ist, gibt es genau einen Homomorphismus $\rho : I \to A$. Mit Lemma 13 folgt

$$t_1^A = \rho(t_1^I) = \rho(t_2^I) = t_2^A$$

"⇐" Gelte $K \models t_1 = t_2$. Wegen $I \in K$ folgt trivialerweise $I \models t_1 = t_2$. Q.E.D.

Beispiel 26: Sei SETNATI folgende Signatur:

sig SETNATI =
sorts Set, Nat
ops empty : Set
$\{\_\}$: Nat → Set
$\_\cup\_$: Set × Set → Set
zero : Nat
succ : Nat → Nat
end

und sei $K \subseteq \mathrm{Alg}(\mathrm{SETNATI})$ die Klasse von Algebren, die folgende Eigenschaften erfüllen:

1. $G \cup (H \cup K) = (G \cup H) \cup K$
2. $G \cup H = H \cup G$
3. $G \cup G = G$
4. $G \cup \mathrm{empty} = G$

Dann ist die Algebra der endlichen Mengen natürlicher Zahlen initial und erfüllt genau die Eigenschaften von K. ◀

Initialität ist abgeschlossen unter Isomorphie.

Lemma 16 *Sei K eine Klasse von Σ-Algebren und $I \in K$ initial in K. Ist $I' \in K$ isomorph zu I, dann ist I' ebenfalls initial in K.*

Beweis. Da $I' \in K$ isomorph zu I ist, gibt es genau einen Σ-Homomorphismus $\phi : I' \to I$.

Da I initial ist, gibt es für jedes $A \in K$ gibt es genau einen Σ-Homomorphismus $\rho : I \to A$.

Also ist $\rho \circ \phi : I' \to A$ ein eindeutig bestimmter Σ-Homomorphismus. Q.E.D.

Definition 17 *Ein* abstrakter Datentyp *für eine Signatur Σ ist eine Klasse von Σ-Algebren, die abgeschlossen ist unter Isomorphie, d.h. eine Klasse $C \subseteq \mathrm{Alg}(\Sigma)$ mit folgender Eigenschaft:*

Falls $A \in C$ und B ist isomorph zu A, so ist auch $B \in C$.

Beispiel 27:

1. Die Klasse der initialen Algebren von *K* bildet einen abstrakten Datentyp.
2. Das Strichzahlenmodell und das Binärzahlenmodell bilden zwei isomorphe Elemente der Klasse der initialen Algebren von NAT0. ◄

Eine weitere interessante und für die Beweisverfahren wichtige Teilklasse von $\mathrm{Alg}(\Sigma)$ sind die erreichbaren (oder termerzeugten) Algebren, für die eine surjektive Abbildung $T(\Sigma) \to A$ existiert.

Definition 18 *Eine* Σ*-Algebra A heißt* Σ-erreichbar (Σ-termerzeugt, „reachable"), *wenn jedes Element von A einen Grundterm interpretiert, d.h. wenn es für alle* $s \in S$ *und* $a \in A_s$ *einen Grundterm* $t \in T(\Sigma)$ *gibt mit* $t^A = a$.

Beispiel 28:

1. Das Standardmodell $N = (\mathbb{N}, 0, \_ + 1)$ ist NAT0-erreichbar.

$$\begin{array}{cccc} 0 & 1 & 2 & \dots \\ \updownarrow & \updownarrow & \updownarrow & \\ \mathrm{zero}^N & \mathrm{succ}(\mathrm{zero})^N & \mathrm{succ}(\mathrm{succ}(\mathrm{zero}))^N & \dots \end{array}$$

2. N_2 ist NAT0-erreichbar.

$$\begin{array}{cc} 0 & 1 \\ \uparrow & \uparrow \\ \mathrm{zero}^N & \mathrm{succ}(\mathrm{zero})^N \end{array}$$

3. Z ist *nicht* NAT0-erreichbar (da es keinen surjektiven NAT0-Homomorphismus von der Termalgebra der Signatur NAT0 nach Z gibt).

$$\begin{array}{cccccc} \underbrace{\dots \quad -2 \quad -1} & & 0 & 1 & 2 & \dots \\ & & \updownarrow & \updownarrow & \updownarrow & \\ \text{„junk" (Burstall, Goguen ca. 1980)} & & \mathrm{zero}^N & \mathrm{succ}(\mathrm{zero})^N & \mathrm{succ}(\mathrm{succ}(\mathrm{zero}))^N & \dots \end{array}$$ ◄

Beispiel 29:

1. N, N_2, N_1 sind NAT0-erreichbar,
2. $\mathbb{N}^*$ ist LISTNATI-erreichbar,
3. $\mathcal{P}^{\mathrm{fin}}(()N)$ ist SETNATI-erreichbar. ◄

Lemma 19 *Eine* Σ*-Algebra A ist* Σ*-erreichbar gdw. A ist Bild eines* Σ*-Homomorphismus von* $T(\Sigma)$ *nach A.*

Beweis. Betrachte den eindeutig bestimmten Σ-Homomorphismus $\rho : T(\Sigma) \to A$. Offensichtlich ist die Surjektivität von ρ äquivalent zur Erreichbarkeit von *A*. Q.E.D.

Folgerung 20 *Sei A* Σ*-erreichbar. Dann gibt es höchstens einen* Σ*-Homomorphismus* $\rho : A \to B$ *von A nach B, der durch* $\rho(t^A) = t^B$ *eindeutig bestimmt ist.*

Beweis. Sei $\rho : A \to B$ ein Σ-Homomorphismus.

Nach Lemma 13 gilt für alle Grundterme t, dass $\rho(t^A) = t^B$. Da A Σ-erreichbar ist, ist ρ damit schon für alle Elemente der Trägermengen von A eindeutig definiert. Q.E.D.

In vielen praktischen Fällen ist die initiale Algebra erreichbar.

Satz 21 *Sei Σ eine bewohnte Signatur, K eine Klasse von Σ-Algebren, die durch eine Menge E von Axiomen der Form*

$$\forall x_1 : s_1 \ldots x_n : s_n.\ G \qquad \textit{mit } G \textit{ quantorenfrei}$$

charakterisiert ist.

Eine Σ-Algebra I ist initial in K gdw.

1. *E gilt in I,*
2. *I ist Σ-erreichbar,*
3. *für alle Grundterme $t_1, t_2 \in T(\Sigma)_s$, $s \in S$ gilt:*

$$I \models t_1 = t_2 \quad \textit{gdw.} \quad K \models t_1 = t_2$$

Beweis.
"$\Rightarrow$": Sei I initial in K.

1. E gilt in I, da $I \in K$.
2. Da E allquantifiziert ist, gilt E auch in der Σ-erreichbaren Unteralgebra $\langle I \rangle_\Sigma$ von I. Da I initial ist, gibt es einen Σ-Homomorphismus $\rho : I \to \langle I \rangle_\Sigma$. Andererseits existiert der Einbettungshomomorphismus $in : \langle I \rangle_\Sigma \to I$. Also sind I und $\langle I \rangle_\Sigma$ isomorph. Da nun $\langle I \rangle_\Sigma$ Σ-erreichbar ist, ist auch I Σ-erreichbar.
3. folgt aus Satz 15.

"$\Leftarrow$": (1), (2), (3) sollen gelten, d.h. E gelte in I, I sei Σ-erreichbar und für alle Grundterme t_1, t_2 gilt $I \models t_1 = t_2$ gdw. $A \models t_1 = t_2$ für alle $A \in K$. Dann ist $I \in K$, ferner ist I initial in K: Zu zeigen ist, dass es für beliebiges $A \in K$ genau einen Σ-Homomorphismus $\rho : I \to A$ gibt.

Sei also $A \in K$. Definiere $\rho(t^I) =_{\text{def}} t^A$ für $t \in T(\Sigma)$. Die Abbildung ρ ist wohldefiniert, denn sei $t^I = u^I$, d.h. $I \models t = u$, so gilt wegen (3) auch $A \models t = u$ und somit $t^A = u^A$. Gemäß Folgerung 20 ist ρ der einzige Σ-Homomorphismus von $I \to A$. Q.E.D.

Zeige: Die kleinste Subalgebra von A ist Σ-erreichbar.

Beweis. Betrachte

$$(\langle A \rangle_\Sigma)_s = \{a \in A_s \mid \text{es gibt } t \in T(\Sigma)_s \text{ mit } t^A = a\}$$

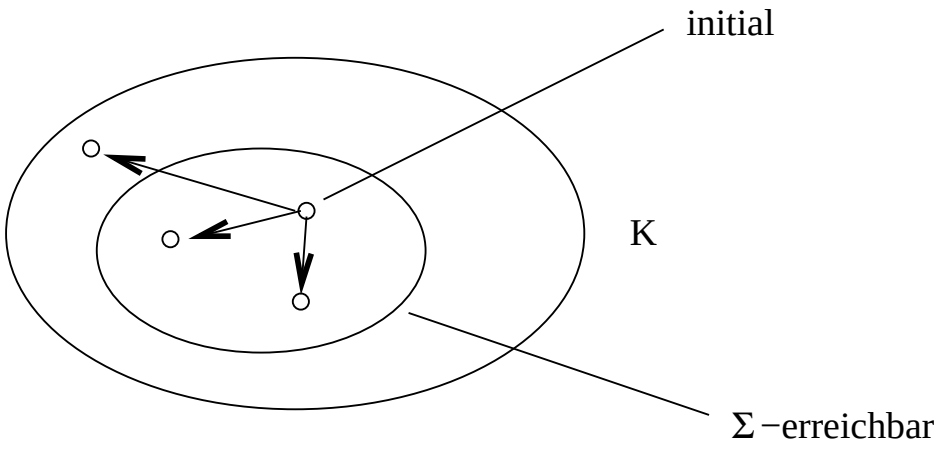
ist erreichbare Σ-Subalgebra, wenn $f^{\langle A \rangle_\Sigma} = f^A \mid_{\langle A \rangle_\Sigma}$. Sei I initial in K. Betrachte $\langle I \rangle_\Sigma$ kleinste Σ-Subalgebra. Es gilt:

1. $\langle I \rangle_\Sigma$ ist Σ-erreichbar.
2. Da I initial, existiert genau ein Σ-Homomorphismus $I \to \langle I \rangle_\Sigma$.
3. Σ-Einbettungshomomorphismus $in : \langle I \rangle_\Sigma \to I$ Also sind I und $\langle I \rangle_\Sigma$ isomorph. Q.E.D.

Folglich: Σ-Erreichbarkeit fordert „no junk"! (Also keine Elemente von A, die nicht durch mindestens einen Grundterm bezeichnet werden.)

Σ-Initialität fordert zusätzlich (falls die Gesetze der Form $\forall x : s.\ G$ sind) „no confusion" d.h.

$$I \models t_1 = t_2 \Rightarrow K \models t_1 = t_2$$



Aus der Erreichbarkeit folgt das Prinzip der strukturellen Induktion. Sei K eine Teilklasse der erreichbaren Σ-Algebren. Dann gilt folgendes Induktionsprinzip:

Satz 22 (strukturelle Induktion) *Sei $\Sigma = (S, C)$, G Formel 1. Stufe, K eine Teilklasse der erreichbaren Σ-Algebren und $s \in S$. Wenn*

1. $K \models G[c/x]$ *für alle $c \in C_{\varepsilon,s}$ („G gilt für alle Konstanten") und*
2. $K \models \forall y_1 : s_1 \ldots \forall x_1, \ldots, x_n : s.\ G[x_1] \wedge \ldots \wedge G[x_n] \Rightarrow G[f(x_1, \ldots, x_n, y_1, \ldots)]$
   für alle $f \in C_{\langle\langle s,\ldots,s,s_1,\ldots\rangle,s\rangle}$ gilt,

dann gilt auch: $K \models \forall x : s.\ G[x]$.

Bemerkung: Beachten Sie, dass in der Induktionsvoraussetzung die Formel G nur für Argumente der Sorte s auftritt, über die der Induktionsbeweis geführt werden soll. (Im allgemeinen ist simultane Induktion über mehrere Argumentsorten notwendig.) Tatsächlich ist die Induktionsbasis (1) ein Spezialfall des Induktionsschritts (2) für $n = 0$.

Beispiel 30: Strukturelle Induktion

1. Die strukturelle Induktion für natürliche Zahlen lautet:

$$\frac{G(\text{zero}) \qquad \forall x : \text{Nat}.\ G[x] \Rightarrow G[\text{succ}(x)]}{\forall x : \text{Nat}.\ G[x]}$$

   (Signatur $\text{NAT0} = (\{\text{Nat}\}, \{\text{zero}, \text{succ}\})$)

2. Die strukturelle Induktion für Wahrheitswerte lautet:

$$\frac{G[\text{true}] \qquad G[\text{false}]}{\forall x : \text{Bool}.\ G[x]}$$

3. Die strukturelle Induktion für die Sorte List von LISTNATI lautet:

$$\frac{G[\text{nil}] \qquad \forall y : \text{Nat}.\ \forall s : \text{List}.\ G[s] \Rightarrow G[\text{cons}(y,s)]}{\forall s : \text{List}.\ G[s]}$$

Man sieht, das dieses Induktionschema nicht auf die Funktionssymbole für natürliche Zahlen Bezug nimmt und damit auch für die Signatur LIST0 gilt:

sig LIST0 =
sorts List, Elem
ops nil : List
cons : Elem × List → List
end

LIST0 ist nicht bewohnt, da für Elem kein Grundterm existiert. Also können unsere Sätze zur Initialität nicht angewendet werden. ◀

3.4 Eine Verallgemeinerung von Initialität: Freie Erweiterung

Listen und Mengen kann man als generische Strukturen auffassen, für die es unerheblich ist, welche Sorte von Elementen sie verwenden. Daher kann man nur schwer Konstruktoren für die Sorte Elem der Elemente angeben, wenn man sich nicht auf eine spezielle Sorte von Elementen festlegen will. Mathematisch hat dies zur Folge, dass z.B. die Signatur LIST0 der generischen Listen nicht bewohnt ist und die Klasse aller LIST0-Algebren keine initiale Algebra besitzt.

Deshalb verallgemeinert man den Begriff der Initialität zur dem der freien Erweiterung.

Definition 23 *Seien Σ_0, Σ_1 Signaturen mit $\Sigma_0 \subseteq \Sigma_1$.*

1. *Sei A eine Σ_1-Algebra. Das* Σ_0-Redukt $A|_{\Sigma_0}$ *von A entsteht aus A durch Weglassen der Sorten und Funktionssymbole, die nicht in Σ_0 vorkommen, d.h.*

$$\begin{array}{rcll} (A|_{\Sigma_0})_s & =_{\text{def}} & A_s & \text{für alle } s \in S_0 \\ f^{A|_{\Sigma_0}} & =_{\text{def}} & f^A & \text{für alle } f \in F_0 \end{array}$$

2. *Sei K eine Klasse von Σ_1-Algebren. A heißt* freie Erweiterung *von $A|_{\Sigma_0}$, wenn für jedes $B \in K$ und jeden Σ_0-Homomorphismus $h : A|_{\Sigma_0} \to B|_{\Sigma_0}$ es genau einen Σ_1-Homomorphismus $h^* : A \to B$ gibt mit $h^*|_{\Sigma_0} = h$.*

Abbildung 4 veranschaulicht den Begriff der freien Erweiterung. Sei $\Sigma_0 = (\{\text{Elem}\}, \emptyset)$ etwa die Signatur, die nur die Sorte Elem einführt, und Σ_1 die Signatur LIST0 aus Beispiel 30. Ferner sei A die Algebra der Listen natürlicher Zahlen (mit $A|_{\Sigma_0} = N$) und B die Algebra der Multimengen von $\{0,1\}$ (mit $B|_{\Sigma_0} = N_2$). Der Homomorphismus

$$h : \begin{cases} A|_{\Sigma_0} \to B|_{\Sigma_0} \\ n \mapsto n \bmod 2 \end{cases}$$

lässt sich auf genau eine Weise zu einem Homomorphismus $h^* : A \to B$ fortsetzen durch

$$h^*(\langle n_1, \ldots, n_k \rangle) = \{n_1 \bmod 2, \ldots, n_k \bmod 2\}$$

(das heißt, h^* ordnet jeder Liste natürlicher Zahlen die Multimenge der Paritäten ihrer Elemente zu).

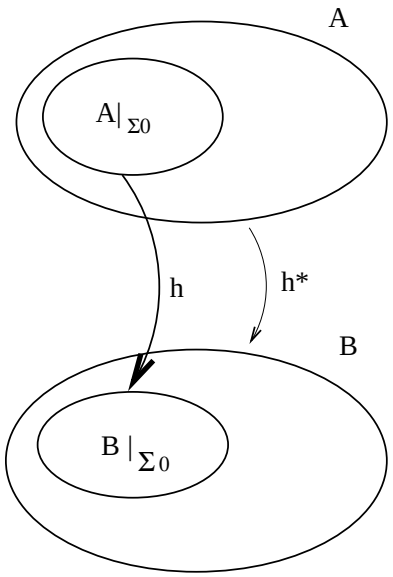


Abbildung 4: Begriff der freien Erweiterung.

Beispiel 31: Freie Erweiterung

$\mathbb{N}^*$ ist eine freie Erweiterung von $\mathbb{N}$ in der Klasse aller LIST0-Algebren (bzgl. der Subsignatur bestehend nur aus der Sorte Elem).

$\mathcal{P}^{\text{fin}}(\mathbb{Z})$ ist eine freie Erweiterung von $\mathbb{Z}$ in der Klasse aller SET0-Strukturen, für die empty ein neutrales Element ist, $\cup$ assoziativ, idempotent und kommutativ ist, und $\in$ die Elementrelation darstellt. ◄

3.5 Zusammenfassung

- Signaturen sind ein formaler Ansatz zur Beschreibung von Schnittstellen. Eine Signatur besteht aus der Angabe von Sorten, Funktionssymbolen und Prädikatensymbolen.
- Durch Σ-Algebren (bei Signaturen ohne Prädikatensymbole) und Σ-Strukturen kann man Schnittstellen eine Interpretation zuordnen.
- Eigenschaften von Schnittstellen bzw. von Strukturen beschreibt man durch Σ-Formeln. Man unterscheidet u.a. aussagenlogische Formeln, Gleichungen, bedingte Gleichungen und allgemeine Formeln der Prädikatenlogik 1. Stufe, für die jeweils spezielle Beweismethoden bzw. Beweiskalküle existieren.
- In der Klasse der Σ-Algebren sind erreichbare und initiale Algebren von besonderem Interesse. Erreichbarkeit impliziert die Gültigkeit der strukturellen Induktion, einem wichtigen Beweisschema für benutzerdefinierte Datenstrukturen. Initialität beschreibt einen abstrakten Datentyp, der genau die geforderten Gleichungen erfüllt und mit Hilfe von Termersetzung ausführbar ist.
- Die freie Erweiterung verallgemeinert den Begriff der Initialität für generische Datentypen.

4 Algebraische Spezifikation mit CASL

Ziele

- Algebraische Spezifikationen kennen lernen
- Erste Spezifikationen in CASL schreiben lernen
- Erreichbarkeit und freie Erweiterung für CASL-Spezifikationen einsetzen.

Algebraische axiomatische Spezifikationen beschreiben eine Klasse von (Rechen-)Strukturen abstrakt, d.h. durch Angabe einer Signatur, bestehend aus

- Sorten für die Objektmengen
- Funktionssymbolen für die Funktionen über Objektmengen
- Prädikatensymbolen für die Relationen zwischen Objekten

und einer (i.a. endlichen) Menge von charakteristischen Eigenschaften (i.a. prädikatenlogische Formeln 1. Stufe)

4.1 Einfache algebraische Spezifikationen

Definition 24 *Sei* $\Sigma = (S, F, P)$ *eine Signatur und E eine Menge von (geschlossenen)* Σ*-Formeln. Dann ist* $SP = \langle \Sigma, E \rangle$ *eine* axiomatische Spezifikation.

Man nennt SP algebraische Spezifikation*, wenn P leer ist (d.h. wenn nur das Gleichheitssymbol als Prädikat vorkommt).*

Bemerkung: In der Logik bezeichnet man $\langle \Sigma, E \rangle$ als Präsentation einer Theorie.

Die Semantik von *SP* ist gegeben durch die *Signatur* und die *Klasse der Modelle*:

Signatur:	$\mathrm{Sig}(\langle \Sigma, E \rangle) = \Sigma$
Modellklasse:	$\mathrm{Mod}(\langle \Sigma, E \rangle) = \{ A \in \mathrm{Struct}(\Sigma) \mid A \models E \}$

In CASL spezifiziert man Axiome durch Angabe der freien (allquantifizierten) Variablen, gefolgt von einer oder mehreren prädikatenlogischen Formeln.

Beispiel 32: Assoziativität

spec ASSOC =
sorts Elem
ops $\_ \circ \_ : \mathrm{Elem} \times \mathrm{Elem} \to \mathrm{Elem}$
vars $x, y, z : \mathrm{Elem}$
axioms $(x \circ y) \circ z = x \circ (y \circ z)$
end

Hier wird die Möglichkeit verwendet, Axiome durch das Schlüsselwort **axiom** bzw. **axioms** einzuleiten.

◀

Beispiel 33: Spezifikation der Rechenstruktur BOOL

spec BOOL1 =
sorts Bool
ops $\text{true}, \text{false} : \text{Bool}$
axioms $\neg(\text{true} = \text{false})$;
$\forall x : \text{Bool}.\ x = \text{true} \vee x = \text{false}$
end

In der folgenden Spezifikation werden zusätzlich die Booleschen Verknüpfungen not, and, or eingeführt.

Die Erweiterung einer Spezifikation um zusätzliche Sorten, Operationen und Axiome wird mit dem Schlüsselwort **then** eingeleitet.

spec BOOL =
BOOL1 **then**
ops $\text{not}\_ : \text{Bool} \to \text{Bool}$;
$\_\ \text{and}\ \_,\ \_\ \text{or}\ \_ : \text{Bool} \times \text{Bool} \to \text{Bool}$
vars $x : \text{Bool}$
axioms $\text{not}(\text{true}) = \text{false}$;
$\text{not}(\text{false}) = \text{true}$;
$\text{true}\ \text{and}\ x = x$;
$\text{false}\ \text{and}\ x = \text{false}$;
$\text{true}\ \text{or}\ x = \text{true}$;
$\text{false}\ \text{or}\ x = x$
end

◀

Beispiel 34: Spezifikation der Gruppen

spec GROUP =
sorts Group
ops $\text{n} :\to \text{Group}$ %%{(neutrales Element)}%
$\_^{-1} : \text{Group} \to \text{Group}$ %%{(Inversenbildung)}%
$\_ \circ \_ : \text{Group} \times \text{Group} \to \text{Group}$ %%{(Verknüpfung)}%
vars $x, y, z : \text{Group}$
axioms %[assoc] $(x \circ y) \circ z = x \circ (y \circ z)$
%[ln] $\text{n} \circ x = x$
%[li] $x^{-1} \circ x = \text{n}$
end

◀

4.2 Spezifikationen mit Konstruktoren

In CASL kann die Erreichbarkeit der erlaubten Modelle durch eine Menge von Konstruktoren explizit gefordert werden.

Definition 25 (Konstruktoren) *Gegeben sei $\Sigma = (S, F, P)$ und $s \in S$.*

1. *Ein Ausdruck g der Form*

   generated { **sorts** *s*; **ops** *C* }

   führt die Sorte s und die Operationen C als Konstruktoren *für die Sorte s ein.*

   Man spricht von Sorten-Erzeugung *(oder „Erreichbarkeitsaxiom", „generating constraint") und fordert syntaktisch, dass jedes $c \in C$ die Sorte s als Wertebereich besitzt, d.h.*

   $$C \subseteq \bigcup_{w \in S^*} F_{\langle w,s \rangle}$$

2. *Eine Σ-Struktur A* erfüllt *g ($A \models g$), wenn jedes Element a von A_s durch einen Konstruktorterm interpretiert werden kann, der nur Funktionssymbole aus C und freie Variablen für Sorten ungleich s enthält, d.h. wenn a zu einem Term $t \in T((S,C),(X_{s'})_{s' \neq s})$ äquivalent ist, oder formal: für alle $a \in A_s$ gilt*

   $$A, \nu \models x = t$$

   für eine Belegung $\nu : X \to A$ mit $\nu(x) = a$ (und $x \in X_s$) und einen Term $t \in T((S,C),(X_{s'})_{s' \neq s})$.

   Wir sagen in diesem Fall auch, dass C Konstruktormenge *für s ist.*

Ist *C* eine Menge von Konstruktoren für die Sorte *s* in der Algebra *A*, so bedeutet dies informell, dass alle Elemente von A_s Interpretationen eines Terms sind, der nur mit Funktionssymbolen aus *C* konstruiert ist.

Beispiel 35: Die Angabe der Konstruktormenge ersetzt das Axiom, das die Wertemenge von Bool auf true, false beschränkt.

spec BOOL_g =
  generated { **sorts** Bool; **ops** true, false : Bool }
  axioms ¬ (true = false)
end

Analyse. Alle Modelle von BOOL_g sind isomorph zum Standardmodell *B* mit:

$$B_{\text{Bool}} = \{\text{T}, \text{F}\}, \qquad \text{true}^B = \text{T}, \qquad \text{false}^B = \text{F}$$

Abkürzung für **generated** { **sorts** Bool; **ops** true, false: Bool }:

generated type Bool ::= true | false

◀

Bemerkung: In der Literatur werden häufig als *(flache) algebraische Spezifikationen* bezeichnet: algebraische Spezifikationen, bei denen

1. jede Sorte s durch „**generated** { **sorts** *s*, **ops** $F_{w,s}$ }" erzeugt wird, (z. B. BOOL_g) und
2. die Axiome aus Gleichungen oder bedingten Gleichungen bestehen.

Konstruktormengen können auch durch eine Datentypdeklaration in BNF-Form angegben werden. Eine Datentypdeklaration für die Sorte s wird geschrieben als

$$s ::= A_1 \mid \ldots \mid A_n$$

wobei die A_i Konstruktoren der Form

$$c(s_1; \ldots; s_k)$$

sind. Dabei bedeutet $c(s_1; \ldots; s_k)$, dass der Konstruktor c die Funktionalität

$$c : s_1 \times \ldots \times s_k \rightarrow s$$

besitzt.

Beispiel 36:

spec BOOL_g =
generated type Bool ::= true | false
axioms $\neg$ (true = false)
end

◄

Weitere Beispiele für flache algebraische Spezifikationen mit Konstruktoren sind:

Beispiel 37:

1. **Natürliche Zahlen**

spec NAT_g =
generated type Nat ::= zero | succ(Nat)
vars x, y : Nat
axioms $\neg$ (zero = succ(x))
succ(x) = succ(y) $\Rightarrow$ $x = y$
end

Analyse. Die Spezifikation NAT_g besitzt bis auf Isomorphie genau das Standardmodell der natürlichen Zahlen.

2. **Ganze Zahlen**

spec INT_g =
generated type Int ::= zero | succ(Int) | pred(Int)
vars x : Int
axioms succ(pred(x)) = x
pred(succ(x)) = x
end

Analyse. Die Spezifikation INT_g hat das **Standardmodell** der ganzen Zahlen als **initiales Modell**, besitzt aber außerdem noch viele andere Modelle wie etwa die endlichen zyklischen Modelle Z_k (für $k > 0$). Auch die einelementige Struktur ist ein Modell von INT_g.

Frage: Welche Axiome müssen zu INT_g hinzugenommen werden, um das Standardmodell der ganzen Zahlen zu axiomatisieren? (Hinweis: Verwende Hilfsfkt „<“)

3. **Ein einfacher Verbunddatentyp: Person** Der Datentyp Person als Paar aus Name und Adresse (vgl. Abb. 5).

spec PERSON =
sorts String, Address
generated type Person ::= makePerson(String; Address)
ops getName : Person → String
getAddress : Person → Address
vars n : String; a : Address
axioms getName(makePerson(n,a)) $= n$
getAddress(makePerson(n,a)) $= a$
end

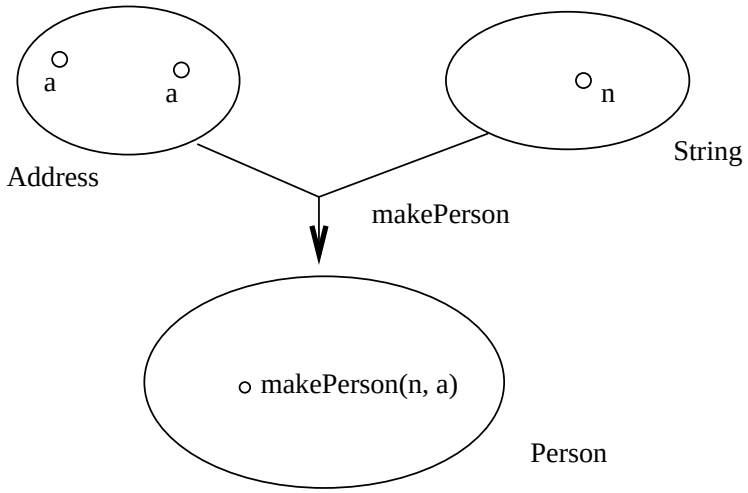


Abbildung 5: Erzeugung der Sorte Person aus String und Address.

Analyse. Die Operationen getName und getAddress sind „Selektoren“, die den Wert der Argumente des Konstruktors selektieren.

Die Trägermengen der Sorte Person werden durch die Terme makePerson(t,t') erzeugt. Die Selektoren sichern, dass makePerson injektiv ist.

Da die Axiome von PERSON keine Ungleichungen enthalten, gibt es viele unterschiedliche Modelle. Für jedes Modell A gilt, dass die Anzahl der Elemente der Trägermenge A_{Person} von Person gleich der Anzahl der Elemente von A_{String} multipliziert mit der Anzahl der Elemente von A_{Address} ist. Für alle Modelle A von PERSON muss A_{Person} erreichbar sein mit makePerson.

Beachte, dass es keine Terme der Sorten String und Address gibt. In einem solchen Fall ist der Begriff der *relativierten Erreichbarkeit* angemessen:

$$A_{\text{Person}} = \{\text{makePerson}^A(n,a) \mid n \in A_{\text{String}}, a \in A_{\text{Address}}\}$$

ist. Dadurch wird folgendes strukturelle Induktionsprinzip induziert:

$$\frac{\forall x : \text{String}.\ \forall y : \text{Address}.\ G[\text{makePerson}(x,y)]}{\forall p : \text{Person}.\ G[p]}$$

Adressen können ebenfalls als Verbunddatentypen spezifiziert werden:

spec ADDRESS =
  STRING **and** INT **then**
  generated type Address ::= makeAddress(String; Int; String)
  axioms ...
end

Eine adäquatere Beschreibung dieser Situation erreicht man durch hierarchische Strukturierung der Spezifikation:

spec STRING =
  sorts String
end

spec ADDRESS =
  STRING and INT **then**
  generated type Address ::= makeAddress(String; Int; String)
  axioms ...
end

spec PERSON =
  STRING and ADDRESS **then**
  generated type Person ::= makePerson(String; Address)
  ops getName : Person → String
    getAddress : Person → Address
  vars n : String; a : Address
  axioms getName(makePerson$(n,a)) = n$
    getAddress(makePerson$(n,a)) = a$
end

◀

Allgemein ist ein Selektor eine inverse Funktion zu einem Konstruktor. Für jeden Selektor $sel_i : s \to s_i$ (für $i = 1,\ldots,k$) zu einem Konstruktor c kann das Axiom

$$sel_i(c(x_1,\ldots,x_i,\ldots x_k)) = x_i$$

angegeben werden.

CASL bietet eine abkürzende Notation für Selektoren direkt als Argumente des Konstruktors bei der Datentypdeklaration:

$$s ::= \ldots \mid c(\ldots; sel_i : s_i; \ldots) \mid \ldots$$

Dadurch werden die Axiome für die Selektoren induziert.

Beispiel 38: Angabe von Selektoren in Konstruktordeklarationen

1. **Person**

   Bei Angabe der Selektoren in der Datentypdeklaration müssen die Selektoraxiome nicht explizit geschrieben werden.

   spec PERSON_g =
   sorts String, Person
   generated type Person ::= makePerson(getName : String; getAddress : Address)
   end

2. **Ganze Zahlen**

   spec INT_g =
   generated type Int ::= zero | succ(pred : Int) | pred(succ : Int)
   end

◀

Über Spezifikationen mit Erreichbarkeitsbedinungen kann man weitere Funktionen und Prädikate durch strukturelle Rekursion definieren; technisch wählt man für die zu definierende Funktion ein Argument, für das für jeden Konstruktor ein Axiom angegeben wird.

Beispiel 39: strukturelle Rekursion zur Definition von Funktionen und Prädikaten

Über NAT_g kann man weitere Funktionen und Prädikate durch strukturell rekursive Axiome einführen.

spec NAT1 =
NAT_g **then**
ops $\_+\_ : \text{Nat} \times \text{Nat} \to \text{Nat}$
preds $< : \text{Nat} \times \text{Nat}$
vars $n, m : \text{Nat}$
axioms $n + \text{zero} = n$
$n + \text{succ}(m) = \text{succ}(n + m)$
$\neg\ (n < \text{zero})$
$\text{zero} < \text{succ}(m)$
$(\text{succ}(n) < \text{succ}(m)) \equiv (n < m)$
end

◀

Eine andere Möglichkeit sind *explizite Definitionen* der Form

$f(x) = \ldots$ bzw. $p(x) \equiv \ldots$

für Funktions- bzw. Prädikatensymbole. Dies ist immer möglich, auch ohne Erreichbarkeitsbedingungen.

Beispiel 40:

spec NAT2 =
NAT1 **then**
ops double : Nat → Nat
preds $\_ \leq \_ : \text{Nat} \times \text{Nat}$

vars n, m : Nat
axioms $\text{double}(n) = n + n$
$(n \leq m) \equiv (n < m \vee n = m)$
end

◀

Man kann die strukturelle Rekursion verfeinern zur Rekursion über mehrere Argumente. Dann muss für die angegebenen Argumente eine Menge von Konstruktortermen gefunden werden, die alle Fälle überdecken, vgl. die Definition von $<$ in NAT1.

4.3 Spezifikation initialer und freier Strukturen

Zur Spezifikation initialer und freier Strukturen bietet CASL das Konstrukt **free** , das ganz analog zu **generated** verwendet wird.

Definition 26 (Freie Substrukturen)

1. *Gegeben sei $\Sigma = (S, F, P)$ und $s \in S$. Ein Ausdruck g der Form*

   free { **sorts** s; **ops** C }

   führt die Sorte s und die Operationen C als freie Konstruktoren *für die Sorte s ein.*

   Wie bei **generated** *fordert man syntaktisch, dass jedes $c \in C$ die Sorte s als Wertebereich besitzt, d.h.*

   $$C \subseteq \bigcup_{w \in S^*} F_{\langle w,s \rangle}$$

2. *Sei S_0 die Menge der in C vorkommenden Sorten ungleich s. Eine Σ-Struktur A* erfüllt *g ($A \models g$), wenn die Substruktur $A|_{(S,C)}$ eine freie Erweiterung von $A|_{(S_0,\emptyset)}$ ist.*

Beispiel 41:

1. **Natürliche Zahlen**

   Das Standardmodell N der natürlichen Zahlen ist initial (und damit frei) in der Klasse aller Algebren von NAT. Durch Angabe von **free** braucht man die Axiome von NAT nicht explizit aufzuschreiben.

   spec NAT_f =
   free type Nat ::= zero | succ(Nat)
   end

2. **Ganze Zahlen**

   spec INT_f =
   free type Int ::= zero | succ(pred : Int) | pred(succ : Int)
   end

Analyse. Jedes Modell von INT_f ist isomorph zum Standardmodell der ganzen Zahlen. Die Modelle von INT_f sind initial in der Modellklasse von INT_g.

3. **Verbundtypen**

spec PERSON_f =
sorts String, Address
free type Person ::= makePerson(getName : String; getAddress : Address)
end

Analyse. Aufgrund der Angabe der Selektoren besteht bei PERSON kein Unterschied zu der Semantik von **generated** . Da String und Address verschieden von Person sind, wird die freie Erweiterung der Trägermengen dieser Sorten gebildet. Die Selektoren sichern, dass die Funktion makePerson injektiv ist. Die Bedingung der „Freiheit“ hat hier keinen zusätzlichen Effekt.

4. **Listen**

Die Struktur der endlichen Listen ist eine freie Erweiterung des Elementtyps Elem.

spec LIST0 =
sorts Elem
free type List ::= nil | cons(Elem; List)
ops first : List $\rightarrow ?$ Elem
rest : List $\rightarrow ?$ List
vars x : Elem; l : List
axioms $\neg$ def first(nil)
$\neg$ def rest(nil)
first(cons$(x,l)) = x$
rest(cons$(x,l)) = l$
end

Hier bedeutet die Angabe des Fragezeichens, dass first und rest (möglicherweise) partielle Selektoren sind, die für nil nicht definiert sind. Das Definiertheitsprädikat „def “ erlaubt die genaue Spezifikation der nicht definierten Funktionswerte. In kompakterer Notation kann man schreiben

spec LIST0 =
sorts Elem
free { **type** List ::= nil | cons(first : ?Elem; rest : ?List)
axioms $\neg$ def first(nil)
$\neg$ def rest(nil)
}
end

◄

4.4 Spezifikation grundlegender Rechenstrukturen

4.4.1 Keller

spec STACK =
sorts Elem

free type Stack ::= empty | push(Elem; Stack)
ops top : Stack →? Elem
pop : Stack → Stack
vars d : Elem; s : Stack
axioms ¬ def top(empty)
top(push(d,s)) = d
pop(empty) = empty
pop(push(d,s)) = s

end

Analyse.

- „Bis auf Umbennung äquivalent“ zu Spezifikation LIST0 (Stack ↦ List, empty ↦ nil, push ↦ cons).
- Elemente von A_{Stack} in Modell A (paarweise verschieden):

$$\{\, \text{empty}^A, \text{push}^A, (d_1, \text{empty}^A), \text{push}^A(d_1, \text{push}^A(d_2, \text{empty}^A)), \ldots \mid d_i \in A_{\text{Elem}} \,\}$$

Keller mit Error-Elementen

spec EELEM =
sorts EElem
ops errord : EElem
end

spec ESTACK =
EELEM **then**
free { **type** EStack ::= errorst | empty | push(EElem; EStack)
vars d : EElem; s : EStack
axioms push(d, errorst) = errorst
push(errord, s) = errorst
}
ops top : EStack → EElem
pop : EStack → EStack
vars d : EElem; s : EStack
axioms top(errorst) = errord
pop(errorst) = errorst
top(empty) = errord
pop(empty) = errorst
top(push(d,s)) = d
pop(push(d,s)) = s

end

Analyse. Die Spezifikation wird durch die („Standard“-)Definition der Errorwerte aufgebläht. Sie ist nicht viel allgemeiner als die Spezifikation mit partieller Funktion für top.

Dieses Beispiel entspricht „strikter“ Fehlerbehandlung:

$$f(\ldots \text{error} \ldots) = \text{error}$$

Es gibt aber auch andere Möglichkeiten, Fehler zu behandeln (vgl. SML-Fehlerbehandlung).

4.4.2 Binäre Bäume mit Knotenmarkierungen

spec TREE =

sorts	Elem
free type	Tree ::= empty $\mid$ node(Tree; Elem; Tree)
ops	left, right : Tree $\to$ Tree
	label : Tree $\to ?$ Elem
vars	t_1, t_2 : Tree; d : Elem
axioms	left(empty) = empty
	$\neg$ def label(empty)
	right(empty) = empty
	left(node(t_1, d, t_2)) = t_1
	label(node(t_1, d, t_2)) = d
	right(node(t_1, d, t_2)) = t_2

end

Achtung: label(empty) ist undefiniert. Besser wäre ev. die Einführung von Fehlerelementen $\#_{\text{Elem}}$ und $\#_{\text{Tree}}$ mit den Axiomen:

$$\text{label}(\text{empty}) = \#_{\text{Elem}}$$
$$\text{left}(\text{empty}) = \text{right}(\text{empty}) = \#_{\text{Tree}}$$

4.4.3 Lose endliche Mengen

spec LSETNAT =
BOOL1 **and** NAT_g **then**

generated type	Set ::= empty $\mid$ add(Nat; Set)
ops	iselem : Nat $\times$ Set $\to$ Bool
vars	b : Bool; x, y : Nat; s : Set
axioms	iselem$(x,$ empty$)$ = false
	iselem$(x,$ add$(y,s))$ = or(eq(x,y), iselem(x,s))

end

Analyse.

1. Es gilt

$$\text{SETNAT} \models \text{iselem}(x, \text{add}(x,s)) = \text{true}$$

Dagegen ist die Gleichung

$$\text{add}(x,\text{add}(x,s)) = \text{add}(x,s)$$

erfüllbar, aber nicht gültig in allen Modellen.

2. Modelle:

   Die folgenden Sig(LSETNAT)-Algebren sind Modelle von LSETNAT:

   (a) Endliche Listen natürlicher Zahlen

   $$\begin{aligned}
   &L_{\text{Nat}} = \mathbb{N}, \quad L_{\text{Bool}} = \{\text{T},\text{F}\},\ldots \quad \text{(Standardmodelle von NAT und BOOL)}\\
   &L_{\text{Set}} = \mathbb{N}^*\\
   &\text{empty}^L = \varepsilon\\
   &\text{add}^L(x,\langle y_1,\ldots,y_k\rangle) = \langle x,y_1,\ldots,y_k\rangle\\
   &\text{iselem}(x,\langle y_1,\ldots,y_k\rangle) = \text{T},\ \text{falls}\ \exists i \in \{1,\ldots,k\} : x = y_i\\
   &\text{iselem}(x,\langle y_1,\ldots,y_k\rangle) = \text{F sonst}
   \end{aligned}$$

   L ist ein initiales Modell, das z.B die folgenden Terme unterscheidet:

   $$\{\text{empty},\text{add}(n_1,\text{empty}),\text{add}(n_2,\text{add}(n_3,\text{empty})),\ldots\}$$

   da es keine Axiome mit $t =_{\text{Set}} t'$ gibt und daher zwischen diesen Termen keine Gleichheit beweisbar ist.

   (b) Endliche Mengen natürlicher Zahlen

   $$\begin{aligned}
   &M_{\text{Nat}} = \mathbb{N}, \quad M_{\text{Bool}} = \{\text{T},\text{F}\},\ldots \quad \text{(Standardmodelle von NAT und BOOL)}\\
   &M_{\text{Set}} = \{\, u \subseteq N \mid u \text{ endlich}\,\}\\
   &\text{empty}^M = \emptyset\\
   &\text{add}(x,\{y_1,\ldots,y_k\}) = \{x\} \cup \{y_1,\ldots,y_k\}\\
   &\text{iselem}(x,\{y_1,\ldots,y_k\}) = \text{T},\ \text{falls}\ x \in \{y_1,\ldots,y_k\}\\
   &\text{iselem}(x,\{y_1,\ldots,y_k\}) = \text{F sonst}
   \end{aligned}$$

   (c) Endliche Multimengen (Bags) natürlicher Zahlen erfüllen

   $$\text{add}(x,\text{add}(y,s)) = \text{add}(y,\text{add}(x,s))$$

   aber nicht

   $$\text{add}(x,s) = \text{add}(x,\text{add}(x,s))$$

3. Struktur der Modellklasse

   Die oben genannten Modelle sind nicht isomorph. Es gibt überabzählbar viele nicht-isomorphe erreichbare Modelle von LSETNAT. Wird SETNAT um die beiden Axiome

   $$\begin{aligned}
   \text{add}(x,s) &= \text{add}(x,\text{add}(x,s))\\
   \text{add}(x,\text{add}(y,s)) &= \text{add}(y,\text{add}(x,s))
   \end{aligned}$$

   ergänzt, so gibt es unter den Modellen, die auf den Standardmodellen N und B basieren, bis auf Isomorphie nur noch das Modell der endlichen Mengen natürlicher Zahlen.

4.5 Strukturierte Spezifikationen

Wie aus verschiedenen Beispielen ersichtlich, ist es sinnvoll, Spezifikationen systematisch zu konstruieren. Im folgenden werden vier grundlegende Strukturierungskonzepte vorgestellt und semantisch definiert.

Sei *Spec* die Menge aller Spezifikationsausdrücke und *Sign* die Menge aller Signaturen.

Für jedes $SP \in Spec$ sei

- $\mathrm{Sig}(SP) \in Sign$ die Signatur von SP und
- $\mathrm{Mod}(SP) \subseteq \mathrm{Struct}(\mathrm{Sig}(SP))$ bzw. $\mathrm{Mod}(SP) \in \mathrm{Alg}(\mathrm{Sig}(SP))$ die Klasse der Modelle von SP.

4.5.1 Summe zweier Spezifikationen

Die Summe von Spezifikationen ist folgendermaßen definiert:

Definition 27

$$\begin{aligned}\mathrm{Sig}(SP_1 \textbf{ and } SP_2) &= \mathrm{Sig}(SP_1) \cup \mathrm{Sig}(SP_2)\\ \mathrm{Mod}(SP_1 \textbf{ and } SP_2) &= \{A \in Alg(sig(SP_1 \textbf{ and } SP_2)) \mid \\ &\qquad A|_{\mathrm{Sig}(SP_1)} \in \mathrm{Mod}(SP_1) \textit{ und } A|_{\mathrm{Sig}(SP_2)} \in \mathrm{Mod}(SP_2)\}\end{aligned}$$

d.h. Modelle von SP_1 **and** *SP_2 sind alle Modelle der Signatur $\mathrm{Sig}(SP_1) \cup \mathrm{Sig}(SP_2)$, die sowohl die Axiome von SP_1 wie von SP_2 erfüllen („Vereinigung der Signaturen, Durchschnitt der Modelle").*

Beispiel 42: Die Spezifikation NAT **and** BOOL beschreibt folgende Modelle:

$$\begin{aligned}\mathrm{Mod}(\text{NAT } \textbf{and } \text{BOOL}) = \{A \in \mathrm{Alg}(\mathrm{Sig}(\text{NAT } \textbf{and } \text{BOOL})) \mid \\ A|_{\mathrm{Sig}(\text{NAT})} \in \mathrm{Mod}(\text{NAT}), A|_{\mathrm{Sig}(\text{BOOL})} \in \mathrm{Mod}(\text{BOOL})\}\end{aligned}$$

◀

Bemerkung: Gemeinsame Sorten- und Funktionssymbole werden identifiziert („geshared"). Sind die gemeinsamen Teile von $SP1$ und $SP2$ inkonsistent, so auch $SP1$ **and** $SP2$.

Beispiel 43: Die Summe der folgenden Spezifikationen SP_1 und SP_2 ist inkonsistent.

$$\begin{aligned}&SP_1 = \langle(\{s\}, \{a, b : s,\ f : s \to s\}), \neg\,(a = b)\rangle\\ &SP_2 = \langle(\{s\}, \{a, b, c : s\}), a = b\rangle\\ &\mathrm{Sig}(SP_1 \textbf{ and } SP_2) = (\{s\}, \{a, b, c : s,\ f : s \to s\})\\ &\mathrm{Mod}(SP_1 \textbf{ and } SP_2) = \emptyset\end{aligned}$$

◀

4.5.2 Erweiterung

Die Erweiterung einer Spezifikation SP mit **then** um neue Sorten, Funktionssymbole und Axiome kann auf die Summe zurückgeführt werden.

Definition 28 (Erweiterung)

$$\begin{array}{l} SP \textbf{ then sorts } S;\ \textbf{ops } F;\ \textbf{axioms } E \textbf{ end} \quad =_{\text{def}} \\ SP \textbf{ and } \langle (sorts(\text{Sig}(SP)) \cup S, ops(\text{Sig}(SP)) \cup F), E \rangle \end{array}$$

Es ist nötig $sorts(\text{Sig}(SP))$ *und* $ops(\text{Sig}(SP))$ *anzugeben, um eine „wohldefinierte" Signatur zu erhalten, die alle Symbole aus E enthält.*

4.5.3 Kapselung durch „Verstecken" von Symbolen

Das Verstecken von Symbolen dient dazu, Hilfssymbole nicht nach außen sichtbar zu manchen. Ein anderer Grund kann sein, dass einem Kunden gewisse Funktionen nicht zur Verfügung gestellt werden sollen.

Seien S_1, F_1 Listen von Sorten und Funktionssymbolen, und sei $\Sigma = (S,F)$ eine Signatur. Mit

$$\Sigma - (S_1, F_1)$$

bezeichnen wir diejenige Signatur, aus der

- alle Sorten und Funktionssymbole aus S_1 und F_1 sowie
- alle Funktionssymbole , die ein Element von S_1 in ihrer Funktionalität aufweisen

gestrichen sind.

Formal sei $\Sigma - (S_1, F_1)$ die Signatur $\Sigma^- = (S^-, F^-)$ mit

$$\begin{array}{lll} S^- & =_{\text{def}} & S \setminus S_1 \\ F^-_{w,s} & =_{\text{def}} & F_{w,s} \setminus (F_{1_{w,s}} \cup \{ f \in F_{w,s} \mid w \text{ oder } s \text{ enthält ein Element von } S_1 \}) \end{array}$$

Damit kann man das Verstecken von Symbolen durch Reduktbildung definieren:

Definition 29 (Hiding)

$$\begin{array}{rcl} \text{Sig}(SP \textbf{ hide } (S_1, F_1)) & = & \Sigma^- \\ \text{Mod}(SP \textbf{ hide } (S_1, F_1)) & = & \{ B|_{\Sigma^-} \mid B \in \text{Mod}(\Sigma) \} \end{array}$$

Beispiel 44: Kapselung von Hilfsfunktionen

Die folgende CASL-Spezifikation beschreibt das Sortieren von Listen natürlicher Zahlen durch Einfügen. Die Hilfsfunktion insert wird nur innerhalb des Moduls benötigt und nach außen „versteckt".

spec INSSORT =
(LISTNAT **then**
ops insert : Nat × List → List
sort : List → List
vars
axioms insert(x, empty) = cons(x, empty)
insert(x, cons(y, l)) = cons(x, cons(y, l)) **when** $x \leq y$ **else** cons(y, insert(x, l))
sort(empty) = empty
sort(cons(x, l)) = insert(x, sort(l))
) **hide** insert
end

◀

4.5.4 Umbenennung

Zur Definition der Umbenennung von Symbolen einer Spezifikation benötigen wir noch den Begriff des Signaturmorphismus und des σ-Redukts.

Ein *Signaturmorphismus* ist eine Abbildung zwischen Signaturen, bei der die Funktionalität der abgebildeten Funktionssymbole mit der Abbildung der Sorten verträglich ist.

Definition 30 (Signaturmorphismus)

1. *Seien* $\Sigma = (S,F)$ *und* $\Sigma' = (S',F')$ *Signaturen. Eine Abbildung* $\sigma = (\sigma_{sort}, \sigma_{op})$ *mit*

   $$\sigma_{sort} : S \to S' \qquad \sigma_{op} : F \to F'$$

   heißt Signaturmorphismus *von* Σ *nach* Σ′*, geschrieben* $\sigma : \Sigma \to \Sigma'$*, wenn für alle* $f \in F_{\langle\langle s_1,\ldots,s_n\rangle,s\rangle}$ *gilt*

   $$\sigma_{op}(f) : \sigma_{sort}(s_1), \ldots, s_{sort}(s_n) \to \sigma_{sort}(s)$$

   das heißt, wenn die Funktionalität von $\sigma_{op}(f)$ *mit der Abbildung der Sorten verträglich ist.*

2. *Sei* $\sigma : \Sigma \to \Sigma'$ *ein* injektiver *Signaturmorphismus,* $A \in \mathrm{Alg}(\Sigma)$*. Die* σ-Übersetzung $\sigma(A)$ *von A ist die Algebra mit*

   $$\sigma(A)_{\sigma_{sort}(s)} =_{\mathrm{def}} A_s \qquad \textit{und} \qquad \sigma_{op}(f)^{\sigma(A)} =_{\mathrm{def}} f^A$$

3. *Sei* $\sigma : \Sigma \to \Sigma'$ *Signaturmorphismus,* $B \in \mathrm{Alg}(\Sigma')$*. Das* σ-Redukt $B|_\sigma$ *ist die* Σ*-Algebra mit*

   $$(B|_\sigma)_s =_{\mathrm{def}} B_{\sigma_{sort}(s)} \qquad \textit{und} \qquad f^{B|_\sigma} =_{\mathrm{def}} (\sigma_{op}(f))^B$$

Beispiel 45: Signaturmorphismen

1. Verträglichkeit von σ_{sort} und σ_{op}.
   Ist etwa $\sigma_{sort}(\mathrm{Nat}) = \mathrm{Int}$ und $\sigma_{op}(\mathrm{succ}) = \mathrm{succ}$ und gilt

   succ : Nat → Nat

   in der Signatur Σ, dann muss in der Bildsignatur gelten

   succ : Int → Int

2. Für die Definition der σ-Übersetzung ist die Injektivität von σ notwendig. Sei z.B.

$$\sigma : (\mathrm{Sig}(\mathrm{MONOID}) \cup \{f : \mathrm{Mon}\}) \to \mathrm{Sig}(\mathrm{NAT})$$
$$\sigma_{sort}(\mathrm{Mon}) = \mathrm{Nat}$$
$$\sigma_{op}(e) = \sigma_{op}(f) = \mathrm{zero}$$
$$\sigma_{op}(\circ) = +$$

Dies ist ein wohldefinierter Signaturmorphismus. Betrachte nun den Monoid M mit

$$M_{\mathrm{Mon}} = \mathbb{N} \qquad e^M = 0 \qquad f^M = 1$$

Dann wäre $\sigma(M)$ nicht wohldefiniert, da sowohl

$$\mathrm{zero}^{\sigma(M)} = (\sigma_{op}(e))^{\sigma(M)} = e^M = 0$$

als auch

$$\mathrm{zero}^{\sigma(M)} = (\sigma_{op}(f))^{\sigma(M)} = f^M = 1$$

gelten müsste. Dagegen ist die Reduktbildung für beliebige Signaturmorphismen möglich. ◀

In CASL definiert man einen Signaturmorphismus durch Angabe der Umbenennung der Sorten und Funktionszeichen.

Definition 31

1. *Sei eine Signatur Σ gegeben. Eine endliche Abbildung σ_0 zwischen Zeichen der Form:*

$$[s_1 \mapsto q_1, \ldots, s_k \mapsto q_k,\ f_1 \mapsto g_1, \ldots, f_n \mapsto g_n]$$

wobei Sorten auf Sorten und Funktionszeichen auf Funktionszeichen (ohne Angabe der Funktionalität) abgebildet werden, induziert einen Signaturmorphismus σ von Σ in die durch die Umbenennung definierte Bildsignatur $\Sigma' = \sigma(\Sigma)$.

Sorten oder Funktionszeichen von Σ, die in der Zeichenabbildung nicht erwähnt werden, werden identisch nach Σ' abgebildet.

Die Umbenennung von Definitions- und Wertebereich der Funktionszeichen ergibt sich aus der Umbenennung der Sorten.

2. *Dies induziert auch eine Möglichkeit der Umbenennung von Spezifikationen: Die Umbenennung*

SP **with** σ

ist definiert als

$$\begin{aligned} \mathrm{Sig}(SP \textbf{ with } \sigma) &= \sigma(\mathrm{Sig}(SP)) \\ \mathrm{Mod}(SP \textbf{ with } \sigma) &= \{\, B \in \mathrm{Alg}(\sigma(\mathrm{Sig}(SP))) \mid B|_\sigma \in \mathrm{Mod}(SP) \,\} \end{aligned}$$

Bemerkung: Ist der durch σ_0 induzierte Signaturmorphismus σ injektiv und damit bzgl. der Ergebnissignatur $\sigma_0(\Sigma)$ bijektiv, dann ist die σ-Übersetzung für Σ-Strukturen wohldefiniert. Die Modellklasse der Umbenennung *SP* **with** σ ist dann gleich

$$\{\, \sigma(A) \in \mathrm{Alg}(\sigma(\Sigma)) \mid A \in \mathrm{Mod}(SP) \,\}$$

Beispiel 46: Umbenennung

Sei folgende Monoid-Signatur gegeben:

$$\mathrm{MON} = (\{\mathrm{Mon}\}, \{\, e : \mathrm{Mon}, \circ : \mathrm{Mon} \times \mathrm{Mon} \to \mathrm{Mon}\,\})$$

1. Dann definiert die Abbildung

   $$m_0 = [\mathrm{Mon} \mapsto \mathrm{Nat}, e \mapsto \mathrm{zero}, \circ \mapsto +]$$

   einen Signaturmorphismus m von MON in das Bild der Umbenennung

   $$\mathrm{NATMON} = (\{\mathrm{Nat}\}, \{\, \mathrm{zero} : \mathrm{Nat}, + : \mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Nat}\,\})$$

   Da m injektiv ist, bildet die m-Übersetzung jedes Modell eines Monoids (wie z.B. das Standardmodell der ganzen Zahlen mit 0 und Addition, Vektoren mit dem Nullvektor und der Vektoraddition oder den Raum der einstelligen Funktionen mit der Identitätsfunktion und der Funktionskomposition) auf die (bis auf Umbenennung gleiche) Struktur der Signatur NATMON ab.

2. Die Umbenennung

   $$\mathrm{MONOID}\ \textbf{with}\ [\mathrm{Mon} \mapsto \mathrm{Nat}, e \mapsto \mathrm{zero}, \circ \mapsto +]$$

   der Spezifikation

   spec MONOID =
   MON **then**
   vars $x, y, z : \mathrm{Mon}$
   axioms $x \circ e = x$
   $e \circ x = x$
   $x \circ (y \circ z) = (x \circ y) \circ z$
   end

   mit dem (durch die Umbenennung induzierten) Signaturmorphismus m beschreibt die Klasse der Monoidstrukturen mit Signatur NATMON, d.h. sie hat als Ergebnis die Klasse der Algebren

   $$\{\, A \in \mathrm{Alg}(\mathrm{NATMON}) \mid A|_m \in \mathrm{Mod}(\mathrm{MONOID})\,\}$$

   Dies ist gleich der Klasse der Algebren

   $$\{\, m(B) \in \mathrm{Alg}(\mathrm{NATMON}) \mid B \in \mathrm{Mod}(\mathrm{MONOID})\,\}$$

◀

Das Standardmodell der natürlichen Zahlen ist ein Modell dieser Spezifikation, aber auch alle Algebren der Signatur NATMON, deren Redukt bezüglich 0 und + ein Monoid über einer abzählbaren Menge ist, sind Modelle.

Ist die Zielsignatur echt umfangreicher als die durch die Umbenennung induzierte Signatur oder ist der Signaturmorphismus nicht injektiv, muss man mit dem Redukt arbeiten. Insbesondere benötigt man die Reduktbildung zur Erklärung von Sichten. Man spricht von einer Sicht auf eine Spezifikation, wenn die Sicht von gewissen Symbolen und Eigenschaften abstrahiert. Z.B. kann man die natürlichen Zahlen als Monoide auffassen: Monoide sind eine Sicht auf die die natürlichen Zahlen.

Zunächst definieren wir den Begriff des Theoriemorphismus.

Ein *Theoriemorphismus* $\alpha : SP_1 \to SP_2$ ist ein Signaturmorphismus so, dass SP_2 alle Axiome der mit α umbenannten Spezifikation SP_1 erfüllt.

Formal definieren wir:

Definition 32 (Theoriemorphismus, Sicht)

1. *Ein* Theoriemorphismus $\alpha : SP_1 \to SP_2$ *ist ein Signaturmorphismus* $\alpha : \mathrm{Sig}(SP_1) \to \mathrm{Sig}(SP_2)$*, so dass für jedes Modell* $M \in \mathrm{Mod}(SP_2)$ *gilt*

   $(*) \qquad M|_\alpha \in \mathrm{Mod}(SP_1)$

2. *Sei* α_0 *eine endliche Abbildung zwischen Zeichen.*

   Ist der durch α_0 *induzierte Signaturmorphismus* $\alpha : \mathrm{Sig}(SP_1) \to \mathrm{Sig}(SP_2)$ *wohldefiniert und ein Theoriemorphismus, dann definiert dies eine* Sicht*, die durch*

   view $SM : SP_1$ **to** $SP_2 = \alpha_0$

   spezifiziert und mit dem Namen SM versehen wird.

   Die **Semantik** *von SM ist der Theoriemorphismus* α*.*

Bemerkung: Die Bedingung $(*)$ ist äquivalent dazu, dass SP_2 die Axiome der umbenannten Spezifikation

$SP1$ **with** α

erfüllt, d.h.

$\mathrm{Mod}(SP_2) \subseteq \mathrm{Mod}(SP_1$ **with** $\alpha)$

Beispiel 47:

1. **σ-Redukt:**

   Die folgende Abbildung ist ein Signaturmorphismus von der Monoidsignatur in die Signatur der natürlichen Zahlen:

   $\mu : \mathrm{Sig}(\mathrm{MONOID}) \to \mathrm{Sig}(\mathrm{NAT})$
   $\mu_{sort}(\mathrm{Mon}) = \mathrm{Nat}$
   $\mu_{op}(e) = \mathrm{zero}$
   $\mu_{op}(\circ) = +$

   Axiome für Monoide:

   - $\circ$ ist assoziativ: $x \circ (y \circ z) = (x \circ y) \circ z$.
   - e ist neutrales Element.

   Durch Reduktbildung erhält man z.B aus dem Standardmodell $N = \langle \mathbb{N}, 0, \_+1, \_+\_, \_*\_, \ldots \rangle$ der natürlichen Zahlen eine Monoidstruktur $N|_\mu$, das μ-Redukt von N:

   $(N|_\mu)_{\mathrm{Mon}} = \mathbb{N}$
   $e^{N|_\mu} = 0, (\_\circ\_)^{N|_\mu} = \_+\_$

2. **Sicht:**

   μ ist auch ein Theoriemorphismus, da NAT bzgl. $0, +$ einen Monoid bildet. Also definiert μ eine Sicht von NAT als MONOID:

   view NAT_as_MONOID : MONOID **to** NAT $= \mu$

◀

4.6 Parametrisierung von Spezifikationen

Parametrisierung ist ein Abstraktionsprozess, bei dem Module von Software zusammengefasst werden. Dabei wird von Namen abstrahiert, indem diese in einem anderen Zusammenhang durch verschiedene Parameter ersetzt werden. Unter dem Aspekt der Spezifikation werden solche Spezifikationen betrachtet, die eine andere Spezifikation als Parameter erhalten. Ein einfaches Beispiel ist die Spezifikation von Mengen. Man kann Mengen spezifizieren, ohne die Elemente einer Menge näher zu bestimmen. Die Spezifikation der Mengenelemente wird im konkreten Fall, wenn eine bestimmte Menge über bestimmten Elementen verwendet werden soll, als Parameter angegeben. In der Praxis ergibt sich durch die Parametrisierung also eine Verringerung des Aufwandes bei der Spezifikation, da man nicht bei jeder Menge über verschiedenen Elementen die Menge an sich neu spezifizieren muß, sondern die mit Parameter versehene, eventuell sogar schon bewiesene Spezifikation von Mengen benutzen kann.

Definition 33 *Eine* parametrisierte Spezifikation *(oder* generische Spezifikation*) hat die Form*

spec $SN[SP]$ =
Body
end

wobei

- *SN der Name der parametrisierten Spezifikation,*
- *SP der Name der formalen Parameterspezifikation und*
- *Body der Rumpf der Spezifikation ist.*

SN ist wohldefiniert, wenn der Rumpf den Parameter erweitert, d.h. wenn

SP **then** *Body*

eine wohldefinierte Spezifikation ist (in der durch die Spezifikationsdeklarationen gegebenen Umgebung). („Der Rumpf der parametrisierten Spezifikation ist eine Erweiterung des formalen Parameters.")

Beispiel 48: parametrisierte Spezifikationen

1. **Parametrisierte Listen**

   Gegeben sei folgende triviale Spezifikation mit einer Sorte:

   spec ELEM =
   sorts Elem
   end

   Die Spezifikation LIST[ELEM] erweitert die Spezifikation ELEM um endliche Listen von Elementen.

   spec LIST[ELEM] =
   free type List[Elem] ::= nil | cons(first : ?Elem; rest : ?List[Elem])
   ops _ ++ _ : List[Elem] × List[Elem] → List[Elem]
   vars e : Elem; l, l' : List[Elem]

axioms $\text{nil} \mathbin{+\!\!+} l = l$
$\text{cons}(e,l) \mathbin{+\!\!+} l' = \text{cons}(e, l \mathbin{+\!\!+} l')$
ops $\text{reverse} : \text{List[Elem]} \to \text{List[Elem]}$
axioms $\text{reverse}(\text{nil}) = \text{nil}$
$\text{reverse}(\text{cons}(e,l)) = \text{reverse}(l) \mathbin{+\!\!+} \text{cons}(e, \text{nil})$

end

2. **„Lose“ endliche Mengen**

Gegeben sei folgende Spezifikation einer Sorte mit einer Gleichheitsoperation:

spec ELEM_EQ =
BOOL **then**
sorts Elem
ops $\text{eq} : \text{Elem} \times \text{Elem} \to \text{Bool}$
vars $x, y, z : \text{Elem}$
axioms $\text{eq}(x,x) = \text{true}$
$\text{eq}(x,y) = \text{eq}(y,x)$
$\text{eq}(x,y) = \text{true} \wedge \text{eq}(y,z) = \text{true} \Rightarrow \text{eq}(x,z) = \text{true}$

end

Die generische Spezifikation loser Mengen ist:

spec LSET[ELEM_EQ] =
generated type Set[Elem] ::= empty | add(Elem; Set[Elem])
ops $\text{iselem} : \text{Elem} \times \text{Set[Elem]} \to \text{Bool}$
vars $x, y : \text{Elem};\ s : \text{Set[Elem]}$
axioms $\text{iselem}(x, \text{empty}) = \text{false}$
$\text{iselem}(x, \text{add}(y,s)) = \text{or}(\text{eq}(x,y), \text{iselem}(x,s))$

end

Lose Mengen natürlicher Zahlen können spezifiziert werden durch

LSET[NAT **fit** Elem $\mapsto$ Nat]

Da [Elem $\mapsto$ Nat] einen Theoriemorphismus von ELEM_EQ nach NAT induziert, ist diese Parameterübergabe korrekt. ◀

Intuitiv kann eine parametrisierte Spezifikation wie LIST[ELEM] als eine Funktion aufgefasst werden, die jedes Modell M von ELEM zu einem Modell $list(M)$ von LIST[ELEM] erweitert, wobei

$$\begin{aligned} list(M)_{Elem} &= M_{Elem} \\ list(M)_{List[Elem]} &= \{\, \langle m_1, \ldots, m_k \rangle \mid m_i \in M_{\text{Elem}}, k \geq 0 \,\} \end{aligned}$$

In jedem Modell M von LSET[ELEM_EQ] gibt es Elemente der Form

$$\text{empty}^M,\ \text{add}(x_0, \text{empty})^M,\ \text{add}(x_0, \text{add}(x_1, \ldots, \text{add}(x_n, \text{empty}) \ldots))^M$$

Beispiel 49: Standardmodelle von LSET[NAT]

- $\mathcal{P}^{\text{fin}}(\mathbb{N})$ endliche Mengen von natürlichen Zahlen

$$\begin{aligned} \mathcal{P}^{\text{fin}}(\mathbb{N})_{\text{Nat}} &= \mathbb{N} \\ \mathcal{P}^{\text{fin}}(\mathbb{N})_{\text{Set[Nat]}} &= \{\{a_1, \ldots, a_n\} \mid n \geq 0, a_i \in \mathbb{N}, i = 1, \ldots, n\} \end{aligned}$$

- $\mathbb{N}^*$ endliche Folgen von natürlichen Zahlen

$$\begin{aligned} \mathbb{N}^*_{\text{Nat}} &= \mathbb{N} \\ \mathbb{N}^*_{\text{Set[Nat]}} &= \{\langle a_1,\ldots,a_n\rangle \mid n \geq 0, a_i \in \mathbb{N}, i = 1,\ldots,n\} \end{aligned}$$

  $\mathbb{N}^*$ ist Modell, da

  1. $\mathbb{N}^*_{\text{Set[Nat]}}$ ist Σ-erreichbar (bzgl. empty, add), denn jedes Element $\langle a_1,\ldots,a_n\rangle$ interpretiert $\text{add}(x_1,\ldots\text{add}(x_n,\text{empty})\ldots)$ mit $\nu(x_i) = a_i, i = 1,\ldots,n$.
  2. $\mathbb{N}^*$ erfüllt die Axiome von $LSET[NAT]$, d.h. die Axiome für iselem.

- Anderes Modell U von LSET[Elem]

$$\begin{aligned} U_{\text{Nat}} &= \mathbb{N} \\ U_{\text{Set[Nat]}} &= \{P \subseteq \mathbb{N} \mid \text{es gibt } a_1,\ldots\ a_n \in \mathbb{N} : P = \mathbb{N} \setminus \{a_1,\ldots,a_n\}\} \\ \text{empty}^U &= \mathbb{N} \\ \text{add}^U(m,P) &= P \setminus \{m\} \\ \text{iselem}^U(m,P) &= (m \notin P) \end{aligned}$$

◀

Zur *Parameterübergabe* muss der formale Parameter mit dem aktuellen Parameter in Beziehung gebracht werden:

Die Signatur des formalen Parameters muss in die Signatur des aktuellen Parameters umbenannt werden und der aktuelle Parameter muss die Anforderungen des formalen Parameters erfüllen, d.h. es muss einen Theoriemorphismus von dem formalen Parameter auf den aktuellen Parameter geben.

Z.B. muss zur Instanzierung von ELEM mit einer Spezifikation der natürlichen Zahlen die Sorte Elem in Nat umbenannt werden.

Ein aktueller Parameter von LSET muss eine Sicht auf eine Gleichheitstheorie besitzen.

Definition 34 *Sei*

spec $SN[SP] = Body$ **end**

eine generische Spezifikation mit formalem Parameter SP, sei AP ein aktueller Parameter und

$SM : SP$ **to** $AP = \sigma_0$

ein Theoriemorphismus von SP nach AP (d.h. SM ist eine Sicht auf AP), dann bezeichnen

$SN[$**view** $SM]$ *oder* $SN[AP$ **fit** $\sigma_0]$

die Instanzierung von *SN* mit *AP*. *Man schreibt einfach*

$SN[AP]$

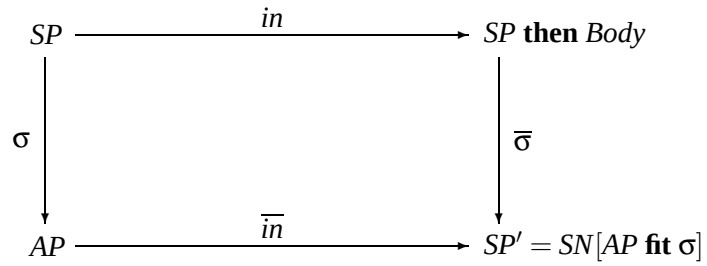
wenn SM eine identische Einbettung ist.

Die Semantik der Instanzierung ist gegeben durch

$(SP$ **then** $Body)$ **with** σ_0 **and** AP

Aus technischen Gründen muss jedes Symbol, das sowohl in Body wie in AP vorkommt schon im formalen Parameter SP vorkommen.

Bemerkung: Man kann die Instanzierung kategorientheoretisch durch folgenden Push-out erklären:



d.h. die Instanzierung SP' ist die „kleinste“ kompatible Erweiterung von AP und $(SP$ **then** $Body)$ bzgl. in, σ, die

- Namen von SP mit σ nach $SN[AP$ **fit** $\sigma]$ abbildet und
- Namen aus $(Body \cup AP) \setminus SP$ identisch, aber disjunkt vereinigt in $SN[AP$ **fit** $\sigma]$ überführt.

Man nennt SP' auch eine „amalgamierte Summe“, das heißt, dass SP' die disjunkte Vereinigung von AP und $Body$ ist, bis auf die (umbenannten) Elemente von SP, die gemeinsam benutzt werden.

Betrachte etwa die generische Spezifikation

spec LIST[ELEM] = **free type** List[Elem] ::= ... **end**

Die Semantik des Spezifikationsausdrucks

LIST[NAT **fit** Elem $\mapsto$ Nat]

ist die Spezifikation

(ELEM **then free type** List[Elem] ::= ...) **with** [Elem $\mapsto$ Nat] **and** NAT

$\equiv$ (**sorts** Nat; **free type** List[Nat] ::= ...) **and** NAT

Da die Beziehung zwischen formalen und aktuellen Parametern ein Theoriemorphismus ist, erfüllt der aktuelle Parameter alle Eigenschaften des formalen Parameters.

Man könnte in der hier vorgestellten vereinfachten Parametrisierung auch definieren:

$SP[AP$ **fit** $\sigma_0] \equiv AP$ **then** $(Body$ **with** $\sigma_0)$

Beispiel 50:

1. Die Instanzierung von LIST[ELEM] mit natürlichen Zahlen wird durch Umbenennung von Elem in Nat definiert:

   LIST[NAT **fit** [Elem $\mapsto$ Nat]]

2. Zur Instanzierung von LSET mit NAT muss zunächst nachgewiesen werden, dass der durch [Elem $\mapsto$ Nat] induzierte Signaturmorphismus ein Theoriemorphismus ist.

   Dann definiert man die Sicht

   view NAT_as_ELEM_EQ : NAT **to** ELEM_EQ = [Elem $\mapsto$ Nat]

   und erhält damit

   LSET[**view** NAT_as_ELEM_EQ]

   als Spezifikation für lose endliche Mengen natürlicher Zahlen. Äquivalent dazu kann man schreiben:

   LSET[NAT **fit** [Elem $\mapsto$ Nat]]

◀

Beispiel 51: Mengen durch Listen

Die Listenspezifikation hat eine andere Signatur als die Spezifikation LSET. Um Listen als Mengen zu sehen, erweitern wir die Signatur der Listen um die Operationen der Mengen:

spec LIST1 =
LIST **then**
ops empty : List
$\{\_\}$: Nat $\rightarrow$ List
add : Nat $\times$ List $\rightarrow$ List
$\_\cup\_$: List $\times$ List $\rightarrow$ List
...
end

Durch Umbenennen $[\text{List} \mapsto \text{Set}]$ erhält man eine Signatur, die die Mengensignatur umfasst. Mit **hide** können die Listenoperationen, die nicht in der Schnittstelle von LSET vorkommen, versteckt werden.

Damit erhalten wir die gleiche Schnittstelle und eine „kleinere“ Modellklasse als LSET, also eine *Verfeinerung* von LSET. ◀

4.7 Zusammenfassung

- Eine CASL Spezifikation beschreibt abstrakt eine Klasse von (Rechen-)Strukturen durch Angabe einer Signatur, bestehend aus
  - Sorten für die Objektmengen
  - Funktionssymbolen für die Funktionen über Objektmengen
  - Prädikatensymbolen für die Relationen zwischen Objekten

  und einer endlichen Menge von charakteristischen Eigenschaften (prädikatenlogischen Formeln 1. Stufe)
- In CASL können „lose“ und „freie“ Datentypkonstruktoren durch **generated** und **free** explizit angegeben werden. Datentypdeklarationen erlauben eine kompakte Notation der Konstruktorenmengen und der zugehörigen Selektoren.
- Spezifikationen können strukturiert aufgebaut werden. Dazu bietet CASL folgende Möglichkeiten: Bildung der
  - Summe SP_1 **and** SP_2 zweier Spezifikationen,
  - Erweiterung SP **then** S, F, E einer Spezifikation mit neuen Sorten S, Funktions- und Prädikatensymbolen F, P und Axiomen E,
  - Umbenennung SP **with** $[s \mapsto s', \ldots, f \mapsto f', \ldots]$ einer Spezifikation.

  Außerdem erlaubt CASL
  - die Kapselung einer Menge SY von Symbolen: SP **hide** SY und
  - die Parametrisierung von Spezifikationen.
- Umbenennung geschieht mit Hilfe von Signaturmorphismen. Ein Signaturmorphismus ist eine Abbildung der Sorten, Funktions- und Prädikatensymbole einer Signatur auf Symbole gleicher Art, so dass die Umbenennung der Funktions- und Prädikatensymbole verträglich mit der Sortenumbenennung ist.

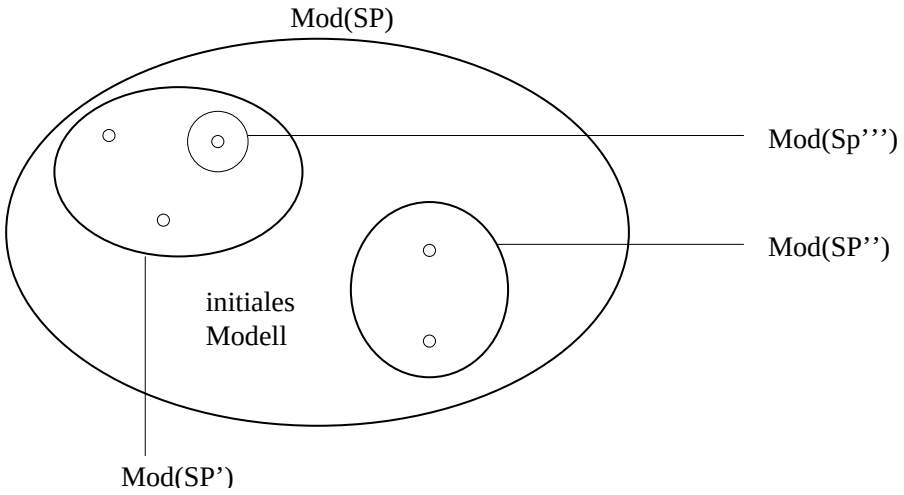


Abbildung 6: Verfeinerungsbeziehungen: $SP \rightsquigarrow SP' \rightsquigarrow SP'''$, $SP \rightsquigarrow SP''$.

- Eine parametrisierte Spezifikation **spec** $SN[SP] = Body$ kann als eine Funktion aufgefasst werden, die jedes Modell der Parameterspezifikation SP zu einem Modell von SP **then** $Body$ erweitert.

- Die Parameterübergabe verlangt die Angabe eines Theoriemorphismus vom formalen Parameter zum aktuellen Parameter. Ein Theoriemorphismus ist ein Signaturmorphismus, der die Eigenschaften des formalen Parameters erhält, d.h. der aktuelle Parameter muss (modulo Umbennenung) alle vom formalen Parameter geforderten Eigenschaften erfüllen. Bei der Parmeterübergabe werden die Symbole des formalen Parameters in die des aktuellen Parameters (gemäß dem Theoriemorphismus) umbenannt.

5 Verfeinerungstechniken

Programmentwicklung geht von den Anforderungen zur Implementierung, d.h. von der Beschreibung des Problems (das „was“) zur Beschreibung der Lösung (das „wie“). Da eine Problembeschreibung idealerweise mehrere Lösungen zulässt, werden bei der Entwicklung Entwurfsentscheidungen getroffen, die den Lösungsraum einschränken. In der Sprechweise der Spezifikationen heißt das, die Modellklasse einer Anforderungsspezifikation so weit einzuschränken, bis man eine ausführbare Spezifikation erhält, die nach Transformation oder Übersetzung auf dem Rechner ausführbar ist. In diesem Abschnitt betrachten wir grundlegende Techniken der **Verfeinerung** von Spezifikationen — und zwar

- Verfeinerung von funktionalen Anforderungen: funktionale Verfeinerung,

- Verfeinerung der Datenstruktur: Wechsel der Datenstruktur und

- Übergang von ausführbaren Spezifikationen zu funktionalen Programmen

5.1 Der Verfeinerungsbegriff: Verfeinerung durch Modellklassen-Inklusionen

Definition 35 *SP' ist eine* Verfeinerung *von SP (geschrieben $SP \rightsquigarrow SP'$), falls* $\mathrm{Sig}(SP') = \mathrm{Sig}(SP)$ *und* $\mathrm{Mod}(SP') \subseteq \mathrm{Mod}(SP)$.

Bemerkung:

1. Intuitiv heißt dies, dass die Signaturen von SP und SP' gleich sind und SP' mehr Eigenschaften erfüllt als SP. Es wird nicht verlangt, dass SP' erfüllbar ist (d.h. mindestens ein Modell besitzt), da in einer Programmentwicklung der Art $SP_n \rightsquigarrow SP_{n-1} \rightsquigarrow \ldots \rightsquigarrow SP_0$, die in einem Programm endet, das durch eine Spezifikation SP_0 repräsentiert wird, automatisch gilt:

   $$|\operatorname{Mod}(SP_0)| = 1 \quad \text{(bis auf Isomorphie)}$$

   und daher

   $$\operatorname{Mod}(SP_i) \neq \emptyset \quad \text{für } i = 0, \ldots, n$$

2. Idealerweise besteht eine Programmentwicklung aus einer Kette von Verfeinerungen:

   $$SP_n \rightsquigarrow SP_{n-1} \rightsquigarrow \ldots \rightsquigarrow SP_0$$

   wobei SP_0 eine ausführbare Spezifikation ist.

   (Die Form der Axiome einer ausführbaren Spezifikation hängt von der gewählten Implementierungssprache ab. In CASL sind es bedingte Gleichungen; bei funktionalen Sprachen wie SML sind es rekursive oder induktive Gleichungen.)

Lemma 36

1. *$\rightsquigarrow$ ist eine partielle Ordnung, d.h. $\rightsquigarrow$ ist reflexiv, transitiv und antisymmetrisch.*
2. *Alle spezifikationsbildenden Operationen (und damit insbesondere Summe, Erweiterung, Umbenennung, Kapselung) sind monoton bzgl. $\rightsquigarrow$. Zum Beispiel gilt also*

   $$SP \rightsquigarrow SP_1 \quad \Rightarrow \quad SP \textbf{ hide } SY \rightsquigarrow SP_1 \textbf{ hide } SY$$

3. *Für jeden mit spezifikationsbildenden Operationen (wie Summe, Erweiterung, Umbenennung, Kapselung) gebildeten Spezifikationsausdruck $C[SP]$ gilt*

   $$SP \rightsquigarrow SP_1 \quad \Rightarrow \quad C[SP] \rightsquigarrow C[SP_1]$$

   Das heißt, jede lokale Verfeinerung ist global.

Methodisch gibt es zwei einfache Techniken, um Spezifikationen zu verfeinern. Bei der axiomatischen Erweiterung reichert man eine Spezifikation um neue Sorten, Funktionssymbole und Axiome an. Bei der axiomatischen Verfeinerung können auch „alte“ Axiome durch neue ersetzt werden; die Gültigkeit der alten Axiome muss dann aber nachgewiesen werden.

Definition 37

1. *(Σ_1, E_1) heißt* axiomatische Erweiterung *von (Σ, E) falls $\Sigma \subseteq \Sigma_1$ und $E \subseteq E_1$. Es gilt*

   $$(\Sigma, E) \rightsquigarrow (\Sigma_1, E_1) \textbf{ hide } (\Sigma_1 \setminus \Sigma)$$

   Man spricht von axiomatischer Anreicherung, *falls nur neue Axiome hinzugefügt werden.*

2. (Σ_1, E_1) *heißt* axiomatische Verfeinerung *von* (Σ, E) *falls* $\Sigma \subseteq \Sigma_1$ *und* $E_1 \vdash E$. *Es gilt*

$$(\Sigma, E) \rightsquigarrow (\Sigma_1, E1) \textbf{ hide } (\Sigma_1 \setminus \Sigma)$$

Man spricht von **axiomatischer Verfeinerung modulo Umbenennung**, falls die Namen in (Σ_1, E_1) angepasst werden müssen, d.h. wenn

$$\sigma : (\Sigma, E) \to (\Sigma_1, E_1)$$

ein Theoriemorphismus ist.

Die axiomatische Verfeinerung modulo Umbenennung ist ein Spezialfall für den Wechsel einer Datenstruktur (siehe Abschnitt 5.2).

Beispiel 52: Größter gemeinsamer Teiler

spec NATDIV =
NAT **then**
ops $\_$ div $\_$, $\_$ mod $\_$: Nat × Nat → Nat
preds $\_ \mid \_$: Nat × Nat %%{ist Teiler von}%
vars x, y : Nat
axioms $x \mid y \equiv \exists z : \text{Nat}.\ x * z = y$
$y > 0 \Rightarrow x = (x \text{ div } y) * y + (x \text{ mod } y)$
$y > 0 \Rightarrow (x \text{ mod } y) < y$
end

spec NATGCD =
NATDIV **then**
ops gcd : Nat × Nat → Nat
vars x, y, z : Nat
axioms $\text{gcd}(x,y) \mid x$
$\text{gcd}(x,y) \mid y$
$z \mid x \wedge z \mid y \Rightarrow z \mid \text{gcd}(x,y)$
end

spec NATEuclid =
NATDIV **then**
ops gcd : Nat × Nat → Nat
vars x, y : Nat
axioms $y = 0 \Rightarrow \text{gcd}(x,y) = x$
$y > 0 \wedge x \leq y \Rightarrow \text{gcd}(x,y) = \text{gcd}(x, y-x)$
$y > 0 \wedge y < x \Rightarrow \text{gcd}(x,y) = \text{gcd}(y,x)$
end

NATEuclid ist eine axiomatische Verfeinerung von NATGCD, denn:

$$Ax_{\text{NATEuclid}} \models Ax_{\text{NATGCD}}$$

Insbesondere wird NATGCD durch NATEuclid verfeinert: NATGCD $\rightsquigarrow$ NATEuclid. ◄

Die folgenden Beispiele erläutern die beiden Grundtechniken:

Beispiel 53: Axiomatische Anreicherung und Verfeinerung

1. Axiomatische Anreicherung: Hinzunahme von Axiomen

spec LSETNAT =
BOOL **and** NAT **then**
sorts Set
ops empty : Set
$\{\_\}$: Nat $\rightarrow$ Set
add : Nat $\times$ Set $\rightarrow$ Set
$\_\cup\_$: Set $\times$ Set $\rightarrow$ Set
$\_$ aus $\_$: Nat $\times$ Set $\rightarrow$ Bool
vars x, y : Nat, s_1, s_2 : Set
axioms add$(x,s) = s \cup \{x\}$
x aus empty $=$ false
x aus $\{y\} \equiv (x = y)$
x aus $(s_1 \cup s_2) = (x$ aus $s_1)$or$(x$ aus $s_2)$
end

spec SETNAT =
LSETNAT **then**
vars s_1, s_2, s_3 : Set
axioms $s \cup s = s$
$s_1 \cup s_2 = s_2 \cup s_1$
$s_1 \cup (s_2 \cup s_3) = (s_1 \cup s_2) \cup s_3$
end

Offensichtlich gilt LSETNAT $\rightsquigarrow$ SETNAT.

SETNAT ist eine axiomatische Anreicherung von LSETNAT.

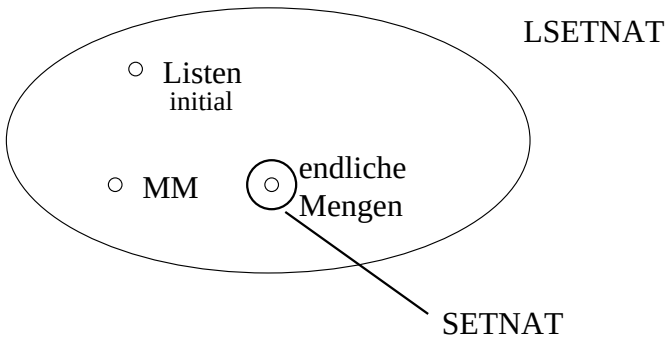


Abbildung 7: Verfeinerung von LSETNAT durch SETNAT.

2. Axiomatische Anreicherung: LSTACK zu STACK.

spec LSTACK =
sorts Stack, Data;

ops empty : Stack
push : Data × Stack → Stack
pop : Stack →? Stack
top : Stack →? Data
vars x : Data; s : Stack
axioms top(push$(x,s)) = x$
pop(push$(x,s)) = s$
end

STACK ist eine axiomatische Anreicherung von LSTACK um die Datentypdeklaration

free type Stack ::= empty | push(Data; Stack)

und die Axiome

¬ def pop(empty)
¬ def top(empty)

Es gilt: LSTACK ⇝ STACK.

3. Axiomatische Verfeinerung modulo Umbenennung: Verfeinerung „loser“ Mengen durch Folgen

Die folgende Spezifikation von Listen natürlicher Zahlen sei gegeben.

spec LISTNAT =
NAT %%{Spez. von nat. Zahlen mit Boolescher Gleichheit eq}% **then**
free type List ::= nil | cons(Elem; List)
ops first : List →? Elem
rest : List →? List
_ ++ _ : List × List → List
vars x : Elem; x_l, y_l : List
axioms ¬ def first(nil)
first(cons$(x,x_l)) = x$
¬ def rest(nil)
rest(cons$(x,x_l)) = x_l$
nil ++ $y_l = y_l$
cons(x,x_l) ++ y_l = cons$(x, x_l$ ++ $y_l)$
end

Dann definiert man eine Erweiterung von LISTNAT um die Mengenfunktionen.

spec SET_by_LIST =
LISTNAT **then**
ops empty : List
{_} : Nat → List
add : Nat × List → List
_ ∪ _ : List × List → List
_ aus _ : Nat × List → Bool
vars x, y : Nat; s, s_1, s_2 : List

axioms %%{Explizite Definition der Mengenoperationen empty, {_}, add, ∪}%
empty = nil
{x} = cons(x, nil)
add(x, s) = cons(x, s)
$s_1 \cup s_2 = s_1$ ++ s_2
%%{Induktive Definition der Mengenoperation _ aus _}%
x aus nil = false
x aus cons(y, s) = (eq(x, y)or(x aus s))

end

spec SET_by_LIST1 =
(SET_by_LIST **with** [List ↦ Set])
hide nil, cons, first, rest, ++
end

In dieser Implementierung wird die Mengenvereinigung durch Konkatenation dargestellt, d.h. Einfügen in eine Menge geschieht in konstanter Zeit. Suche erfordert lineare Zeit: alle Elemente der Menge s müssen durchgesehen werden, um festzustellen, dass x nicht in s ist. Es gilt

LSETNAT ⇝ SET_by_LIST1

◀

5.2 Wechsel der Datenstruktur

Jede Programmiersprache stellt Konstrukte zur Beschreibung von konkreten Datenstrukturen zur Verfügung, durch die natürlich nicht alle abstrakt definierten Daten direkt modelliert werden können. Außerdem ist es aus Effizienzgründen häufig notwendig, bestimmte Datenstrukturen zu verwenden. Deshalb besteht eine wichtige Aufgabe bei der Programmentwicklung darin, Methoden zum korrekten Wechsel der Datenstruktur bereitzustellen.

„Simulation“ einer Σ-Struktur durch eine Σ′-Struktur

Intuitiv wird eine (S,F)-Struktur A durch eine (S_1,F_1)-Struktur B simuliert, wenn jede Trägermenge A_s von A durch eine Teilmenge

$$Rep_s \subseteq B_{s'}$$

einer Trägermenge $B_{s'}$ von B repräsentiert wird, und wenn jedes Funktionssymbol $f \in F$ durch ein Funktionssymbol $f_1 \in F_1$ repräsentiert wird.

Dabei können mehrere Elemente von Rep_s das gleiche Element von A_s repräsentieren, d.h. es existiert eine Äquivalenzrelation $\sim_s$ auf Rep_s mit $b_1 \sim_s b_2$ gdw. b_1 und b_2 das gleiche Element repräsentieren.

Natürlich muss $\sim_s$ mit den Operationen von A_s verträglich sein, d.h.

$$b_1 \in Rep_s, b_2 \in Rep_s \text{ und } b_1 \sim_s b_2 \text{ impliziert } f_1^B(b_1) \sim_s f_1^B(b_2) \in Rep_s$$

für alle Repräsentanten f_1 von Funktionssymbolen $f \in F$.

Rep muss abgeschlossen sein unter den Operationen von F, d.h. für $f : s \to s'$ muss für alle $b \in Rep_s$ gelten:

$$f_1(b) \in Rep_{s'}$$

Formal definieren wir :

Definition 38 (Simulation)

1. *Sei* $\Sigma \subseteq \Sigma_1$.

   Eine Σ_1*-Struktur B* simuliert identisch *eine* Σ*-Struktur A bzgl.* $Rep^B, \sim^B$*, falls*

   (a) $Rep^B_s \subseteq B_s$ *für alle* $s \in S$,

   (b) $\sim^B_s$ *ist eine* Σ*-Kongruenz auf* Rep^B_s *für alle* $s \in S$, *und*

   (c) *A ist isomorph zu* $Rep^B/\sim^B$ *wobei* $Rep^B/\sim^B =_{\text{def}} ((Rep^B_s)/\sim^B_s)_{s\in S}$.

2. *Eine* Σ_1*-Struktur B* simuliert eine Σ-Struktur *A* bzgl. Umbenennung $\rho : \Sigma \to \Sigma_1$, Rep^B *und* $\sim^B$ *falls*

   (a) $Rep^B_s \subseteq B_{\rho(s)}$ *für alle* $s \in S$,

   (b) $\sim^B_s$ *ist eine* $\rho(\Sigma)$*-Kongruenz auf* Rep^B_s *für alle* $s \in S$, *und*

   (c) $A \cong Rep^B/\sim^B$.

   Jede identische Simulation ist also eine Simulation.

Beispiel 54: Listen simulieren Mengen

Wir definieren die Umbenennung

$$\rho : \text{Sig}(\text{SETNAT}) \to \text{Sig}(\text{SET\_by\_LIST})$$

wie vorher als $[\text{Set} \mapsto \text{List}]$, d.h. ρ ist folgendermaßen definiert:

$$\rho(\text{Set}) = \text{List} \quad \text{und} \quad \rho(x) = x, \text{ sonst}$$

Die Struktur $\mathbb{N}^*$ der endlichen Listen simuliert auf folgende verschiedene Arten die Struktur der endlichen Mengen $\mathcal{P}^{\text{fin}}(\mathbb{N})$ über natürlichen Zahlen bzgl. der Umbenennung ρ.

1. Simulation der Mengen durch die Struktur *U* der ungeordneten Listen.

$$\begin{array}{rcl}
Rep^U_{\text{Set}} & = & N^* \\
Rep^U_{\text{Nat}} & = & \mathbb{N} \\
Rep^U_{\text{Bool}} & = & \{\text{T},\text{F}\} \\
\text{empty}^U & = & \varepsilon \\
x \text{ aus}^U \langle x_1,\ldots,x_n\rangle & \Leftrightarrow & x = x_i \text{ für ein } i \in \{1,\ldots,n\} \\
\{x\}^U & = & \langle x\rangle \\
\text{add}^U(x,\langle x_1,\ldots,x_n\rangle) & = & \langle x,x_1,\ldots,x_n\rangle \\
\langle x_1,\ldots,x_n\rangle \cup^U \langle y_1,\ldots,y_m\rangle & = & \langle x_1,\ldots,x_n,y_1,\ldots,y_m\rangle \\
\langle x_1,\ldots,x_n\rangle \sim^U \langle y_1,\ldots,y_m\rangle & \Leftrightarrow & \{x_1,\ldots,x_n\} = \{y_1,\ldots,y_m\}, \\
 & & \text{d.h. die beiden Folgen haben die gleichen Elemente}
\end{array}$$

2. Simulation der Mengen durch die Struktur G der geordneten Listen.

$$\begin{array}{rcl}
Rep^G_{\mathrm{Set}} & = & \{\langle x_1,\ldots,x_n\rangle \mid x_1 < \ldots < x_n, n \geq 0\} \\
\mathrm{empty}^G & = & \varepsilon \\
x \text{ aus}^G \langle x_1,\ldots,x_n\rangle & \Leftrightarrow & x = x_i \text{ für ein } i \in \{1,\ldots,n\} \\
\{x\}^G & = & \langle x\rangle \\
\mathrm{add}^G(x,\langle x_1,\ldots,x_n\rangle) & = & \begin{cases} \langle x_1,\ldots,x_i,x,x_{i+1},\ldots,x_n\rangle & \text{falls } x_i < x < x_{i+1} \\ \langle x_1,\ldots,x_n\rangle & \text{falls } x = x_i \text{ für ein } i \end{cases} \\
\langle x_1,\ldots,x_n\rangle \cup^G \langle y_1,\ldots,y_m\rangle & = & \langle z_1,\ldots,z_k\rangle \in Rep^G_{\mathrm{Set}}, \\
 & & \text{wobei } \{x_1,\ldots,x_n,y_1,\ldots,y_m\} = \{z_1,\ldots,z_k\} \\
s_1 \sim^G s_2 & \Leftrightarrow & s_1 = s_2
\end{array}$$

3. Simulation der Mengen durch die Struktur SG der schwach geordneten Listen

$$\begin{array}{rcl}
Rep^{SG}_{\mathrm{Set}} & = & \{\langle x_1,\ldots,x_n\rangle \mid x_1 \leq \ldots \leq x_n, n \geq 0\} \\
\mathrm{empty}^{SG} & = & \varepsilon \\
\{x\}^{SG} & = & \langle x\rangle \\
x \text{ aus}^{SG} \langle x_1,\ldots,x_n\rangle & \Leftrightarrow & x = x_i \text{ für ein } i \in \{1,\ldots,n\} \\
\mathrm{add}^{SG}(x,\langle x_1,\ldots,x_n\rangle) & = & \langle x_1,\ldots,x_i,x,x_i+1,\ldots,x_n\rangle \text{ falls } x_i \leq x \leq x_{i+1} \\
\langle x_1,\ldots,x_n\rangle \cup^{SG} \langle y_1,\ldots,y_m\rangle & = & \langle z_1,\ldots,z_k\rangle \in Rep^{SG}_{\mathrm{Set}}, \text{ wobei } \langle z_1,\ldots,z_k\rangle \\
 & & \text{eine Permutation von } \langle x_1,\ldots,x_n\rangle \mathbin{++} \langle y_1,\ldots,y_m\rangle \text{ ist} \\
\langle x_1,\ldots,x_n\rangle \sim^{SG} \langle y_1,\ldots,y_m\rangle & \Leftrightarrow & \{x_1,\ldots,x_n\} = \{y_1,\ldots,y_m\}, \\
 & & \text{d.h. die beiden Folgen haben die gleichen Elemente}
\end{array}$$

Bei U wird ein Quotient von $\mathbb{N}^*$ gebildet, G verwendet eine Teilmenge von $\mathbb{N}^*$, SG kombiniert Teilmengen und Quotientenbildung. ◀

Da die Signatur der simulierenden Struktur i.a. noch weitere Symbole enthält (in Beispiel 54 die Grundoperationen auf Listen zusätzlich zu den Mengenoperationen), entspricht die Konstruktion folgenden Schritten:

1. Forget: Vergessen aller Symbole, die nicht aus $\rho(\Sigma)$ stammen
2. Restrict: Einschränkung auf die Repräsentantenmengen Rep_s
3. Identify: Quotientenbildung bzgl. $\sim_s$.

Man spricht deshalb auch von **„Forget-Restrict-Identify"-Verfeinerung**.

Definition 39 *Eine Spezifikation SP_1* FRI-implementiert *eine Spezifikation SP bzgl. eines Signaturmorphismus $\sigma : \mathrm{Sig}(SP) \to \mathrm{Sig}(SP_1)$ (geschrieben $SP_1 \rightsquigarrow_\sigma SP$), falls jedes Modell B von SP_1 ein Modell von SP bzgl. passender Rep^B und $\sim^B$ simuliert.*

Satz 40 *Die Implementierungsbeziehung $\rightsquigarrow_\sigma$ ist transitiv: Gilt $SP_1 \rightsquigarrow_{\sigma_1} SP_2$ und $SP_2 \rightsquigarrow \sigma_2 SP_3$, so folgt $SP_1 \rightsquigarrow_{\sigma_1 \circ \sigma_2} SP_3$.*

Beweis. Wir beweisen den Satz der Einfachheit halber nur für identische Simulationen und lassen Sortenangaben weg. Es gelte $SP_1 \rightsquigarrow_{id} SP_2$ und $SP_2 \rightsquigarrow_{id} SP_3$.

Sei $B_3 \in \mathrm{Mod}(SP_3)$.

Dazu gibt es $B_2 \in \mathrm{Mod}(SP_2)$, $Rep^{B_3} \subseteq B_3$ und eine $\mathrm{Sig}(SP_2)$-Kongruenz $\sim_{B_3}$ mit $Rep^{B_3}/\sim_{B_3} \cong B_2$. Sei $abs_3 : Rep^{B_3} \to B_2$ die vom Isomorphismus induzierte Abbildung. Ebenso gibt es für B_2 ein $B_1 \in \mathrm{Mod}(Sp_1)$, $Rep^{B_2} \subseteq B_2$ und eine $\mathrm{Sig}(SP_1)$-Kongruenz $\sim_{B_2}$ mit $Rep^{B_2}/\sim_{B_2} \cong B_1$. Sei $abs_2 : Rep^{B_2} \to B_1$ die vom Isomorphismus induzierte Abbildung (vgl. Abb. 8).

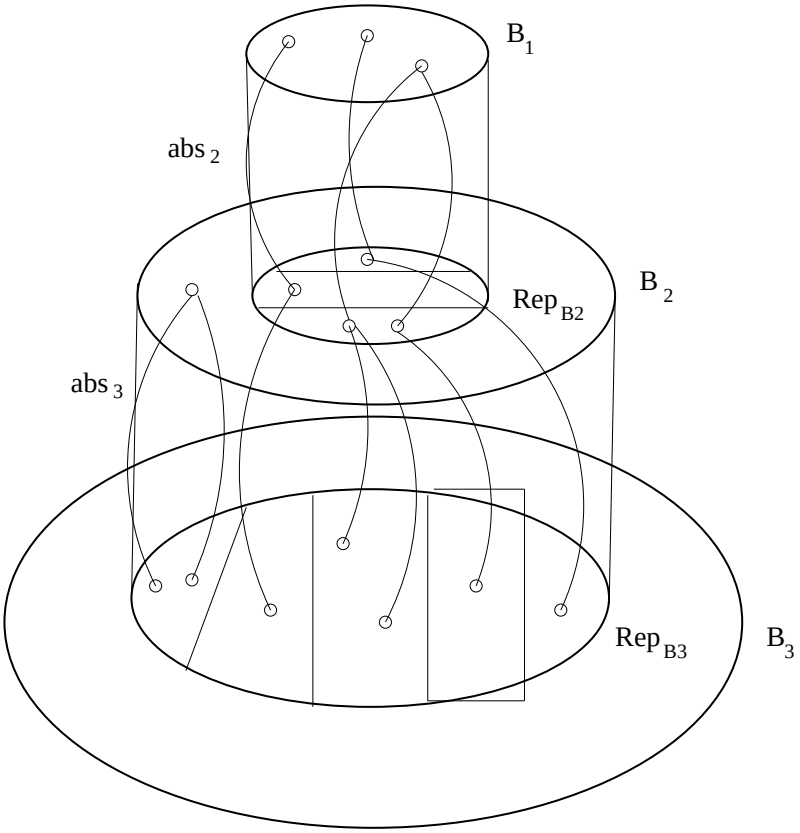


Abbildung 8: Transitivität der FRI-Implementierung.

Definiere

$$Rep_s \;=_{\mathrm{def}}\; \{x \in Rep_s^{B_3} \mid \text{es gibt } y \in Rep_s^{B_2} : abs_3(x) = y\}$$

und sei $abs = abs_2 \circ abs_3$. Dann wird die gewünschte Kongruenz auf Rep definiert durch

$$x \sim y \;\Leftrightarrow\; abs_2(abs_3(x)) = abs_2(abs_3(y))$$

Q.E.D.

Als abschließendes Beispiel betrachten wir die Implementierung von Kellern durch Felder.

Beispiel 55: Spezifikation „funktionaler“ Felder

Die folgende Spezifikation ARRAY beschreibt Felder mit Elementen vom Typ Data über einem Indextyp, spezifiziert durch INDEX. INDEX kann sowohl für einen endlichen Indextyp wie für einen unendlichen Indextyp (wie NAT oder INT) stehen.

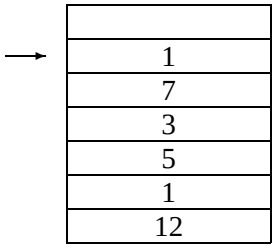


Abbildung 9: Darstellung von Kellern durch Feld und Zeiger.

spec ARRAY[INDEX] =
DATA **then**
sorts Array
generated type Array ::= $\omega \mid$ put(Array; Index; Data)
%%{ω leeres Feld, put Eintragen eines Elements}%
ops $\_[\_] : \text{Array} \times \text{Index} \to \text{Data}$ %%{direkter Zugriff}%
vars $i,j : \text{Index};\ x,y : \text{Data};\ a : \text{Array}$
axioms $\neg\ \text{def}\ \omega[j]$
$\text{put}(a,i,x)[j] = x$ **when** $i = j$ **else** $a[j]$
%%{Zusätzlich kann man noch verlangen:}%
$i = j \Rightarrow \text{put}(\text{put}(a,i,x),j,y) = \text{put}(a,j,y)$
$\neg\ (i = j) \Rightarrow \text{put}(\text{put}(a,i,x),j,y) = \text{put}(\text{put}(a,j,y),i,x)$
end

Bemerkung: Die Spezifikation ARRAY bietet nur die unbedingt notwendige Funktionalität für Felder. In einer komfortableren Spezifikation wird man natürlich noch weitere Funktionen wie „Länge eines Feldes“, „Test, ob der Inhalt eines Feldelements definiert ist“ anbieten. ◄

Die Pointer-Array Implementierung stellt jeden Keller als Paar, bestehend aus einem Feld und einer natürlichen Zahl (Adresse des top-Pointers) dar (vgl. Abb. 9). Die Kelleroperationen werden als explizite Definitionen über den Array-Pointer-Paaren angegeben.

spec STACK_by_ARRAYPOINTER =
ARRAY[NAT **fit** Index $\mapsto$ Nat] **then**
generated type Stack ::= pair(Array; Nat)
ops $\text{push} : \text{Data} \times \text{Stack} \to \text{Stack}$
$\text{empty} : \text{Stack}$
$\text{pop} : \text{Stack} \to \text{Stack}$
$\text{top} : \text{Stack} \to \text{Stack}$
vars $x : \text{Data};\ i : \text{Nat};\ a : \text{Array}$
axioms $\text{empty} = \text{pair}(\omega, 0)$
$\text{push}(x, \text{pair}(a,i)) = \text{pair}(\text{put}(a,i,x), \text{succ}(i))$
$\text{pop}(\text{pair}(a, \text{succ}(i))) = \text{pair}(a,i)$
$\neg\ \text{def}\ \text{pop}(\text{pair}(a,0))$
$\text{top}(\text{pair}(a, \text{succ}(i))) = a[i]$
$\neg\ \text{def}\ \text{top}(\text{pair}(a,0))$
end

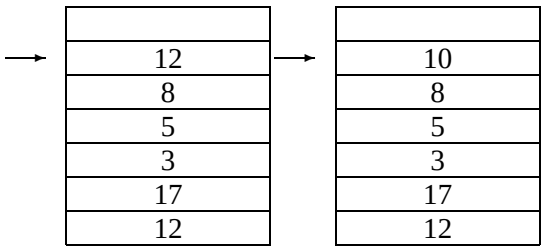
Beispiel 56: Implementierung von STACK durch STACK_by_ARRAYPOINTER: Beweis über Modelle

Um zu zeigen, dass STACK_by_ARRAYPOINTER eine FRI-Implementierung von STACK ist, wählen wir ein beliebiges Modell M von STACK_by_ARRAYPOINTER und definieren folgende Repräsentationsmenge *Rep* und folgende Kongruenz auf M, wobei die Stack-Elemente durch Konstruktorterme der Form $\text{pair}(a,i)$ dargestellt werden mit $a \in M_{\text{Array}}$ und $i \in M_{\text{Nat}}$:

$$\begin{array}{rcl} Rep^M_{\text{Data}} & = & M_{\text{Data}} \\ Rep^M_{\text{Stack}} & = & \{\text{pair}^M(a,i) \mid a \in M_{\text{Array}}, i \in M_{\text{Nat}}\} \\ \text{pair}^M(a,i) \sim^M \text{pair}^M(b,j) & \Leftrightarrow & i = j \wedge \forall k : \text{Nat}.\ k < i \Rightarrow a[k]^M = b[k]^M \end{array}$$

d.h. zwei Felder mit Zeigern sind genau dann gleich, wenn die Zeiger übereinstimmen und die Felder unterhalb des Zeigers übereinstimmen.

Die folgenden beiden Repräsentationen von Kellern sind äquivalent:



Man sieht sofort, dass $M' =_{\text{def}} Rep^M/\sim_M$ eine wohldefinierte Sig(STACK)-Algebra ist, die durch empty und push erzeugt ist.

Wir zeigen die Gültigkeit der beiden zentralen Axiome:

1. $M' \models \text{top}(\text{push}(x,\text{pair}(a,i))) = x$

   Sei dazu ν eine beliebige Belegung. Dann gilt

$$\begin{array}{rcll} M',\nu \models & & \text{top}(\text{push}(x,\text{pair}(a,i))) & \\ & = & \text{top}(\text{pair}(\text{put}(a,i,x),\text{succ}(i))) & \text{[Def. von push]} \\ & = & \text{put}(a,i,x)[i] & \text{[Def. von top]} \\ & = & x & \text{[Def. von put}(a,i,x)[i] \text{ in ARRAY]} \end{array}$$

2. $M' \models \text{pop}(\text{push}(x,\text{pair}(a,i))) = \text{pair}(a,i)$

   Sei ν eine beliebige Belegung.

$$\begin{array}{rcll} M',\nu \models & & \text{pop}(\text{push}(x,\text{pair}(a,i))) & \\ & = & \text{pop}(\text{pair}(\text{put}(a,i,x),\text{succ}(i))) & \text{[Def. von push]} \\ & = & \text{pair}(\text{put}(a,i,x),i) & \text{[Def. von pop]} \\ & \sim & \text{pair}(a,i) & \text{[für alle } k < i \text{ gilt: put}(a,i,x)[k] = a[k]\text{]} \end{array}$$

◀

Meist beweist man die Implementierungsbeziehung aber nicht auf Modellebene, sondern auf der Ebene der Spezifikationen. Dann benutzt man anstelle der Menge *Rep* das charakteristische Prädikat von *Rep*.

Wir formulieren folgenden Satz für flache Spezifikationen.

Satz 41 *Sei $SP = (\Sigma, E)$ eine flache Spezifikation, SP' eine Spezifikation mit $\mathrm{Sig}(SP') \supseteq \Sigma$ und sei $Ax(Rep, \sim)$ eine Axiomatisierung des charakteristischen Prädikats von Rep und der Kongruenzrelation $\sim$ über SP'. Sei*

$$\textbf{spec } SP'' = SP' \textbf{ then } Ax(Rep, \sim) \textbf{ end}$$

Dann ist SP' eine FRI-Implementierung von SP, falls SP'' die Axiome E von SP auf Rep modulo $\sim$ erfüllt, d.h.

$$SP'' \models G_{Rep,\sim} \quad \textit{für alle } G \in E$$

wobei $G_{Rep,\sim}$ induktiv definiert ist durch

$$\begin{array}{rcl}
p(t_1,\ldots,t_n)_{Rep,\sim} & \equiv & p(t_1,\ldots,t_n) \\
(u = v)_{Rep,\sim} & \equiv & u \sim v \\
(G_1 \wedge G_2)_{Rep,\sim} & \equiv & (G_1)_{Rep,\sim} \wedge (G_2)_{Rep,\sim} \\
(\neg\, G)_{Rep,\sim} & \equiv & \neg\, (G_{Rep,\sim}) \\
(\forall x : s.\ G)_{Rep,\sim} & \equiv & \forall x : s.\ Rep_s(x) \Rightarrow G_{Rep,\sim} \\
(\exists x : s.\ G)_{Rep,\sim} & \equiv & \exists x : s.\ Rep_s(x) \wedge G_{Rep,\sim}
\end{array}$$

Beispiel 57: Modellfreier Implementierungsbeweis für STACK_by_ARRAYPOINTER

Um auf der Ebene der Spezifikationen zu zeigen, dass STACK_by_ARRAYPOINTER eine FRI-Implementierung von STACK ist, definieren wir das charakteristische Prädikat *Rep* der Repräsentationsmenge und die Kongruenz $\sim$ axiomatisch über STACK_by_ARRAYPOINTER:

preds Rep_{Data} : Data
Rep_{Stack} : Stack
$\sim_{\mathrm{Data}}$: Data $\times$ Data
$\sim_{\mathrm{Stack}}$: Stack $\times$ Stack

vars x, y : Data; s : Stack; a, b : Array; i, j : Nat

axioms $Rep_{\mathrm{Data}}(x)$ %%{Rep_{Data} gilt für alle x : Data}%
$Rep_{\mathrm{Stack}}(s) \Leftrightarrow \exists a$: Array; i : Nat. $s = \mathrm{pair}(a, i)$
$x \sim_{\mathrm{Data}} y \Leftrightarrow x = y$
$\mathrm{pair}(a, i) \sim_{\mathrm{Stack}} \mathrm{pair}(b, j) \Leftrightarrow i = j \wedge \forall k$: Nat. $k < i \Rightarrow a[k] = b[k]$

Dann beweist man die Stackaxiome folgendermaßen:

1. STACK_by_ARRAYPOINTER" erfüllt das Axiom

   $$\forall x : \mathrm{Data};\ s : \mathrm{Stack}.\ \mathrm{top}(\mathrm{push}(x, s)) = x$$

   relativiert bezüglich der Axiome für *Rep* und $\sim$, also die Formel

   $$\forall x : \mathrm{Data};\ s : \mathrm{Stack}.\ Rep_{\mathrm{Data}}(x) \wedge Rep_{\mathrm{Stack}}(s) \Rightarrow \mathrm{top}(\mathrm{push}(x, s)) \sim_{\mathrm{Data}} x$$

   Einsetzen der Definitionen von *Rep* und $\sim$ sowie prädikatenlogische Umformung ergibt

   $$\forall x : \mathrm{Data};\ a : \mathrm{Array};\ i : \mathrm{Nat}.\ \mathrm{top}(\mathrm{push}(x, \mathrm{pair}(a, i))) = x$$

Der Beweis gelingt mit Hilfe der Axiome von STACK_by_ARRAYPOINTER:

$$\begin{array}{lll} & \mathrm{top}(\mathrm{push}(x,\mathrm{pair}(a,i))) & \\ = & \mathrm{top}(\mathrm{pair}(\mathrm{put}(a,i,x),\mathrm{succ}(i))) & \text{[Def. von push]} \\ = & \mathrm{put}(a,i,x)[i] & \text{[Def. von top]} \\ = & x & \text{[Def. von put}(a,i,x)[i] \text{ in ARRAY]} \end{array}$$

2. STACK_by_ARRAYPOINTER" erfüllt das Axiom

$$\forall x : \mathrm{Data};\ s : \mathrm{Stack}.\ \mathrm{pop}(\mathrm{push}(x,s)) = s$$

relativiert bezüglich der Axiome für *Rep* und $\sim$, also die Formel

$$\forall x : \mathrm{Data};\ s : \mathrm{Stack}.\ Rep_{\mathrm{Data}}(x) \wedge Rep_{\mathrm{Stack}}(s) \Rightarrow \mathrm{pop}(\mathrm{push}(x,s)) \sim_{\mathrm{Stack}} s$$

Einsetzen der Definition von *Rep* und prädikatenlogische Umformung ergibt

$$\forall x : \mathrm{Data};\ a : \mathrm{Array};\ i : \mathrm{Nat}.\ \mathrm{pop}(\mathrm{push}(x,\mathrm{pair}(a,i))) \sim_{\mathrm{Stack}} \mathrm{pair}(a,i)$$

Der Beweis erfolgt wieder mit Hilfe der Axiome von STACK_by_ARRAYPOINTER:

$$\begin{array}{lll} & \mathrm{pop}(\mathrm{push}(x,\mathrm{pair}(a,i))) & \\ = & \mathrm{pop}(\mathrm{pair}(\mathrm{put}(a,i,x),\mathrm{succ}(i))) & \text{[Def. von push]} \\ = & \mathrm{pair}(\mathrm{put}(a,i,x),i) & \text{[Def. von pop]} \\ \sim_{\mathrm{Stack}} & \mathrm{pair}(a,i) & \text{[für alle } k < i \text{ gilt put}(a,i,x)[k] = a[k]] \end{array}$$

◀

Bemerkung: Beachten Sie, dass in der Implementierung das Gesetz

$$\mathrm{pop}(\mathrm{push}(x,s)) = s$$

nicht gilt, sondern nur

$$\mathrm{pop}(\mathrm{push}(x,s)) \sim_{\mathrm{Stack}} s$$

5.3 Exkurs: Kurze Einführung in SML

SML ist ein funktionale Programmiersprache, die ab ca. 1978 von Robin Milner als Metasprache für LCF, eines der ersten interaktiven Beweissysteme, entwickelt wurde. Die Sprache erfreute sich bald so großer Beliebtheit, dass eine Standardisierung notwendig wurde (ca. 1985). Weiterentwickelt wurde SML u.a. von Milner, MacQueen (Modulsystem), Tofte, Harper, Cardelli etc.

SML ist eine streng typisierte Sprache mit Konzepten für rekursive Datentypen, Ausnahmen, parametrische Polymorphie und Moduln. Einer der Vorteile von SML ist, dass der Programmierer fast keine Typinformation zum Programm geben muss, da der allgemeinste Typ eines SML-Ausdrucks vom Typinferenzsystem von SML berechnet wird.

Literatur

- L. Paulson: SML for the Working Programmer. Cambridge University Press, 2. Auflage 1997 (???)
- ...

5.3.1 Typen

Datenobjekte in SML sind Objekte/Werte 1.Klasse. Sie können

1. als Argumente und Resultate von Funktionen auftreten,
2. in einer Deklaration benannt sein und
3. als Teil eines strukturierten Objektes auftreten.

In SML gibt es unter anderem die drei Standardtypen `bool`, `int` und `real`. Weiterhin gibt es drei Möglichkeiten, strukturierte Typen aufzubauen: Sind `s` und `s'` Typen, so auch

- `s => s'`: Funktionen von `s` nach `s'`,
- `s * s'`: kartesisches Produkt und
- `s list`: Listen mit Elementen vom Typ `s`.

Beispiel 58: Typen in SML

- `int * int`
- `int * bool * (int => bool)`
- `int list`
- `(int * bool) list`
- `(int => int) list` ◄

Der Typ der Listen ist **polymorph** (parametrische Polymorphie), was durch Unbestimmte für Typen (sogenannte „Typvariablen“) ausgedrückt wird. Eine Typvariable wie `'a` wird mit einem führenden Apostroph geschrieben. Man verwendet Doppelapostroph (`''a`), wenn der Parametertyp eine Gleichheit `=` anbieten muss. Bei `real` ist dies z.B. nicht der Fall.

Beispiel 59: parametrisierte Typen

- `'a list`
- `''b set`
- `int list` (`'a` mit int instanziert)
- `(int * 'b) list` ◄

Die Listenkonstruktoren von SML sind

- `nil` bzw. `[]: 'a list` die leere Liste
- `:: : 'a * 'a list => 'a list` Anfügen eines Elements an eine Liste (`::` assoziiert nach rechts).

  Ein Ausdruck wie `12 :: -17 :: 3 :: []` kann auch geschrieben werden als `[12, -17, 3]`.

Beispiel 60: Spezielle endliche Listen

Das folgende Beispiel zeigt die Interaktion mit einem SML-Interpreter. Eingaben (eingeleitet durch den Prompt -) sind kursiv, Ausgaben (eingeleitet durch >) normal dargestellt. Während beim ersten Beispiel der Typ der Eingabe explizit vom Benutzer angegeben wird, geben die weiteren Beispiele einen ersten Eindruck von der Typinferenz in SML.

```
- []: 'a list;
> []: 'a list
- 1 :: [];
> [1]: int list
- 1 :: 2 :: 3 :: [];
> [1, 2, 3]: int list
```

$[x_1, \ldots, x_n]$ ist die Normalform für endliche Listen. ◀

Beispiel 61: Listen von Paaren und Listen von Listen

- `[2, 3]: int list`
- `[(2, 3), (3, 4)]: (int * int) list`
- `[[1, 2], [2, 3, 4]]: (int list) list`
- ```
  - true::false::nil;
  > [true, false] : bool list
  - true::1::nil;
  > Fehler!
  ```

◀

Selbstdefinierte Typen können auf zweierlei Weisen eingeführt werden: Einfache *Typabkürzungen* haben die Form

`type s =` τ

für einen Typausdruck τ.

Beispiel 62: Selbstdefinierte Typen

- `type rat = int * int`
- `type vector = int * real * real`
- `type 'a listpair = 'a list * 'a list` ◀
  ```

Datentypen mit Konstruktoren werden deklariert als

`datatype` τ `= C | ...| D of (`s_1 `* ...*` s_n`)`

wobei τ und `C, D` Namen sind und $s_1,\ldots, s_n$ Typausdrücke, in denen τ auftreten kann. Durch diese Deklaration werden der Typ τ und die Konstruktoren

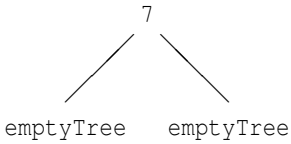
`C :` τ

`D : (`s_1 `* ...*` s_n`) => `τ

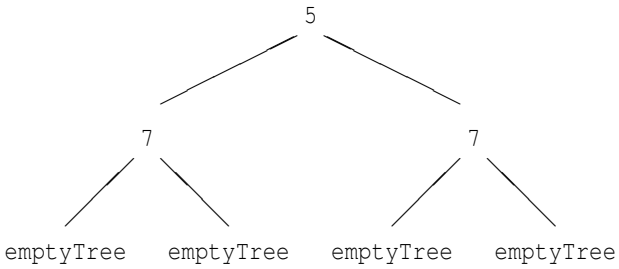
eingeführt.

Beispiel 63: Binäre Bäume mit Knotenmarkierung

```
datatype 'a tree = emptyTree | node of 'a tree * 'a * 'a tree
```




```
- val leaf7 = node(emptyTree, 7, emptyTree)
> val leaf7 = node(emptyTree, 7, emptyTree): int tree
```




```
- val tree5 = node(leaf7, 5, leaf7)
> val tree5 = node(node(...), 5, node(...)) : int tree
```

◄

Beispiel 64: Currency

Im Datentyp `currency` wird zwischen deutscher und österreichischer Währung unterschieden.

```
- datatype currency = DM of int | OeS of int;
> con DM = fn : int => currency
con OeS = fn : int => currency
```

Durch

`DM : int => currency` und
`OeS : int => currency`

werden zwei verschiedene Konstruktoren definiert. ◄

5.3.2 Werte und Funktionen

Jeder SML-Ausdruck hat einen *Wert*, der einer ganz bestimmten Menge von Objekten angehört, die durch den Typ (oder die Sorte) des Ausdrucks beschrieben wird. Der Typ eines Ausdruckes wird hinter einem Doppelpunkt angegeben.

Eine Wertdeklaration hat die Form

```
val c = e
```

wobei `c` ein Name und `e` ein Ausdruck ist.

Funktionen sind spezielle Werte mit Deklarationen der Form

```
fun f x = e
```

wobei `f` der Funktionsname, `x` der aktuelle Parameter und `e` ein Ausdruck ist, in dem `f` vorkommen darf.

Beispiel 65: Rekursive Definition der Summe

Eine Funktion zum Aufsummieren der Zahlen von 1 bis n, also zur Berechnung von $\sum_{i=1}^{n} i$, kann in SML folgendermaßen definiert werden:

```
- fun sum n = if n = 1 then 1 else sum (n - 1) + n;
> val sum = fn: int => int
```

◀

Funktionsdefinition durch Mustervergleich (Pattern Matching)

Die Konstruktoroperatoren von Datentypen erlauben es in einfacher Weise, Fallunterscheidungen bei der Beschreibung des Funktionsrumpfes anzugeben. Wir schreiben

```
fun ⟨name⟩ ⟨pattern 1⟩ = ⟨ausdruck 1⟩
  | ⟨name⟩ ⟨pattern 2⟩ = ⟨ausdruck 2⟩
  ...
  | ⟨name⟩ ⟨pattern n⟩ = ⟨ausdruck n⟩
```

Beispiel 66:

1. Prüfen auf leere Liste

   ```
   - fun null nil = true
     | null (x::l) = false;
   > val null = fn : 'a list => bool
   ```

2. Quadrieren der Elemente einer Liste

   ```
   fun quad x : int = x * x
   fun quadlist nil = nil
     | quadlist (x::l) = (quad x) :: quadlist l
   ```

3. Wechseln in Schilling

   ```
   fun wechsle_in_OeS (DM x) = OeS (7 * x)
     | wechsle_in_OeS (OeS x) = OeS x
   ```

4. Vergleich auf dem Typ currency

```
fun equal (OeS x, OeS y) = (x = y)
  | equal (DM  x, DM  y) = (x = y)
  | equal (OeS x, DM  y) = false
  | equal (DM  x, OeS y) = false
```

◀

5.3.3 Ausnahmen

Um undefinierte Situationen in SML exakt zu behandeln, gibt es eine **Fehlerbehandlung** in SML. Mit

`exception` ⟨*name*⟩

wird eine einfache Ausnahme definiert, die nur den Namen ⟨*name*⟩ angibt und den trivialen Typ besitzt, der genau ein Objekt `{}` als Element hat.

Mit

`raise` ⟨*name*⟩

wird die Berechnung abgebrochen und die Fehlerbehandlung aufgerufen. Damit lässt sich auch für andere Funktionen die Ausnahmebehandlung genau eingrenzen.

Beispiel 67:

1. Fakultätsfunktion für nichtnegative Zahlen

```
exception fac;
fun fac n = if n < 0 then raise fac
            else if n = 0 then 1
            else n * fac(n-1)
```

2. Letztes Element einer Liste

```
exception last
fun last nil = raise last
  | last (x :: nil) = x
  | last (x :: y :: l) = last (y :: l)
```

◀

5.3.4 Signaturen

In SML wird die Signatur

$$\begin{array}{lcll} \Sigma & = & (S,F) & \text{mit} \\ S & = & \{s_1,\ldots,s_n\} & \text{und} \\ F & = & \{f_1:\tau_1,\ldots,f_m:\tau_m\} & \end{array}$$

folgendermaßen dargestellt:

```
signature Σ =
sig
    type s_1
    ⋮
    type s_n
    val  f_1 : τ_1
    ⋮
    val  f_m : τ_m
end
```

Beispiel 68: Signatur Σ_{NUM} der zahlartigen Strukturen

```
signature NUMSIG =
sig
    type num
    val  add: num * num => num
    val  mult: num * num => num
end
```

◀

Allgemein deklariert eine Signatur eine Liste von Namen für Typen oder Werte, wobei für Werte der zugehörige Typ angegeben wird. Man schreibt

`type 'a T`	zur Angabe des polymorphen Typs `'a T`,
`val c : 'a T`	zur Angabe einer Konstante und
`val f : 'a T => 'b U`	zur Angabe einer einstelligen Funktion (nicht `fun f`!)

In SML werden vordefinierte Typen und Konstanten automatisch zu jeder Signatur gezählt; deshalb können `bool`, `int`, `'a list` etc. immer verwendet werden, im Gegensatz zu benutzerdefinierten Signaturen.

5.3.5 Strukturen

Analog zum Signaturkonzept gibt es in SML ein Konzept zur Deklaration von Strukturen: Durch

```
structure D : Σ =
struct
    ⟨Liste von Deklarationen⟩
end
```

wird eine neue Struktur `D` definiert. Die Liste $d_1 \ldots d_n$ von Deklarationen muss für jedes Element von Σ eine Deklaration enthalten; es dürfen auch noch mehr Symbole deklariert werden, diese sind außerhalb von `D` aber nicht sichtbar (sogenannte verborgene Symbole).

Unter den Deklarationen können natürlich

- Typdeklarationen (der Form `type s =` τ oder `datatype t = ...`),
- Wertdeklarationen (der Form `val c = e`),
- Funktionsdeklarationen(der Form `fun f x = e`),
- Ausnahmedeklarationen (der Form `exception E`) und selbst wieder

- Strukturdeklarationen (der Form `structure D1 = e`)

auftreten.

Vordefinierte Typen und Funktionen können ohne explizite Deklaration verwendet werden.

Beispiel 69: Zahlartige Strukturen

Zur Signatur `NUMSIG` aus Beispiel 68 passen die beiden folgenden Strukturen, in denen die Sorte `num` durch ganze bzw. rationale Zahlen implementiert wird.

1. Struktur der ganzen Zahlen

```
structure INT : NUMSIG =
struct
    type num = int
    fun  add(x,y) = x + y
    fun  mult(x,y) = x * y
end
```

2. Die Struktur der rationalen Zahlen kann folgendermaßen definiert werden.

```
structure RAT : NUMSIG =
struct
    type num = int * int
    fun  add((z1, n1), (z2, n2)) = (z1 * n2 + z2 * n1, n1 * n2)
    fun  mult((z1, n1), (z2, n2)) = (z1 * z2, n1 * n2)
end
```

◀

Um auf eine Funktion `f` oder einen Typ `s` aus einer Struktur `D` (außerhalb von `D`) zugreifen zu können, schreibt man `D.f` bzw. `D.s`.

Dadurch werden Namenskonflikte zwischen gleichbezeichneten Symbolen aus verschiedenen Strukturen vermieden. Mithilfe des Befehls

```
open D
```

öffnet man eine Struktur `D` und kann dann auf `f` bzw. `s` direkt zugreifen. Allerdings muss man hierbei auf mögliche Namenskonflikte achten.

5.4 Übergang zu funktionalen Programmen

Axiomatische Spezifikationen sind im Allgemeinen nicht ausführbar.

Beispiel 70: Inverse einer (bijektiven) Funktion

spec INV =
INT **then**
ops $f, g : \text{Int} \to \text{Int}$
vars $x : \text{Int}$
axioms $f(g(x)) = x$
$g(f(x)) = x$
end

Die Spezifikation INV definiert zwei zueinander inverse Funktionen f und g über den ganzen Zahlen; da nichts weiteres über f und g bekannt ist, ist z.B. $f(5)$ nicht ausführbar. ◄

In vielen Fällen sind Spezifikationen weniger abstrakt und direkt ausführbar bzw. direkt in ein funktionales Programm überführbar. Wir geben im Folgenden eine spezielle Form von ausführbaren Spezifikationen an, die in funktionale Programme übersetzbar sind.

Definition 42 *Sei SP eine Spezifikation, bei der*

- *jede Sorte s absolut frei, d.h. durch Konstruktoren folgendermaßen definiert ist*

  free type $s ::= c_1 \mid \ldots \mid c_n(s_1; \ldots; s_m)$

  oder ein Grunddatentyp in SML ist.

- *jede Operation f induktiv durch Axiome der Form*

  $b(\overline{t}) \Rightarrow f(\overline{u}) = e$

  definiert ist, wobei $\overline{t} = (t_1, \ldots, t_k)$ *und* $\overline{u} = (u_1, \ldots, u_n)$ *Tupel von Konstruktortermen sind.*

Dann heißt SP in ausführbarer Normalform.

Sei *SP* eine Spezifikation in ausführbarer Normalform. Dann kann *SP* folgendermaßen in ein SML-Programm übersetzt werden:

- Jede Sorte *s* mit Konstruktoren $c_1, \ldots, c_n$ wird dargestellt als Datentyp

  ```
  datatype s = c1 | ...| cn of s1 * ...* sm
  ```

- Jede Operation f wird induktiv durch Pattern Matching definiert:

  ```
  fun f(u1) = if b11 then e11 else ...
    | ...
    | f(uk) = if bk1 then ek1 else ...
  ```

  wobei alle Axiome für `f(u)` auf der rechten Seite durch eine Fallunterscheidung geeignet zusammengefasst werden.

  - Für undefinierte Situationen werden Ausnahmen deklariert und ausgelöst.
  - Parametrisierung mit dem polymorphen trivialen Typ ELEM wird durch eine polymorphe Typvariable `'elem` und
  - Parametrisierung mit dem polymorphen Gleichheitstyp ELEM_EQ wird durch eine polymorphe Typvariable `''elem` ausgedrückt.

- Jede algebraische Signatur wird in eine korrespondierende SML-Signatur übersetzt.

- Jede Spezifikation wird in eine implementierende Struktur übersetzt.

- Eine hierarchische Spezifikation **spec** $SP = SP_1$ **then** *body* **end** wird übersetzt in eine Implementierung

  ```
  structure SP1 = struct ...end
  ```

  von SP_1 und eine Implementierung von *SP*

  `structure SP = struct` ⟨Übersetzung von *body*⟩ `end`

  wobei auf ein Symbol f von SP_1 in der Übersetzung von *body* mittels `SP1.f` zugegriffen wird.

Beispiel 71: Keller

Die Spezifikation der Keller

spec STACK =
ELEM **then**
free type Stack ::= empty | push(Elem; Stack)
ops top : Stack →? Elem
pop : Stack →? Stack
vars x : Elem; s : Stack
axioms ¬ def top(empty)
top(push(d,s)) = d
¬ def pop(empty)
pop(push(d,s)) = s
end

ist in ausführbarer Normalform und kann deshalb nach obigem Schema folgendermaßen nach SML übersetzt werden.

Wir bilden zunächst die Signatur

```
signature STACKSIG =
sig
    exception emptyException
    type 'a stack
    val  empty : 'a stack
    val  push : 'a * 'a stack => 'a stack
    val  top : 'a stack => 'a
    val  pop: 'a stack => 'a stack
end
```

Dann ergibt sich folgende Struktur aus der Datentrypdeklaration und den Axiomen der Spezifikation:

```
structure STACK =
struct
    exception emptyException
    datatype 'a stack = empty | push of 'a * 'a stack
    fun  top(empty) = raise emptyException
      |  top(push(x, s)) = x
    fun  pop(empty) = raise emptyException
      |  pop(push(x, s)) = s
end
```

◄

5.5 Zusammenfassung

- Programmentwicklung geht von den Anforderungen zur Implementierung, d.h. von der Beschreibung des Problems (das „Was“) zur Beschreibung der Lösung (das „Wie“). Man beschreibt den Programmentwicklungsprozess formal durch eine Kette von Verfeinerungen

  $$SP_n \quad \rightsquigarrow \quad SP_{n-1} \quad \rightsquigarrow \quad \ldots \quad \rightsquigarrow \quad SP_0$$

  mit SP_n als Anforderungsspezifikation und SP_0 als Lösung in ausführbarer Form.

- Eine Spezifikation SP' ist eine *Verfeinerung* von SP (geschrieben $SP \rightsquigarrow SP'$), falls die Signaturen von SP und SP' gleich sind und SP' mehr Eigenschaften erfüllt als SP. Eine Verfeinerung heißt *axiomatische Anreicherung*, falls nur neue Axiome hinzugefügt werden, und *axiomatische Verfeinerung modulo Umbenennung*, falls die Namen in SP' an die Namen von SP angepasst werden müssen.

- Man spricht von *Wechsel der Datenstruktur*, wenn eine Σ_1-Algebra B eine Σ-Algebra A folgendermaßen simuliert:

  Jede Trägermenge von A wird durch eine Teilmenge *Rep* einer Trägermenge von B repräsentiert, und jedes Funktionssymbol von Σ wird durch ein Funktionssymbol von Σ_1 repräsentiert. Dabei können mehrere Elemente von *Rep* das gleiche Element von A repräsentieren, d.h. es muss eine Äquivalenzrelation $\sim$ auf *Rep* existieren, die die Repräsentanten gleicher Elemente in Beziehung setzt.

  Für eine Simulation muss gelten, dass $\sim$ eine Σ-Kongruenz auf *Rep* ist und dass die Quotientenalgebra $Rep/\sim$ isomorph zu A ist.

- Eine Spezifikation SP_1 *FRI-implementiert* eine Spezifikation SP bzgl. eines Signaturmorphismus ρ (geschrieben $SP_1 \rightsquigarrow_\rho SP$), falls jedes Modell von SP_1 ein Modell von SP bzgl. passend gewählten *Rep* und $\sim$ simuliert. Die FRI-Implementierungsrelation ist transitiv und kann deshalb anstelle der Verfeinerungsrelation verwendet werden.

- Man beweist die Implementierungsbeziehung auf der Ebene der Spezifikationen. Dann benutzt man anstelle der Menge *Rep* das charakteristische Prädikat von *Rep*. Eine Spezifikation SP' *FRI-implementiert* eine Spezifikation SP, wenn es eine Axiomatisierung des charakteristischen Prädikats von *Rep* und von der Kongruenzrelation $\sim$ über SP' so gibt, dass in der Erweiterung von SP' um diese Axiomatisierung die Kongruenzeigenschaft von $\sim$ bzgl. *Rep* und die Axiome von SP bewiesen werden können.

- Eine Spezifikation ist in ausführbarer Normalform, wenn jede Sorte absolut frei ist, und wenn jede Operation strukturell rekursiv durch bedingte Gleichungen über Konstruktortermen der Form

  $$b(\overline{t}) \Rightarrow f(\overline{u}) = e$$

  definiert ist. Solche Spezifikationen können schematisch in die funktionale Sprache SML übersetzt werden.

Teil III

Spezifikation zustandsbasierter Systeme

In diesem Teil der Vorlesung betrachten wir Systeme, die einen Zustand besitzen. Die Ausführung der Operationen ist im Gegensatz zur funktionalen Sichtweise abhängig von einem implizit gegebenen Zustand und verändert möglicherweise den Zustand.

6 Zustände und Transitionssysteme

Ziele

- Den Unterschied zwischen explizit und implizit gegebenem Zustand verstehen
- Den Begriff des Zustands und des Transitionssystems verstehen lernen

In der funktionalen Sichtweise werden alle relevanten Daten inklusive des Zustands als Parameter übergeben. Zustände werden häufig durch Konstruktorterme beschrieben.

Beispiel 72: Keller

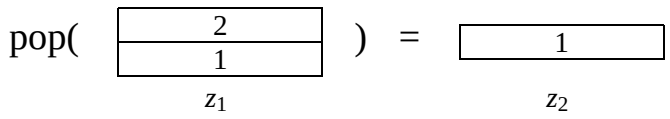
Die Signatur der Kellerstrukturen über natürlichen Zahlen ist gegeben durch

sig STACKSIG =
sorts Nat, Stack
ops empty : Stack
push : Nat × Stack → Stack
top : Stack → Nat
pop : Stack → Stack
end

Ein Zustand z_1 sei gegeben durch folgenden Konstruktorterm

$$z_1 \;=_{\text{def}}\; \text{push}(2, \text{push}(1, \text{empty}))$$

Der Aufruf $\text{pop}(z_1)$ führt den „Zustand“ z_1 in den „Zustand“ $z_2 = \text{push}(1, \text{empty})$ über.



◄

Bei einem zustandsbasierten System ist der Zustand implizit, d.h. der Typ Stack wird durch den trivialen Typ Unit ersetzt und erscheint nicht mehr in der Funktionalität der Operationen.

Die Operationen erhalten folgende Typen:

ops empty : Unit
push : Nat → Unit
top : Unit → Nat
pop : Unit → Unit

Ein Aufruf von pop() führt den Zustand z_1 in den Zustand z_2 über. Ein darauf folgendes push(4) ergibt den Zustand

$$z_3 \;=_{\text{def}}\; \text{push}(4, \text{push}(1, \text{empty}))$$

Beispiel 73: Person

Die Beschreibung von Personen erfordert mehrere Attribute:

name : Name
address : Address
age : Nat

d.h. ein Zustand ist durch mehrere Komponenten gegeben, deren Werte man (im impliziten Ansatz) z.B. über folgende Zugriffsoperationen erhalten kann:

getName : Unit $\rightarrow$ Name
getAdress : Unit $\rightarrow$ *Address*
getAge : Unit $\rightarrow$ *Nat*

Funktional (algebraisch) definiert man eine Sorte

sorts Person

für die Menge der Zustände mit einem Konstruktor

makePerson : Name $\times$ Address $\times$ Nat $\rightarrow$ Person

und den Selektoroperationen

getName : Person $\rightarrow$ Name
getAdress : Person $\rightarrow$ Address
getAge : Person $\rightarrow$ Nat

Imperativ beschreiben wir einen Zustand durch Angabe der Werte seiner Komponenten. ◄

Definition 43 *Gegeben sei eine Signatur $\Sigma = (S, F, P)$ und eine Σ-Struktur A.*

1. *Eine* Zustandssignatur *ist eine Familie von (System-) Variablen $(X_s)_{s \in S}$ mit X_s abzählbar für alle $s \in S$.*
2. *Ein* A-Zustand *ist eine Belegung $\sigma : X_s \rightarrow A_s$ für alle $s \in S$.*

Beispiel 74: Person

Die Zustandssignatur von Person ist

name : Name
address : Address
age : Nat

Eine Struktur *A* mit den Sorten Name, Address, Nat ist z.B.

$$\begin{aligned} A_{\text{Name}} &=_{\text{def}} \text{char}^* \\ A_{\text{Address}} &=_{\text{def}} \text{char}^* \times \mathbb{N} \times \text{char}^* \\ A_{\text{Nat}} &=_{\text{def}} \mathbb{N} \end{aligned}$$

Ein Zustand s ist z.B.

$$\begin{aligned} s(\text{name}) &= \text{“wirsing”} \\ s(\text{address}) &= (\text{“Schlagintweitstr.”}, 18, \text{“München”}) \\ s(\text{age}) &= 51 \end{aligned}$$

◀

Ein Transitionssystem beschreibt eine Menge von Zustandsübergängen:

Definition 44 *Gegeben sei eine Signatur* $\Sigma = (S, F, P)$.

1. *Ein* Transitionssystem $\Gamma = (Z, \delta)$ *ist gegeben durch*
   - *eine Menge Z von* Σ*-Zuständen und*
   - *eine Transitionsrelation* $\delta \subseteq Z \times Z$.
2. *Ein* markiertes Transitionssystem $\Gamma = (Z, \mathcal{A}, \delta)$ *ist gegeben durch*
   - *eine Menge Z von Zuständen*
   - *eine Menge* $\mathcal{A}$ *von Aktionen und*
   - *eine Transitionsrelation* $\delta \subseteq Z \times \mathcal{A} \times Z$.

Beispiel 75: Keller

Sei K eine Kellerstruktur mit

$$K_{\text{Stack}} =_{\text{def}} \mathbb{N}^*$$

und Zustandssignatur

stack : Stack

Dann sei die Menge der Zustände gleich K_{Stack}, d.h.

$$Z =_{\text{def}} K_{\text{Stack}} = \mathbb{N}^*$$

die Menge der Aktionen sei gleich den Aufrufen der (imperativen) Stackoperationen, d.h.

$$A =_{\text{def}} \{\text{push}(n) \mid n \in \mathbb{N}\} \cup \{\text{pop}(), \text{top}()\}$$

und das markierte Transitionssystem habe die Transitionsrelation δ mit

$$\begin{aligned} &(\langle \text{stack} = s \rangle, \text{push}(n), \langle \text{stack} = \langle n, s \rangle \rangle) \in \delta \\ &(\langle \text{stack} = \langle n, s \rangle \rangle, \text{pop}(), \langle \text{stack} = s \rangle) \in \delta \\ &(\langle \text{stack} = \langle n, s \rangle \rangle, \text{top}(), \langle \text{stack} = \langle n, s \rangle \rangle) \in \delta \end{aligned}$$

für alle $n \in \mathbb{N}$.

Anmerkung:

- $\text{stack} = s$ ist Abkürzung für die Aussage $\sigma(\text{stack}) = s$
- Wir beschreiben die Wirkungen der Operationen häufig auch folgendermaßen:

$$\begin{aligned} \text{stack} = s &\xrightarrow{\text{push}(n)} \text{stack} = \langle n, s\rangle \\ \text{stack} = \langle n, s\rangle &\xrightarrow{\text{pop}()} \text{stack} = s \\ \text{stack} = \langle n, s\rangle &\xrightarrow{\text{top}()} \text{stack} = \langle n, s\rangle \end{aligned}$$

◀

In diesem Beispiel wird top nicht adäquat berücksichtigt; deshalb betrachten wir Aktionen mit Ein- und Ausgabevariablen d.h.

$$\begin{aligned} \mathcal{A} =_{\text{def}} \quad & \mathbb{N} \times \text{push} \times \text{Unit} \\ \cup \quad & \text{Unit} \times \text{pop} \times \text{Unit} \\ \cup \quad & \text{Unit} \times \text{top} \times \mathbb{N} \end{aligned}$$

Die Transitionsrelation hat dann die Form:

$$\begin{aligned} &(\langle \text{stack} = s\rangle, (n, \text{push}, ()), \langle \text{stack} = \langle n, s\rangle\rangle) \\ &(\langle \text{stack} = \langle n, s\rangle\rangle, ((), \text{pop}, ()), \langle \text{stack} = s\rangle) \\ &(\langle \text{stack} = \langle n, s\rangle\rangle, ((), \text{top}, n), \langle \text{stack} = \langle n, s\rangle\rangle) \end{aligned}$$

für alle $n \in \mathbb{N}$.

7 Modellorientierte Spezifikationen am Beispiel von Z

Algebraische Spezifikationen sind eigenschaftsorientiert, sie beschreiben „nur“ die erwünschten Eigenschaften, die ein Programm oder Software-System erfüllen soll.

Eine modellorientierte Spezifikation basiert auf der Konstruktion von Spezifikationen, ausgehend von festen Strukturen, deren Existenz bekannt ist. Operationen werden durch Angabe von Eigenschaften der Vor- und Nachzustände beschrieben.

Z wurde ab 1978 in Oxford von B. Suffrin, J.R. Abrial und der „Oxford Programming Research Group“ entwickelt. Eine andere bekannte modellorientierte Spezifikationssprache ist VDM, „Vienna Development Method“, die ab 1970 im IBM-Labor Wien von C. Jones und D. Bjørner entwickelt wurde. Neuere Entwicklungen kombinieren VDM mit algebraischen Spezifkationen (RAISE, D. Bjørner ab 1985) oder führen objekt-orientierte Konstrukte ein (Object-Z, Carrington, Duke et al. 1990; Z++, Lano 1994).

7.1 Die Spezifikationssprache Z

Z ist eine mengenorientierte Spezifikationssprache. Spezifikationen in Z beschreiben Mengen, und Z-Konstrukte sind Operationen auf Mengen. Mathematische Basis ist die axiomatische Mengenlehre, wie sie von Zermelo und Fraenkel definiert wurde („ZF-Mengenlehre“). Eine Z-Spezifikation besteht aus einer Mischung von formalem mathematischen Text und informellen natürlichsprachlichen Erklärungen.

Der formale Text besteht aus einer Folge von „Paragraphen“, die Schritt für Schritt („Definition vor Gebrauch“) Schemata, Grundtypen, globale Variablen und Konstanten einführen.

Algebraische Spezifikationen bauen auf einer einfacheren Welt als der von Z auf, nämlich auf der der Sorten und Funktionssymbole, deren Eigenschaften durch Formeln, meist bedingte Gleichungen, angegeben werden. Algebraische Spezifikationen werden anders geschrieben: Man überlegt sich die Konstruktoren, die Selektoren und evtl. weitere interessante Funktionen, die zu einer Sorte gehören.

Z-Spezifikationen enthalten also zwei Hauptkomponenten:

1. mathematische Grundtypen zur Datenmodellierung, die mit Hilfe von Mengenlehre und Prädikatenlogik beschrieben werden,
2. Schemata, aus denen Spezifikationen aufgebaut sind und die als unvollständige Spezifikationen verstanden werden können.

Mit Schemata werden sowohl statische als auch dynamische Aspekte eines Systems modelliert. Statische Aspekte beinhalten:

- die möglichen Zustände eines Systems,
- die Invarianten, die erhalten bleiben, wenn das System in einen anderen Zustand übergeht.

Dynamische Aspekte beinhalten:

- die möglichen Operationen,
- die Ein-/Ausgabebeziehung,
- die möglichen Zustandsänderungen.

7.2 Grundrechenstrukturen und Logik von Z

Die Z-Notation basiert auf dem Prädikatenkalkül 1. Stufe und typisierter Mengenlehre. Sie ist sehr nahe zur üblichen mathematischen Notation. Wir folgen hier dem Z-Standard.

7.2.1 Logik

Formeln werden gebildet aus atomaren Formeln (wie $x \in S$, $S \subseteq T$, $x \underline{R} y$ für eine Relation R), den üblichen logischen Verknüpfungen und typisierten Quantorenformeln:

$\neg P$	nicht P
$P \wedge Q$	P und Q
$P \vee Q$	P oder Q
$P \Rightarrow Q$	P impliziert Q
$P \Leftrightarrow Q$	P gilt genau dann, wenn Q gilt
$\forall x : T \mid P \bullet Q$	Für alle x vom Typ T, die P erfüllen, gilt Q
$\forall x : T \bullet Q$	Für alle x vom Typ T gilt Q (Spezialfall)
$\exists x : T \mid P \bullet Q$	Es existiert ein x vom Typ T, das P und Q erfüllt
$\exists_1 x : T \mid P \bullet Q$	Es existiert genau ein x vom Typ T, das P und Q erfüllt

Bemerkung:

- Die Begriffe "Typ" und "Menge" werden in Z austauschbar benutzt.
- Der eindeutige Existenzquantor $\exists_1$ kann als Abkürzung eingeführt werden:

$$\exists_1 x : T \mid P \bullet Q \;\; =_{\text{def}} \;\; (\exists x : T \bullet P \wedge Q) \wedge (\forall x, y : T \bullet P[y/x] \wedge Q[y/x] \wedge P \wedge Q \Rightarrow x = y)$$

7.2.2 Mengenlehre

Die Notation für Mengen umfaßt:

$x \in S$	x ist Element von S
$S \subseteq T$	S ist Teilmenge von T, d.h. $\forall x : S \bullet x \in T$
$\emptyset$, $\{\}$	leere Menge
$\{x_1, \ldots, x_n\}$	Menge, die genau aus $x_1, \ldots, x_n$ besteht
$\{x : T \mid P\}$	Menge der x vom Typ T, die P erfüllen
$\{x : T \mid P \bullet t\}$	Menge aller Werte von t für alle x, die P erfüllen, d.h. $\{t(x) \mid x \in T \wedge P(x)\}$
$\mu x : T \mid P$	das einzige x vom Typ T, das P erfüllt
$\mu x : T \mid P \bullet t$	der Wert von t für das einzige x vom Typ T, das P erfüllt
$(x_1, \ldots, x_n)$	geordnetes n-Tupel
$S_1 \times \ldots \times S_n$	Kartesisches Produkt, d.h. $\{x_1 : S_1; \ldots; x_n : S_n \bullet (x_1, \ldots, x_n)\}$
$\mathbb{P} S$	Menge aller Teilmengen von S
$\mathbb{F} S$	Menge aller endlichen Teilmengen von S
$S \cap T$	Durchschnitt von S und T, d.h. $\{x : S \mid x \in T\}$
$S \cup T$	Vereinigung von S und T, d.h. $\{x : X \mid x \in S \vee x \in T\}$ (X Typ der Elemente von S und T)
$S \setminus T$	Mengendifferenz, d.h. $\{x : S \mid x \notin T\}$
$\bigcup SS$	Verallgemeinerte Vereinigung, d.h. $\{x : X \mid (\exists S : SS \bullet x \in S)\}$
$\#S$	Anzahl der Elemente der endlichen Menge S
$\mathbb{N}$, $\mathbb{Z}$	die natürlichen bzw. ganzen Zahlen
$m \ldots n$	Intervall von m bis n, d.h. $\{k : \mathbb{N} \mid m \leq k \wedge k \leq n\}$

Z basiert formal auf den folgenden **Axiomen der Zermelo-Fraenkel-Mengenlehre, ZF**. (Wir führen hier die klassischen, ungetypten Axiome auf.)

1. Extensionalität

   $$\forall x, y.\ x = y \Leftrightarrow (\forall z.\ z \in x \Leftrightarrow z \in y)$$

   $x = y$ gilt genau dann, wenn x und y dieselben Elemente haben.

2. Null-Menge

   $$\exists x.\ \forall y.\ \neg\, y \in x$$

3. Paarmengen

   $$\forall x, y.\ \exists z.\ \forall u.\ u \in z \Leftrightarrow u = x \vee u = y$$

   Sind x, y Mengen, dann auch $\{x, y\}$.

4. Vereinigungen:

   $$\forall x.\ \exists y.\ \forall z.\ z \in y \Leftrightarrow (\exists w.\ z \in w \wedge w \in x)$$

   Ist x eine Menge, dann auch $\bigcup x$.

5. Potenzmenge

   $$\forall x.\ \exists y.\ \forall z.\ z \in y \Leftrightarrow (\forall w.\ w \in z \Rightarrow w \in x)$$

   Ist x eine Menge, dann auch $\mathbb{P}x$.

6. Unendlichkeit

   $$\exists x.\ (\exists y.\ y \in x) \wedge (\forall y.\ y \in x \Rightarrow \exists z.\ y \in z \wedge z \in x)$$

   Es gibt unendliche Mengen.

7. Regularität

   $$\forall x.\ (\exists y.\ y \in x) \Rightarrow (\exists y.\ y \in x \wedge \neg\ \exists z.\ z \in y \wedge z \in x)$$

   Jede nichtleere Menge ist disjunkt zu einem ihrer Elemente.

8. Teilmengen-Auszeichnung

   $$\forall x.\ \exists y.\ \forall z.\ z \in y \Leftrightarrow (z \in x \wedge G(z, u_1, \ldots, u_n))$$

   wobei G eine Formel ist, in der y nicht frei vorkommt. Dieses Axiom berechtigt formal zur Definition von Teilmengen einer gegebenen Grundmenge durch Angabe einer charakteristischen Eigenschaft:

   $$y = \{z \in x : G(z, u_1, \ldots, u_n)\}$$

9. Collection

   $$(\forall x.\ x \in u \Rightarrow \exists z.\ G(x, z, u, v_1, \ldots, v_n)) \Rightarrow$$
   $$(\exists y.\ \forall x.\ x \in u \Rightarrow \exists z.\ z \in y \wedge G(x, z, u, v_1, \ldots, v_n))$$

   wobei G eine Formel ist, in der y nicht auftritt. Ist für alle $x \in u$ die Klasse

   $$C_x = \{\, z \mid G(x, z, \ldots) \,\}$$

   nichtleer, dann gibt es eine Menge y, deren Durchschnitt mit jeder der Klassen nichtleer ist.

Die Axiome (8) und (9) sind keine einzelnen Axiome, sondern Schemata für unendlich viele Axiome.

Das *Teilmengen-* und das *Collection-Schema* werden häufig zu einem einzigen Axiomenschema, dem folgenden *Ersetzungsschema* kombiniert:

$$(\forall x.\ \exists_1 z.\ G(x, z, u, v_1, \ldots, v_n)) \Rightarrow (\exists y.\ \forall z.\ z \in y \Leftrightarrow \exists x.\ x \in u \wedge G(x, z, u, v_1, \ldots, v_n))$$

wobei G eine Formel ist, in der y nicht auftritt.

Ist $f(x)$ das einzige z, für das $G(x, z, \ldots)$ gilt, dann ist $\{f(x) : x \in u\}$ eine Menge.

Die Zermelo-Fraenkel Mengenlehre mit Auswahl hat die obigen 9 Axiome und

10. Auswahl

    $$\forall x.\ \exists y.\ y \text{ ist eine Funktion mit Definitionsbereich } x\ \wedge$$
    $$\forall z.\ (z \in x \wedge \exists u.\ u \in z) \Rightarrow y(z) \in z$$

    Jede Menge hat eine Auswahlfunktion.

7.2.3 Relationen

Relationen werden in Z als Mengen von Paaren dargestellt. Es gibt u.a. folgende Schreibweisen zur Bildung und Verknüpfung von Relationen:

$X \leftrightarrow Y$	Binäre Relationen zwischen X und Y, d.h. $\mathbb{P}(X \times Y)$
$x \underline{R} y$	x und y stehen in der Relation R, d.h. $(x,y) \in R$
$x \mapsto y$	„Maplet“ von x und y, äquivalent zu (x,y)
$\mathrm{dom}\, R$	Definitionsbereich von R, d.h. $\{x : X \mid (\exists y : Y \bullet xRy)\}$
$\mathrm{ran}\, R$	Wertebereich von R, d.h. $\{y : Y \mid (\exists x : X \bullet xRy)\}$
$R_1 \fatsemi R_2$	Relationenkomposition, $\{x : X;\ z : Z \mid (\exists y : Y \bullet x \underline{R_1}\, y \wedge y \underline{R_2}\, z) \bullet (x \mapsto z)\}$
R^{-1} (auch $R^{\sim}$)	Umkehrung von R, d.h. $\{y : Y;\ x : X \mid x \underline{R}\, y \bullet (y \mapsto x)\}$
$\mathrm{id}\, S$	Identitätsrelation von S, d.h. $\{x : S \bullet x \mapsto x\}$
$R(\mid S \mid)$	Relationenbild, d.h. $\{y : Y \mid (\exists x : S \bullet x \underline{R}\, y)\}$
$S \lhd R$	Einschränkung des Def.bereichs, $\{x : X;\ y : Y \mid x \in S \wedge x \underline{R}\, y \bullet (x \mapsto y)\}$
$S \lhd\!\!\!- R$	Antirestriktion des Def.bereichs, $\{x : X;\ y : Y \mid x \notin S \wedge x \underline{R}\, y \bullet (x \mapsto y)\}$
$R \rhd T$	Einschränkung des Wertebereichs, $\{x : X;\ y : Y \mid x \underline{R}\, y \wedge y \in T \bullet (x \mapsto y)\}$
$R \rhd\!\!\!- T$	Antirestriktion des Wertebereichs, $\{x : X;\ y : Y \mid x \underline{R}\, y \wedge y \notin T \bullet (x \mapsto y)\}$
$R_1 \oplus R_2$	Überschreiben von R_1 an allen Stellen, an denen R_2 definiert ist, $(\mathrm{dom}\, R_2 \lhd\!\!\!- R_1) \cup R_2$

Beispiel 76: Relationen

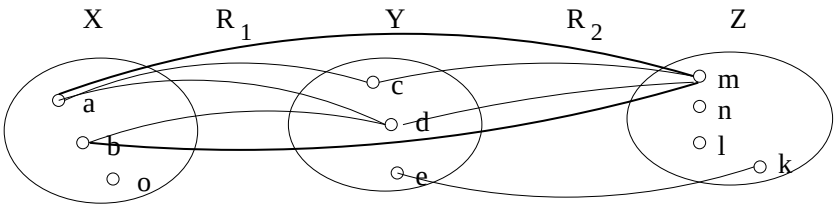


Abbildung 10: Beispiel für Relationen.

Für die in Abbildung 10 dargestellten Relationen $R_1 \subseteq X \times Y$ und $R_2 \subseteq Y \times Z$ gilt

$$\begin{aligned} \mathrm{dom}\, R_1 &= \{a,b\} \\ \mathrm{ran}\, R_1 &= \{c,d\} \\ R_1 \fatsemi R_2 &\subseteq X \times Z \\ R_1 \fatsemi R_2 &= \{(a,m),(b,n)\} \quad \text{da z.B. gilt:} (a,c) \in R_1, (c,m) \in R_2 \ \text{ sowie } \ (b,d) \in R_1, (d,m) \in R_2 \end{aligned}$$

◀

7.2.4 Funktionen

(Partielle) Funktionen werden als rechtseindeutige Relationen modelliert. Damit sind alle für Relationen eingeführten Schreibweisen auch auf Funktionen anwendbar. Verschiedene Pfeilsymbole kennzeichnen Bedingungen an Funktionen wie Injektivität, Surjektivität und endlichen Wertebereich. Typische Datenstrukturen wie Arrays oder Tabellen werden in Z abstrakt durch Funktionen und Relationen beschrieben.

$X \nrightarrow Y$	partielle Funktionen von X nach Y, $\{f : X \leftrightarrow Y \mid \forall x : X;\ y_1, y_2 : Y \bullet x\ \underline{f}\ y_1 \wedge x\ \underline{f}\ y_2 \Rightarrow y_1 = y_2\}$
$X \rightarrow Y$	totale Funktionen von X nach Y, $\{f : X \nrightarrow Y \mid \mathrm{dom} f = X\}$
$X \pfun Y$	endliche partielle Funktionen von X nach Y, $\{f : X \nrightarrow Y \mid \mathrm{dom} f \in \mathbb{F} X\}$
$X \rightarrowtail Y$	partielle injektive Funktionen von X nach Y, $\{f : X \nrightarrow Y \mid f^{-1} \in Y \nrightarrow X\}$
$X \fun Y$	totale injektive Funktionen von X nach Y, $(X \rightarrow Y) \cap (X \rightarrowtail Y)$
$X \twoheadrightarrow Y$	partielle surjektive Funktionen von X nach Y, $\{f : X \nrightarrow Y \mid \mathrm{ran} f = Y\}$
$X \rightarrow\!\!\!\!\!\rightarrow Y$	totale surjektive Funktionen von X nach Y, $(X \rightarrow Y) \cap (X \twoheadrightarrow Y)$
$X \bij Y$	Bijektionen von X nach Y, $(X \fun Y) \cap (X \rightarrow\!\!\!\!\!\rightarrow Y)$
$f\,x$, $f(x)$	Anwendung von Funktion f auf das Argument x, $\mu y : Y \mid x\ \underline{f}\ y$
$\lambda x : T \mid P \bullet t$	Lambda-Notation, $\{x : T \mid P \bullet x \mapsto t\}$

7.2.5 Sequenzen

Endliche Folgen über einer Menge X werden als partielle Funktionen $s : \mathbb{N} \nrightarrow X$ mit dem Definitionsbereich $1\,..\,n$ dargestellt, wobei n die Länge der Folge s ist (dies ist gleichzeitig die Anzahl der Paare, welche die Funktion s bilden).

$\mathrm{seq} X$	Sequenzen über X, $\{s : \mathbb{N} \nrightarrow X \mid \mathrm{dom} s = 1\,..\,\#s\}$
$\#s$	Länge von s (siehe # bei Mengen)
$\langle\rangle$	leere Sequenz ε, repräsentiert durch $\emptyset$
$\langle x_1, \ldots, x_n \rangle$	Aufzählung einer endlichen Folge, $\{(1 \mapsto x_1), \ldots, (n \mapsto x_n)\}$
$s \frown t$	Konkatenation von s und t, $s \cup \{i : 1\,..\,\#t \bullet (i + \#s \mapsto t(i))\}$

7.3 Grundschemata

Basierend auf diesen Grundlagen werden Schemata definiert.

Ein (Name für einen) Datentyp D wird eingführt durch

$[D]$

Ein *Schema* S wird syntaktisch notiert durch

S
$x_1 : T_1;\ \ldots;\ x_n : T_n$

P

wobei

- $x_1 : T_1;\ \ldots;\ x_n : T_n$ eine Menge von Deklarationen ist und
- P ein Prädikat, das neben $x_1, \ldots, x_n$ noch eine Menge G von globalen Variablen enthalten darf.

Die Semantik von S ist wie bei algebraischen Spezifikationen gegeben durch eine Signatur und eine Klasse von Modellen. Es werden nur „konkrete Datentypen“ definiert, deren Repräsentation mit Hilfe von Grundtypen wie $\mathbb{N}$ und Typoperatoren wie $\mathrm{seq}, \mathbb{P}, \times, \ldots$ gegeben ist. Es ist also keine Wahl bei

der Zuordnung von Namen zu Trägernamen möglich, die Typen (Sorten) der Signatur werden aus den Typisierungen bestimmt und nicht angegeben.

$$\begin{array}{rcl} \mathrm{Sig}(S) & =_{\mathrm{def}} & G \cup \{x_1 : T_1, \ldots, x_n : T_n\} \\ \mathrm{Mod}(S) & =_{\mathrm{def}} & \{A \in \mathrm{Struct}(\mathrm{Sig}(S)) \mid A \models P\} \end{array}$$

Jede Struktur A denotiert einen möglichen Zustand der Variablen des Schemas.

Beispiel 77: Grundschemata

1. Die Semantik des Schemas

   S_0
   $x : \mathbb{Z}$
   $y : \mathrm{seq}\,\mathbb{Z}$

   $x < \#y$

   ist gegeben durch

   $$\mathrm{Sig}(S_0) = \{x : \mathbb{Z}, y : \mathrm{seq}\,\mathbb{Z}\} \quad \text{mit Typen } \mathbb{Z}, \mathrm{seq}\,\mathbb{Z}$$
   $$\mathrm{Mod}(S_0) = \{A \in \mathrm{Struct}(\mathrm{Sig}(S_0)) \mid A \models x < \#y\}$$

   d.h. die Semantik besteht aus allen Strukturen über den Standardmodellen $\mathbb{Z}$ und $\mathbb{Z}^*$ der ganzen Zahlen und der endlichen Sequenzen ganzer Zahlen mit Belegungen ν der Variablen x und y, die die Bedingung $\nu(x) < \#\nu(y)$ erfüllen.

2. Das Schema

   T
   $z : 1..10$
   $x : \mathbb{N}$

   $x = z * z$

   ist Abkürzung für

   T
   $z : \mathbb{Z}$
   $x : \mathbb{Z}$

   $z \in 1..10$
   $x \in \mathbb{N}$
   $x = z * z$

   Bei der Signatur werden die Typen der Variablen betrachtet, nicht die Zustandsinformation, die dort kodiert ist.

   $$\mathrm{Sig}(T) = G \cup \{x : \mathbb{Z}, z : \mathbb{Z}\}$$
   $$\mathrm{Mod}(T) = \{A \in \mathrm{Struct}(T) \mid A \models z \in 1..10 \wedge x \in \mathbb{N} \wedge x = z^2\}$$

3. Geburtstagskalender:

   Die folgende Spezifikation ist ein Ausschnitt aus einem System, das Geburtstage speichert und an sie erinnert. Gegebene Grunddatentypen („Mengen“), die nicht näher spezifiziert werden, sind die Datentypen für „Name“ und „Datum“.

In Z werden durch

$$[NAME, DATE]$$

Basis-Datentypen *NAME* und *DATE* eingeführt, die als „bekannt“ vorausgesetzt werden.

Der Zustandsraum wird durch folgendes Schema beschrieben:

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\, birthday$$

Das Schema *BirthdayBook* führt zwei Zustandsvariablen *known* und *birthday* ein, deren Typen als Menge von Namen bzw. (partielle) Funktion von Namen nach (Geburts-)Daten deklariert werden. Außerdem wird die *Invariante*

$$known = \mathrm{dom}\, birthday$$

zugesichert, die in jedem Zustand des Systems gelten muss.

Bemerkung: *known* ist ein abgeleitetes Konzept, aber hilfreich für das Verständnis des Systems.

Ein möglicher Zustand ist

$$\langle\!| known = \{\text{“Martin”}, \text{“Thomas”}, \text{“Sabine”}\},$$
$$birthday = \{\text{“Martin”} \mapsto \text{“24. 12.”}, \text{“Thomas”} \mapsto \text{“8. 02.”}, \text{“Sabine”} \mapsto \text{“8. 02.”}\} \,|\!\rangle$$

(wobei die Grunddatentypen *NAME* und *DATE* durch Zeichenreihen dargestellt werden). Semantisch bezeichnet *BirthdayBook* eine Klasse von Strukturen A mit

$$\mathrm{Sig}(BirthdayBook) = G \cup (\{NAME, \mathbb{P}\, NAME, DATE\},$$
$$\{known : \mathbb{P}\, NAME, birthday : NAME \pfun DATE\})$$
$$\mathrm{Mod}(BirthdayBook) = \{A \mid A \models known = \mathrm{dom}\, birthday\}$$

◀

Ein Schema S kann auch aufgefasst werden als ein Record mit Selektoren $x_1 : T_1, \ldots, x_n : T_n$. Der Bindungsbereich der x_i ist das gesamte Schema, im Prädikat können auch globale Variablen und Werte verwendet werden. Umbenennung eines Schemas ergibt ein neues Schema.

So ist z. B.

$$S_1 = [a : \mathbb{Z};\ b : \mathrm{seq}\, \mathbb{Z} \mid a < \#b]$$

verschieden von S_0 .

Bei der Kombination von Schemata bleibt der Namensstamm erhalten (siehe 7.4). Hintereinanderfügen von Schemata erweitert die Umgebung um neue Schemanamen.

7.4 Schemakombination

Zum Aufbau von Spezifikationen aus Schemata bietet Z eine Reihe von Möglichkeiten. Schemata können erweitert und mit logischen Operatoren verknüpft werden. Außerdem können die Variablen in Schemata systematisch dekoriert werden und (dekorierte) Schemata sequentiell kombiniert werden.

7.4.1 Schemainklusion

Ein Schema R kann eine Erweiterung eines Schemas S sein; das Schema

$$\begin{array}{|l} \hline R \\ S \\ y_1 : R_1;\ \ldots;\ y_m : R_m \\ \hline P_1 \\ \hline \end{array}$$

bezeichnet das Schema

$$\begin{array}{|l} \hline R \\ x_1 : T_1;\ \ldots;\ x_n : T_n \\ y_1 : R_1;\ \ldots;\ y_m : R_m \\ \hline P \wedge P_1 \\ \hline \end{array}$$

Beispiel 78:

$$\begin{array}{|l} \hline R \\ S_0 \\ z : \mathbb{Z} \\ \hline z < x \\ \hline \end{array}$$

steht für

$$\begin{array}{|l} \hline R \\ x, z : \mathbb{Z} \\ y : \mathrm{seq}\,\mathbb{Z} \\ \hline x < \#y \wedge z < x \\ \hline \end{array}$$

◄

7.4.2 Logische Kombination von Schemata

1. Konjunktion

   $$S \wedge T \ =_{\text{def}}\ S \textbf{ and } T$$

   d.h.

   $$\begin{array}{rcl} \mathrm{Sig}(S \wedge T) & = & \mathrm{Sig}(S) \cup \mathrm{Sig}(T) \\ \mathrm{Mod}(S \wedge T) & = & \{ A \in \mathrm{Struct}(\mathrm{Sig}(S \wedge T)) \mid \\ & & \quad A|_{\mathrm{Sig}(S)} \in \mathrm{Mod}(S) \text{ und } A|_{\mathrm{Sig}(T)} \in \mathrm{Mod}(T) \} \end{array}$$

   $S \wedge T$ entspricht dem „Durchschnitt“ der Modelle (bei i.a. größerer Signatur).

   Beispiel 79: Die Konjunktion der Schemata S_0 und T entspricht dem folgenden Schema (allerdings sind “anonyme” Schemata in Z syntaktisch nicht vorgesehen).

$$S_0 \wedge T \quad = \quad \begin{array}{|l} \hline x : \mathbb{N} \\ y : \mathrm{seq}\,\mathbb{Z} \\ z : 1\,..\,10 \\ \hline x < \#y \wedge x = z * z \\ \hline \end{array}$$

◀

2. Disjunktion

   Die Disjunktion $S \vee T$ bezeichnet die „Vereinigung der Modelle“, formal:

$$\begin{array}{rcl} \mathrm{Sig}(S \vee T) & = & \mathrm{Sig}(S) \cup \mathrm{Sig}(T) \\ \mathrm{Mod}(S \vee T) & = & \{ A \in \mathrm{Struct}(\mathrm{Sig}(S \vee T)) \mid \\ & & \quad A|_{\mathrm{Sig}(S)} \in \mathrm{Mod}(S) \text{ oder } A|_{\mathrm{Sig}(T)} \in \mathrm{Mod}(T) \} \end{array}$$

   Beispiel 80:

$$S_0 \vee T \quad = \quad \begin{array}{|l} \hline x : \mathbb{Z} \\ y : \mathrm{seq}\,\mathbb{Z} \\ z : 1\,..\,10 \\ \hline x < \#y \vee (x \in \mathbb{N} \wedge x = z * z) \\ \hline \end{array}$$

◀

3. Negation

   Das Schema $\neg\, S$ repräsentiert das „Komplement der Modelle unter Beibehaltung der Typen“, formal:

$$\begin{array}{rcl} \mathrm{Sig}(\neg\, S) & = & \mathrm{Sig}(S) \\ \mathrm{Mod}(\neg\, S) & = & \{ A \in \mathrm{Struct}(\mathrm{Sig}(S)) \mid A \notin \mathrm{Mod}(S) \} \end{array}$$

   Beispiel 81:

$$\neg\, T \quad = \quad \begin{array}{|l} \hline x : \mathbb{Z} \\ z : \mathbb{Z} \\ \hline x \notin \mathbb{N} \vee z \notin \mathbb{N} \vee z \notin 1\,..\,10 \vee x \neq z * z \\ \hline \end{array}$$

◀

4. Quantifizierung

   Durch Quantifizierung werden Variablen verborgen. Das Schema

   $$Q x_1 : T_1;\ \ldots;\ x_k : T_k \mid P \bullet S$$

   (wobei $k < n$, d.h. die Variablen $x_1, \ldots, x_k$ treten in S auf und haben den gleichen Typ) entspricht dem Schema

$$\begin{array}{|l} \hline x_{k+1} : T_{k+1};\ \ldots;\ x_n : T_n \\ \hline Q x_1 : T_1;\ \ldots;\ x_k : T_k \mid P \bullet S \\ \hline \end{array}$$

Beispiel 82:

$$\exists z : \mathbb{N} \mid z > 5 \bullet T \quad = \quad \begin{array}{|l} \hline x : \mathbb{N} \\ \hline \exists z : \mathbb{N} \mid z > 5 \bullet z \in 1..10 \wedge x = z * z \\ \hline \end{array}$$

Das Schema kann vereinfacht werden zu

$$\begin{array}{|l} \hline x : \mathbb{N} \\ \hline \exists z : 6..10 \bullet x = z * z \\ \hline \end{array}$$

◀

Falls über alle deklarierten Variablen eines Schemas S quantifiziert wird, schreibt man

$$QS \bullet T$$

als Abkürzung von

$$Qx_1 : T_1; \ldots; x_n : T_n \mid P \bullet T$$

wobei

$$\begin{array}{|l} \hline S \\ \hline x_1 : T_1; \ldots; x_n : T_n \\ \hline P \\ \hline \end{array}$$

5. Export und Verbergen von Symbolen

   (a) $S \upharpoonright (x_1, \ldots, x_k)$

   (b) $S \setminus (x_1, \ldots, x_k)$

   sind Schemata, die die Signatur von S beschränken. In (5a) geschieht dies durch *Beschränkung auf* $x_1, \ldots, x_k$ und in (5b) durch *Verstecken von* $x_1, \ldots, x_k$, wobei jeweils $x_1, \ldots, x_k$ in S deklariert sein müssen.

   Diese Operationen können unter Verwendung der Quantifizierung definiert werden durch

$$\begin{array}{lll} S \upharpoonright (x_1, \ldots, x_k) & =_{\text{def}} & \exists x_{k+1} : T_{k+1}; \ldots; x_n : T_n \bullet S \\ S \setminus (x_1, \ldots, x_k) & =_{\text{def}} & \exists x_1 : T_1; \ldots; x_k : T_k \bullet S \end{array}$$

   (Dabei seien $x_1 : T_1, \ldots, x_n : T_n$ alle im Schema S deklarierten Zustandsvariablen.)

Beispiel 83:

$$T \upharpoonright x \quad = \quad \exists z : \mathbb{N} \bullet T \quad = \quad \begin{array}{|l} \hline x : \mathbb{N} \\ \hline \exists z : \mathbb{N} \mid z \in 1..10 \bullet x = z * z \\ \hline \end{array}$$

◀

7.4.3 Dekorationen

Die Identifikatoren in Schemas können systematisch mit Beistrichen, Indizes, usw. dekoriert werden. Es ist S' gleich

$$\begin{array}{|l} S' \\ \hline x'_1 : T_1;\ \ldots;\ x'_n : T_n \\ \hline P[x'_1/x_1, \ldots, x'_n/x_n] \\ \hline \end{array}$$

Beispiel 84:

$$\begin{array}{|l} S'_0 \\ \hline x' : \mathbb{Z} \\ y' : \mathrm{seq}\,\mathbb{Z} \\ \hline x' < \#y' \\ \hline \end{array}$$

◀

Damit lassen sich für jedes Schema Zustandsfolgen $S, S', S'', S''', \ldots$ definieren, die ausgehend von Schema S, das die Klasse des Anfangszustands definiert, jeweils die Klassen der möglichen Nachfolgezustände definieren.

Semantik von S':

$$\begin{array}{rcl} \mathrm{Sig}(S') & = & \{x' : T \mid x : T \in \mathrm{Sig}(S)\} \cup \mathrm{Sig}(\mathit{Basis}) \\ \mathrm{Mod}(S') & = & \{A \in \mathrm{Struct}(\mathrm{Sig}(S')) \mid \exists B \in \mathrm{Mod}(S) : A|_{\mathit{copy}} = B\}, \end{array}$$

wobei $\mathit{copy}(x) = x'$ für alle $x : T \in \mathrm{Sig}(S)$ der Signaturmorphismus ist, der alle deklarierten Symbole von S mit „ $'$ “ dekoriert. Eine äquivalente Formulierung der Semantik ist

$$S' = S \textbf{ with } \mathit{copy}$$

Basis bezeichnet alle Grunddatenstrukturen mit den gegebenen Operationen.

Zustandsübergänge: $\Delta S \ =_{\mathrm{def}} \ S \wedge S'$

Im allgemeinen denotiert hier das Schema S einen Zustandsraum eines Abstrakten Datentyps (ADT). ΔS ist implizit definiert als die Kombination von S mit einem Nachfolgezustand S' des ADT. Jede Operation des ADT kann durch Erweiterung von ΔS durch Dekoration der Ein- und Ausgabevariablen und durch prädikatenlogische Spezifikation der Vor- und Nachbedingungen spezifiziert werden.

Jedes Modell von ΔS hat die Signatur $\mathrm{Sig}(S) \cup \mathrm{Sig}(S')$, d.h. sei $\mathrm{Sig}(S) = \{x_1 : T_1, \ldots, x_n : T_n\} \cup \mathrm{Sig}(\mathit{Basis})$. Dann gilt

$$\begin{array}{rcl} \mathrm{Sig}(\Delta S) & = & \{x_1 : T_1, \ldots, x_n : T_n, x'_1 : T_1, \ldots, x'_n : T_n\} \cup \mathrm{Sig}(\mathit{Basis}) \\ \mathrm{Mod}(\Delta S) & = & \{A \in \mathrm{Struct}(\mathrm{Sig}(\Delta S)) \mid A|_{\mathrm{Sig}(S)} \in \mathrm{Mod}(S) \text{ und } A|_{\mathrm{Sig}(S')} \in \mathrm{Mod}(S')\} \end{array}$$

Äquivalent dazu lässt sich ΔS als ein Zustandsübergang auffassen, d.h. jedes Element von $\mathrm{Mod}(\Delta S)$ besteht aus einem Paar $\langle B, B' \rangle$ von Algebren, auch geschrieben als

$$B \rightarrow_S B'$$

wobei $B = A|_{\mathrm{Sig}(S)}$ und $B' = A|_{\mathrm{Sig}(S')}$ für ein $A \in \mathrm{Mod}(\Delta S)$ gilt.

In ΔS ist der Zusammenhang zwischen B und B' nur durch die Axiome von S gegeben, d.h. B und B' können beliebige Modelle sein (modulo Umbenennung). Durch Angabe weiterer Axiome (wie in *AddBirthday*, vgl. Bsp. 86) werden die Resultate der zu spezifizierenden Operation festgelegt.

Abkürzung: $\Xi S = \Delta S \wedge \bigwedge_{i=1,\ldots,n} x_i = x_i'$

ΞS ist eine Abkürzung dafür, dass die Werte der durch S deklarierten Variablen im Nachfolgezustand nicht verändert werden.

Bemerkung: Eines der wichtigen Kennzeichen imperativer Programmiersprachen sind lokale Änderungen einer einzigen Variable, ausgedrückt durch „$x := e$“. Um den Effekt dieser Zuweisung zu spezifizieren, müssen im allgemeinen n Gleichungen angegeben werden.

Sei $\mathrm{Sig}(S) = \{x_1 : T_1, \ldots, x_n : T_n\}$. Dann gilt als Nachbedingung von $x_i := e$:

$$x_1' = x_1 \wedge \ldots \wedge x_i' = e \wedge \ldots \wedge x_n' = x_n$$

Es müssen also n Axiome angegeben werden, während eine einzige Anweisung genügt. Um auch in der Spezifikation kürzere Beschreibungsmöglichkeiten zur Verfügung zu haben, wurde in Z das Konstrukt ΞS eingeführt.

Beispiel 85: Zähler

Zustände und Operationen in Z — Spezifikation des Zustandsraumes:

Counter

$value, limit : \mathbb{N}$

$value < limit$

Definiere Operation *Inc* (increment):

Inc

$\Delta Counter$

$value' = value + 1$

$limit' = limit$

Initialer Zustand:

InitCounter

Counter

$value = 0 \wedge limit = 100$

Erfüllbarkeitsbedingung für Schema *InitCounter*:

$$\exists Counter \bullet InitCounter \quad \equiv$$
$$\exists value, limit : \mathbb{N} \bullet value < limit \wedge value = 0 \wedge limit = 100$$

Addieren (mit Ein- / Ausgabe):

Add

$\Delta Counter$

$jump? : \mathbb{N}, new\_value! : \mathbb{N}$

$value' = value + jump?$

$limit' = limit$

$new\_value! = value'$

◄

Beispiel 86: Operationen auf dem Geburtstagkalender (vgl. Bsp. 77)

Das folgende Schema beschreibt die Operation *AddBirthday*, die auf Eingabe von zwei Werten den Zustand von *Birthday* – *Book* ändert.

Dabei bezeichnen

known, *birthday*	Zustandsbeobachtung vor der Zustandsänderung
known′, *birthday*′	Zustandsbeobachtung nach der Zustandsänderung
name?, *date*?	Eingabevariablen.

BirthdayBook wird also importiert und geändert. Hierfür schreiben wir Δ*BirthdayBook*. Das Fragezeichen steht konventionsgemäß für Eingabe, der Beistrich ′ für den Wert nach Ausführung einer Operation.

AddBirthday

$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

Folgende Eigenschaft lässt sich beweisen:

$$known' = known \cup \{name?\}$$

Beweis.

$$\begin{array}{lll}
& known' & \\
= & \mathrm{dom}\, birthday' & [\text{ Invariante von } Birthday'] \\
= & \mathrm{dom}(birthday \cup \{name? \mapsto date?\}) & [\text{Spezifikation } AddBirthday] \\
= & \mathrm{dom}\, birthday \cup \mathrm{dom}\{name? \mapsto date?\} & [\text{ Mengenlehre }] \\
= & \mathrm{dom}\, birthday \cup \{name?\} & [\text{ Eigenschaft von dom }] \\
= & known \cup \{name?\} & [\text{ Invariante von } Birthday]
\end{array}$$

Verwendete mathematische Eigenschaften:

$$\begin{array}{rcl}
\mathrm{dom}(f \cup g) & = & (\mathrm{dom}\, f) \cup (\mathrm{dom}\, g) \\
\mathrm{dom}\{a \mapsto b\} & = & \{a\}
\end{array}$$

Q.E.D.

Semantisch beschreibt *AddBirthday* einen Zustandsübergang von $\mathrm{Sig}(BirthdayBook)$-Algebren in Nachfolgealgebren mit Eingabevariablen *name*? und *date*?:

$$A \in \mathrm{Mod}(BirthdayBook) \rightarrow_{AddBirthday} A'$$

wobei in A' gilt:

$$\begin{array}{rcl}
known^{A'} & = & \mathrm{dom}\, birthday^{A'} \\
birthday^{A'} & = & birthday^{A} \cup \{name?^{A'} \mapsto date?^{A'}\} \\
date?^{A'} & & \text{„irgendein Element“ der Trägermenge } Date^{A'} = Date^{A} \\
name?^{A'} & & \text{„irgendein Element } \notin known^{A}\text{“ der Trägermenge } Name^{A'} = Name^{A}
\end{array}$$

Die folgenden Schemata beschreiben Abfragen von Daten aus dem Geburtstagskalender. Ausgabevariablen werden dabei durch ein Ausrufezeichen gekennzeichnet. Diese Operationen ändern den Zustand von *BirthdayBook* nicht. Dies wird ausgedrückt durch „Ξ*BirthdayBook*“:

$$\Xi BirthdayBook \;\equiv\; \Delta BirthdayBook \wedge known' = known \wedge birthday' = birthday$$

FindBirthday

$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$
$date! = birthday(name?)$

Ähnlich ist die Operation *Remind* definiert. Die Mengenklammern im Axiom von *Remind* bezeichnen die „Mengenkomprehension“.

Remind

$\Xi BirthdayBook$
$today? : DATE$
$cards! : \mathbb{P}\, NAME$

$cards! = \{ n : NAME \mid n \in known \wedge birthday(n) = today? \}$

Der Anfangszustand wird durch das Schema *InitBirthdayBook* beschreiben. Da $known = \emptyset$ vorausgesetzt wird, folgt aus der Schema-Invariante

$$known = \mathrm{dom}\, birthday$$

dass *birthday* die total undefinierte Funktion ist (der Graph ist leer).

InitBirthdayBook

$BirthdayBook$

$known = \emptyset$

◀

Sequentielle Komposition: Zwei Schemata S_1 und S_2 können sequentiell durch $S_1 \fatsemi S_2$ kombiniert werden. Dabei wird der Nachfolgezustand von S_1 mit dem Anfangszustand von S_2 identifiziert. Endzustand ist der daraus entstehende Endzustand von S_2, d.h.

$$A \rightarrow_{S_1} A'', A'' \rightarrow_{S_2} A'' \quad \Rightarrow \quad A \rightarrow_{S_1 \fatsemi S_2} A'$$

Formal: Seien S_1 und S_2 über der gleichen Signatur Σ definiert, dann gilt :

$$S_1 \fatsemi S_2 \;=_{\mathrm{def}}\; \exists S'' \bullet S_1[S''/S'] \wedge S_2[S''/S]$$

Gemeinsame Schemakomponenten (z.B. Ein- und Ausgabevariablen) von S_1 und S_2 werden identifiziert.

Beispiel 87: Komposition von Zähleroperationen

1. $Inc \fatsemi Inc \quad = \quad \exists value'', limit'' : \mathbb{N} \bullet$

$$\begin{array}{|l} \Delta Counter \\ \hline value'' = value + 1 \\ limit'' = limit \\ value' = value'' + 1 \\ limit' = limit'' \end{array}$$

   Dies ist äquivalent zum Schema

$$\begin{array}{|l} Inc \fatsemi Inc \\ \hline \Delta Counter \\ \hline value' = value + 2 \\ limit' = limit \end{array}$$

2. Die Operation $Inc \fatsemi Add$ ist äquivalent zum Schema

$$\begin{array}{|l} Inc \fatsemi Add \\ \hline \Delta Counter \\ jump? : \mathbb{N}, new\_value! : \mathbb{N} \\ \hline value' = value + jump? + 1 \\ limit' = limit \\ new\_value! = value' \end{array}$$

3. Dagegen ist die Komposition $Add \fatsemi Inc$ äquivalent zum folgenden Schema; der Wert der Ausgabevariablen ist also der neue Zählerstand minus 1.

$$\begin{array}{|l} Add \fatsemi Inc \\ \hline \Delta Counter \\ jump? : \mathbb{N}, new\_value! : \mathbb{N} \\ \hline value' = value + jump? + 1 \\ limit' = limit \\ new\_value! = value + jump? \quad (= value' - 1) \end{array}$$

◀

7.5 Zusammenfassung

- Z ist eine modell-orientierte Spezifikationssprache für zustandsbasierte Systeme. Ausgangspunkt sind konkrete Rechenstrukturen (natürliche Zahlen, Mengen, Relationen, Funktionen, Sequenzen).
- Systemzustände werden durch Z-Schemata beschrieben, die Zustandsvariablen und eine *Zustandsinvariante* einführen, die in allen möglichen Zuständen gelten muss.

$$\begin{array}{|l} S \\ \hline x_1 : T_1; \ldots; x_n : T_n \\ \hline P \end{array} \qquad \text{bzw. äquivalent} \qquad S \mathrel{\widehat{=}} [x_1 : T_1; \ldots; x_n : T_n \mid P]$$

  Einführung von Basis-Sorten: $[SORTE]$, z.B $[ADDRESS]$.

- Kombination von Schemata

aussagenlogische Verknüpfungen	$\wedge, \vee, \neg$
Quantifizierung	$\exists \vec{x} : \vec{T} \bullet S, \quad \forall \vec{x} : \vec{T} \bullet S$
Export/Verstecken	$S \upharpoonright (\vec{x}), \quad S \setminus (\vec{x})$

- Dekoration von Namen mit „ $'$ “ (Nachzustand), „ ? “ (Eingabe) oder „ ! “ (Ausgabe)
- Spezifikation von Operationen

$$\begin{aligned} \Delta S &\equiv S \wedge S' \\ \Xi S &\equiv \Delta S \wedge \theta S = \theta S' \end{aligned}$$

- Sequentielle Komposition $S_1 \fatsemi S_2$

  macht den Zustandsübergang von vor S_1 bis ans Ende von S_2 atomar.

$$S_1 \fatsemi S_2 \equiv \exists \Sigma'' \bullet S_1[\Sigma''/\Sigma'] \wedge S_2[\Sigma''/\Sigma]$$

  wobei Σ die gemeinsame Signatur von S_1 und S_2 ist.

8 Spezifikationsentwicklung in Z

In diesem Abschnitt betrachten wir einen Ansatz zur Spezifikationsentwicklung in Z, der wie bei algebraischen Spezifikationen auf Verfeinerungsrelationen beruht. In Z unterscheidet man zwischen der Verfeinerung von Operationen, der Verfeinerung von Daten und allgemein der Verfeinerung von Schemata.

8.1 Verfeinerung von Operationen

Zur Verfeinerung von Operationen betrachtet man Schemata folgender Form, die durch Beschreibung von Zustandsübergängen das Verhalten von Prozeduren charakterisieren.

OP

$\Delta State$
$x? : Input$
$y! : Output$
$P(s, x?, s', y!)$

wobei s, s' Variablen der Sorte State sind. In sequentieller Notation lautet das Schema *OP*:

$$OP \mathrel{\widehat{=}} [\Delta State;\ x? : Input;\ y! : Output \mid P(s, x?, s', y!)]$$

Bemerkung: Im allgemeinen sind s und s' von verschiedener Sorte $State_1$ und $State_2$, wobei $State_1$ und $State_2$ wieder Records sind, z.B. $State_1 = \{x : \mathbb{N}, y : \mathrm{seq}\,\mathbb{N}\}$, $State_2 = \{x : \mathbb{N}\}$; die Vereinfachung $State_1 = State_2 = State$ ist aber ausreichend für die weiteren Beispiele.

Falls mehrere Eingabe- oder Ausgabevariable vorkommen, schreiben wir ebenso $x?$ bzw. $y!$ für Vektoren von Ein- bzw. Ausgabevariablen.

Die Semantik von *OP* lässt sich durch die Relation charakterisieren, die zwischen Ein- und Ausgabevariablen sowie Vor- und Nachzustand besteht.

$$[\![OP]\!] = \{\, \langle \langle s, x? \rangle, \langle s', y! \rangle \rangle \in (State \times Input) \times (State \times Output) \mid P(s, x?, s', y!) \,\}$$

Im allgemeinen wird durch *OP* eine Relation beschrieben, die für jede Eingabe mehrere mögliche Nachzustände und Ausgabewerte zulässt, d.h. *OP* kann als eine nichtdeterministische Funktion aufgefasst werden.

Beispiel 88: Sei folgende Spezifikation gegeben.

$$\begin{array}{|l} S \\ \hline x?, y! : \mathbb{N} \\ \hline (0 < x? < 3) \wedge (y! < x? + 1) \\ \hline \end{array}$$

bzw. in sequentieller Notation

$$S \mathrel{\widehat{=}} [x?, y! : \mathbb{N} \mid (0 < x? < 3) \wedge (y! < x? + 1)]$$

Dann ist

$$[\![S]\!] \;=\; \{\, (x?, y!) : \mathbb{N} \times \mathbb{N} \;\mid\; 0 < x? < 3 \wedge y! < x? + 1 \,\}$$

Für die Eingabe $x? = 2$ sind als Ausgabe $y!$ die Werte 0, 1, 2 möglich. Die Eingabe $x? = 7$ ist nicht korrekt.

Folgendes Schema ist eine Verfeinerung von S:

$$\begin{array}{|l} R_1 \\ \hline x?, y! : \mathbb{N} \\ \hline x? < 5 \wedge y! = x \\ \hline \end{array}$$

R_2 hingegen ist keine Verfeinerung von S, sondern S Verfeinerung von R_2

$$\begin{array}{|l} R_2 \\ \hline x?, y! : \mathbb{N} \\ \hline (0 < x? < 3) \wedge y! < (x?)^2 + 1 \\ \hline \end{array}$$

R_3 und S sind unvergleichbar:

$$\begin{array}{|l} R_3 \\ \hline x?, y! : \mathbb{N} \\ \hline (x? = 1) \wedge y! = x? \\ \hline \end{array}$$

◀

Idee der Verfeinerung:

1. Die Relation wird deterministischer (weniger nichtdeterministisch).
2. Der Definitionsbereich der Verfeinerung umfasst den Definitionsbereich der abstrakten Relation.

R ist Operationsverfeinerung von S, falls

1. Definitionsbereich von S ist enthalten im Definitionsbereich von R.
2. R ist deterministischer als S, falls beide Relationen definiert sind.

Jede Implementierung R von S in einer konventionellen Programmiersprache ist notwendigerweise *deterministisch*, d.h. für jede Eingabe gibt es genau ein Ergebnis (nämlich einen Wert oder „undefiniert“, wenn R nicht terminiert), und R sollte alle Eingabewerte von S akzeptieren.

R ist eine *operationelle (oder relationale) Verfeinerung* eines Schemas S, wenn R alle Eingaben von S akzeptiert (d.h. $\mathrm{dom}\, S \subseteq \mathrm{dom}\, R$) und wenn R determinierter als S ist auf den Eingaben von S (d.h. $((\mathrm{dom}\, S) \lhd R) \subseteq S$).

Definition 45 *Seien Input, Output, State Mengen und R, S* $\subseteq (State \times Input) \times (State \times Output)$.

R verfeinert *S* operational, *wenn*

1. $\mathrm{dom}\, S \subseteq \mathrm{dom}\, R$ *und*
2. $((\mathrm{dom}\, S) \lhd R) \subseteq S$) *gilt.*

Dies lässt sich auch folgendermaßen ausdrücken: Sei für die Relation

$$T \subseteq M_1 \times M_2$$

das *charakteristische Prädikat*

$$\mathrm{pre}\ T$$

des *Definitionsbereichs* folgendermaßen definiert:

$$\begin{array}{lll} (\mathrm{pre}\ T)(a) & =_{\mathrm{def}} & a \in M_1 \wedge \exists b \in M_2 : a\ \underline{T}\ b \\ (\mathrm{pre}\ T)(a) & \Leftrightarrow & a \in \mathrm{dom}\, T \end{array}$$

Dann ist *R operationale Verfeinerung* von *S*, wenn gilt:

1. $\mathrm{pre}\ S \Rightarrow \mathrm{pre}\ R$, d.h.

   $$\forall a, x? : (a, x?) \in \mathrm{dom}\, S \Rightarrow (a, x?) \in \mathrm{dom}\, R$$

2. $\mathrm{pre}\ S \wedge R \Rightarrow S$, d.h.

   $$\forall a, x?, b, y! : (a, x?) \in \mathrm{dom}\, S \wedge (a, x?)\ \underline{R}\ (b, y!) \Rightarrow (a, x?)\ \underline{S}\ (b, y!)$$

Beispiel 89: Eine Verfeinerung von *BirthdayBook*.

Eine korrekte Implementierung von *AddBirthday* trägt Geburtstage ein, solange die Eingabe korrekt ist. Die Spezifikation verbietet es aber, einen Geburtstag für einen schon bekannten Namen einzutragen.

Im folgenden wird eine Verfeinerung angegeben, die mit Hilfe der Z-Schema-Operationen in die erste Spezifikation integriert werden kann. Wir führen eine zusätzliche Ausgabevariable *result*! ein, die drei verschiedene Werte annehmen kann: *ok*, *already_known*, *not_known*.

Die vorhergehende Beschreibung wird mit zwei neuen Schemata kombiniert. Zunächst definieren wir den Aufzählungstyp

$$REPORT ::= ok \mid already\_known \mid not\_known$$

und die zwei Hilfsschemata

Sucess

$result! : REPORT$
$result! = ok$

AlreadyKnown

$\Xi BirthdayBook$
$name? : NAME$
$result! : REPORT$
$name? \in known$
$result! = already\_known$

Für die Kombination stehen in Z die „logischen“ Operatoren $\wedge$ und $\vee$ zur Verfügung. Das neue Schema

$$RAddBirthday \mathrel{\widehat{=}} (AddBirthday \wedge Success) \vee AlreadyKnown$$

ist für alle Eingaben definiert. Dieses strukturierte Schema ist äquivalent zu folgenden einfachen Schema.

RAddBirthday

$\Delta Birthday$
$name? : NAME$
$date? : DATE$
$result! : REPORT$

$(name? \notin known \wedge birthday' = birthday \cup \{name? \mapsto date?\} \wedge result! = ok) \quad \vee$
$(name? \in known \wedge birthday' = birthday \wedge result! = already\_known)$

Bemerkung: Die Variable *known* muss in *RAddBirthday* nicht axiomatisiert werden, da ihr Wert wegen der Invariante

$$known = \mathrm{dom}\, birthday$$

aus den anderen Axiomen folgt.

Eine robuste Version von *FindBirthday* und *Remind* erhält man unter Verwendung des Hilfsschemas

NotKnown

$\Xi BirthdayBook$
$name? : NAME$
$result! : REPORT$

$name? \notin known$
$result! = not\_known$

durch die Definition

$$RFindBirthday \mathrel{\widehat{=}} (FindBirthday \wedge Success) \vee NotKnown$$
$$RRemind \mathrel{\widehat{=}} Remind \wedge Success$$

◄

Satz 46

1. *RAddBirthday ist operationale Verfeinerung von AddBirthday.*
2. *RFindBirthday ist operationale Verfeinerung von FindBirthday.*
3. *RRemind ist operationale Verfeinerung von Remind.*

Beweis. (Skizze)

zu 1.: Offensichtlich umfasst der Definitionsbereich von *RAddBirthday* den Definitionsbereich von *Addbirthday*.

$$\mathrm{pre}\ AddBirthday \Rightarrow \mathrm{pre}\ RAddBirthday$$

und es gilt

	pre *AddBirthday* $\wedge$ *RAddBirthday*	
$\equiv$	$name? \notin known \wedge$ *RAddBirthday*	[Def. pre *AddBirthday*]
$\Rightarrow$	$\Delta BirthdayBook \wedge name? \notin known$	
	$\wedge\ birthday' = birthday \cup \{name? \mapsto date?\}$	[Def. *RAddBirthday*, Aussagenlogik]
$\Rightarrow$	*AddBirthday*	[Def. *AddBirthday*] Q.E.D.

8.2 Wechsel der Datenstruktur

Im allgemeinen werden bei der Spezifikationsentwicklung die „abstrakten“ Datentypen der Spezifikation durch „konkrete“ Datentypen ersetzt, die näher an den Datentypen der gewählten Programmiersprache sind.

Bei einer Verfeinerung muss zwischen der „abstrakten“ Spezifikation *AS* und der „konkreten“ Spezifikation *KS* eine Relation $R \subseteq AS \times KS$ bestehen.

Außerdem müssen natürlich auch Verträglichkeitsbedingungen zwischen den initialen Zuständen von *AS* und *KS* sowie zwischen den Operationen *AOP* und *KOP* von *AS* und *KS* bestehen (darüber später in 8.3).

Beispiel 90:

1. Repräsentation von 2-dimensionalen Matrizen durch 1-dimensionale Vektoren:

   MATRIX
   $a : (1..r) \times (1..s) \to \mathbb{N}$

   VEKTOR
   $c : (1..r * s) \to \mathbb{N}$

   Die Repräsentationsrelation kann folgendermaßen definiert werden:

   $$R \mathrel{\widehat{=}} [MATRIX;\ VEKTOR \mid \forall i : 1..r;\ j : 1..s \bullet a(i,j) = c((i-1) * s + j)]$$

   Diese Repräsentation ist bijektiv.

2. Repräsentation von Bags durch Sequenzen

   Ein Bag ist eine Multimenge, d.h. die Häufigkeit der Elemente wird gezählt, die Reihenfolge spielt jedoch keine Rolle.

   Z.B. gilt für die Bags

   $$b_1 = [\![0,1,1]\!], \quad b_2 = [\![0,1]\!], \quad b_3 = [\![1,0,1]\!]$$

   $b_1 = b_3$, aber $b_1 \neq b_2$.

   In Z wird jeder Bag b mit Elementen vom Typ X durch eine partielle Funktion $b : X \nrightarrow \mathbb{N}$ dargestellt. Zum Beispiel gilt

   $$b_1 : \mathbb{N} \nrightarrow \mathbb{N} \text{ mit } b_1(0) = 1, b_1(1) = 2, b_1(x) \text{ undefiniert für } x > 1$$

   Konstruktoren sind der leere Bag $[\![]\!]$, einelementige Bags $[\![x]\!]$ und die Vereinigung $\uplus$ von Bags; fragen kann man nach der Anzahl $count(b,x)$ der Vorkommen von x in b.

Bags werden häufig durch ungeordnete Sequenzen natürlicher Zahlen repräsentiert, wobei beim Hinzufügen eines Elementes n dieses Element an die Sequenz angehängt wird. Formal definieren wir die Repräsentationsrelation $R \subseteq \mathrm{bag}\,\mathbb{N} \times \mathrm{seq}\,\mathbb{N}$ mit

$$[\![b_1,\ldots,b_n]\!]\ \underline{R}\ \langle c_1,\ldots,c_k\rangle \text{ gdw. } k=n \text{ und } c_1,\ldots,c_k \text{ ist eine Permutation von } b_1,\ldots,b_n$$

Jeder Bag $[\![b_1,\ldots,b_n]\!]$ mit $n \geq 2$ hat mehrere Repräsentanten, z.B. hat

$[\![0,1]\!]$	die Repräsentanten	$\langle 0,1\rangle$ und $\langle 1,0\rangle$
$[\![0,1,1]\!]$	die Repräsentanten	$\langle 0,1,1\rangle$, $\langle 1,0,1\rangle$ und $\langle 1,1,0\rangle$

Umgekehrt hat jeder Repräsentant genau einen zugeordneten Bag.

3. Repräsentation von Bags von Bitsequenzen mit Paritycheck durch Sequenzen von Paritychecks

   Betrachte folgende Spezifikation eines Systems zum Speichern von Bitfolgen mit Zustands-Schema *AS*, Anfangszustand *InitAS* und einer Operation *AOP* zur Eingabe einer Bitfolge mit Ausgabe der Quersumme modulo 2 der eingegebenen Folge.

AS

$a : \mathrm{bag}\,\mathrm{seq}\{0,1\}$

InitAS

AS

$a = [\![]\!]$

AOP

ΔAS
$x? : \mathrm{seq}\{0,1\}$
$y! : \{0,1\}$

$a' = a \uplus [\![x?]\!]$
$y! = \Sigma_2(x?)$

Hierbei sei $\Sigma_2(x?) \ =_{\mathrm{def}}\ (x?(1)+\ldots+x?(\#x?)) \bmod 2$.

Die Implementierung speichert lediglich eine Folge von Quersummen anstelle der Folge der eingegebenen Folgen.

CS

$c : \mathrm{seq}\{0,1\}$

InitCS

CS

$c = \langle\rangle$

COP

ΔCS
$x? : \mathrm{seq}\{0,1\}$
$y! : \{0,1\}$

$c' = c \frown \langle \Sigma_2(x?)\rangle$
$y! = \Sigma_2(x?)$

Die Repräsentationsrelation R setzt jedes Element eines Bags mit seinem Paritycheck in Beziehung. Eine Sequenz c mit Elementen aus $\{0,1\}$ repräsentiert einen Bag a , wenn die Anzahl der „Einsen“ in c gleich der Anzahl der Elemente von a ist, die einen Paritycheck 1 besitzen, und die Anzahl der „Nullen“ in c gleich der Anzahl der Elemente von a ist, die einen Paritycheck 0 besitzen.

R

AS

CS

$\forall j \in \{0,1\} \bullet \#(c \upharpoonright \{j\}) = sum\{x : \mathrm{dom}\, a \mid \Sigma_2(x) = j \bullet x \mapsto a(x)\}$

wobei $sum\, b$ die Anzahl der Elemente eines Bags (mit ihrer Häufigkeit) zählt:

$[X]$

$sum : \mathrm{bag}\, X \to \mathbb{N}$

$sum\, [\![]\!] = 0$

$sum(b \uplus [\![x]\!]) = sum(b) + 1$

Die Relation R erlaubt, dass abstrakte Elemente mehrere konkrete Repräsentationen haben und umgekehrt. Zum Beispiel werden beide „abstrakten“ Folgen

$$a_1 = \langle\langle 0\rangle, \langle 0,1\rangle\rangle \quad a_2 = \langle\langle 0,0\rangle, \langle 0,1\rangle\rangle$$

durch die beiden „konkreten“ Folgen

$$c_1 = \langle 0,1\rangle \quad c_1 = \langle 1,0\rangle$$

repräsentiert. ◀

8.3 Verifikationsbedingungen für Verfeinerungen

Erinnerung: C operationelle Verfeinerung von A (vgl. 8.1)

1.	C anwendbar, wenn A anwendbar	$\mathrm{dom}\, A \subseteq \mathrm{dom}\, C$	$\mathrm{pre}\, A \Rightarrow \mathrm{pre}\, C$
2.	C determinierter als A	$(\mathrm{dom}\, A \lhd C) \subseteq A$	$(\mathrm{pre}\, A) \wedge C \Rightarrow A$

Jetzt: Erweiterung dieser Bedingungen für den Fall des Wechsels der Datenstruktur

Der Zusammenhang zwischen Zustandsdarstellungen in „abstrakter“ Spezifikation und „konkreter“ Implementierung wird gegeben durch eine Repräsentationsrelation („Kopplungsinvariante“, vgl. 8.2).

Idee: Die Implementierung einer Operation ...

1. ist anwendbar in jedem Repräsentanten eines Zustands, in dem die abstrakte Operation anwendbar ist, und
2. führt zu einem Repräsentanten eines möglichen Ergebniszustands der abstrakten Operation.

Motivation dieser Definition.

Spezifikationen beschreiben häufig Komponenten größerer Systeme. Komponentenspezifikationen dienen

- einerseits als Ausgangspunkt zur Entwicklung der Komponente und
- andererseits als Beschreibung der Komponenten-Schnittstelle, die von anderen Komponenten (bzw. deren Entwicklern) genutzt wird.

Im Verlauf der Komponenten-Entwicklung darf ...

- der Nichtdeterminismus von Operationen reduziert werden:
  die übrigen Komponenten müssen *jedes* durch die ursprüngliche Spezifikation erlaubtes Ergebnis akzeptieren.
- der Vorbereich von Operationen nicht eingeschränkt werden:
  Entwickler der übrigen Komponenten vertrauen darauf, für jede gemäß der abstrakten Beschreibung zulässige Eingabe ein Ergebnis zu erhalten, welches die Spezifikation erfüllt.

Eine Ausweitung des Vorbereichs von Operationen bei der Verfeinerung ist zulässig, kann aber von den übrigen Komponenten nicht genutzt werden.

Beispiel 91: Mittelwert berechnen

Die folgenden Schemata beschreiben einen „Taschenrechner“ zur Berechnung des Mittelwerts einer Multimenge ganzer Zahlen. Während die „abstrakte“ Spezifikation die eingegebenen Zahlen speichert und erst bei Ausführung der Operation *AbsMean* aufsummiert, speichert die „Implementierung“ lediglich die Summe und Anzahl der eingegebenen Werte.

$\sum b$ und $card\, b$ bezeichnen die Summe bzw. die Anzahl der Elemente von Bag b.

AbsCalculator

$$store : \mathrm{bag}\,\mathbb{Z}$$

Calculator

$$sum : \mathbb{Z}$$
$$cnt : \mathbb{N}$$

AbsInitCalculator

$$AbsCalculator$$
$$store = \emptyset$$

InitCalculator

$$Calculator$$
$$sum = cnt = 0$$

AbsEnter

$$\Delta AbsCalculator$$
$$n? : \mathbb{Z}$$
$$store' = store \uplus [\![n?]\!]$$

Enter

$$\Delta Calculator$$
$$n? : \mathbb{Z}$$
$$sum' = sum + n?$$
$$cnt' = cnt + 1$$

AbsMean

$$\Delta AbsCalculator$$
$$mean! : \mathbb{Z}$$
$$store \neq \emptyset$$
$$store' = store$$
$$mean! = (\textstyle\sum store)\ \mathrm{div}\ (card\, store)$$

Mean

$$\Delta Calculator$$
$$mean! : \mathbb{Z}$$
$$cnt \neq 0$$
$$sum' = sum$$
$$cnt' = cnt$$
$$mean! = sum\ \mathrm{div}\ cnt$$

Der Zusammenhang zwischen *Calculator* und *AbsCalculator* wird beschrieben durch die Repräsentationsrelation (*Kopplungsinvariante*)

RepCalculator
AbsCalculator
Calculator

$sum = \sum store$
$cnt = card\, store$

Zum Beispiel werden die „abstrakten“ Zustände

$$\langle store = [\![2,2,4,6]\!] \rangle \qquad \text{und} \qquad \langle store = [\![1,3,5,5]\!] \rangle$$

beide durch den Implementierungszustand

$$\langle sum = 14, cnt = 4 \rangle$$

beschrieben. Umgekehrt ist offensichtlich jedem „abstrakten“ Zustand genau ein Implementierungszustand zugeordnet, d.h. die Repräsentationsrelation ist funktional.

Das beobachtbare Verhalten (d.h. die Folge möglicher Operationen mit ihren Ein- und Ausgaben) der Implementierung entspricht dem Verhalten der Spezifikation. ◄

Beispiel 92: ID-Vergabe

Das folgende Beispiel beschreibt die Vergabe eindeutiger Identifikatoren. Während die „abstrakte“ Spezifikation intern die Menge der vergebenen IDs speichert und in der Operation *AbsAssignID* nichtdeterministisch eine beliebige noch nicht vergebene ID auszuwählen erlaubt, benutzt die Implementierung einen Zähler, dessen Wert die jeweils nächste zu vergebende ID angibt.

Ein interessanter Unterschied ergibt sich daraus für die Operation zur „Rückgabe“ einer ID durch die Benutzer: während *AbsReclaimID* die zurückgegebene ID aus der Menge der vergebenen IDs löscht, so dass sie wieder vergeben werden kann, bewirkt die Operation *ReclaimID* der Implementierung keine Zustandsänderung.

AbsID

$used : \mathbb{P}\,\mathbb{N}$

ID

$nextID : \mathbb{N}$

AbsInitID
AbsID

$used = \emptyset$

InitID
ID

$nextID = 0$

AbsAssignID
$\Delta AbsID$
$newID! : \mathbb{N}$

$newID! \notin used$
$used' = used \cup \{newID!\}$

AssignID
ΔID
$newID! : \mathbb{N}$

$newID! = nextID$
$nextID' = nextID + 1$

AbsReclaimID
$\Delta AbsID$
$freeID? : \mathbb{N}$

$used' = used \setminus \{freeID?\}$

ReclaimID
ΞID
$freeID? : \mathbb{N}$

- deterministische Implementierung: Zähler statt Menge

- Die Spezifikation schließt das Verhalten der Implementierung nicht aus, erlaubt aber Abläufe, die von der Implementierung nicht erzeugt werden können.
- Kopplungsinvariante: $used \subseteq 0\,..\,nextID - 1$ ◀

Allgemeiner Fall: Gegeben zwei Schematupel

$$Abs = \langle AbsState, AbsInit, AbsOps \rangle \qquad \text{und} \qquad Conc = \langle ConcState, ConcInit, ConcOps \rangle$$

und eine Kopplungsinvariante, beschrieben durch ein Schema

$$Rep \mathrel{\widehat{=}} [AbsState;\ ConcState \mid RepInv]$$

Conc verfeinert *Abs* bzgl. *Rep*, wenn gilt:

1. Jeder mögliche Anfangszustand von *Conc* repräsentiert einen möglichen Anfangszustand von *Abs*.
2. Für jede Operation *COp* von *Conc* gibt es eine Operation *AOp* von *Abs*, so dass gilt
   (a) Ist *AOp* anwendbar in Zustand *a* und *c* ein möglicher Repräsentant von *a* (d.h. *a* und *c* erfüllen die Kopplungsinvariante *Rep*), so ist *COp* anwendbar in *c*.
   (b) Ist c' ein möglicher Ergebniszustand von *COp*, angewandt auf Zustand *c*, und ist *c* Repräsentant eines Zustand *a*, auf den *AOp* anwendbar ist, so ist c' ein Repräsentant eines möglichen Ergebniszustands a' von *AOp*, angewandt auf *a*.

Definition 47 *Ein Tupel* $Conc = \langle CState, CInit, COps \rangle$ verfeinert *ein Tupel* $Abs = \langle AState, AInit, AOps \rangle$ *bezüglich einer Kopplungsinvariante Rep, wenn gilt:*

Initialisierung. $CInit \Rightarrow \exists AState \bullet AInit \wedge Rep$

Operationen. *Für jede Operation* $COp \in COps$ *existiert eine Operation* $AOp \in AOps$ *mit:*

Anwendbarkeit. $Rep \wedge (\text{pre } AOp) \Rightarrow \text{pre } COp$

Korrektheit. $Rep \wedge (\text{pre } AOp) \wedge COp \Rightarrow \exists AState' \bullet AOp \wedge Rep'$

Oft wird zusätzlich gefordert: Für jede Operation $AOp \in AOps$ existiert eine implementierende Operation $COp \in COps$.

Bemerkungen:

1. Operationsverfeinerung (Definition 45 aus Abschnitt 8.1) ist Spezialfall von Definition 47 mit $CState = AState$, $CInit = AInit$ und der Identität als Repräsentationsrelation.
2. Manchmal werden zusätzliche Operationen der Implementierung ohne Ein- und Ausgaben erlaubt, die keine Auswirkungen auf den abstrakten Zustand haben. In diesem Fall fordert man

   $$Rep \wedge COp \wedge \Xi AState \Rightarrow Rep'$$

   Dies ermöglicht „interne“ Übergänge der Implementierung, die für den Benutzer unsichtbar sind.

Beweisverpflichtungen der Verfeinerungsbedingungen für Beispiel 92

Kopplungsinvariante

IDRep
AbsID
ID

$used \subseteq (0\,..\,nextID-1)$

Initialisierung

$$ID \wedge nextID = 0 \;\Rightarrow\; \exists used \bullet used \subseteq (0\,..\,-1)$$

Assign

(a) $IDRep \wedge (\text{pre } AbsAssignID) \;\Rightarrow\; \text{pre } AssignID$

(b) $IDRep \wedge (\text{pre } AbsAssignID) \wedge AssignID$
$\Rightarrow\; \exists AbsID' \bullet AbsAssignID \wedge IDRep'$
$\iff$
$used \subseteq (0\,..\,nextID-1) \wedge newID! = nextID \wedge nextID' = nextID+1$
$\Rightarrow\; \exists used' \bullet used' = used \cup \{newID!\} \wedge used' \subseteq (0\,..\,nextID'-1)$

Reclaim

(a) $IDRep \wedge (\text{pre } AbsReclaimID) \;\Rightarrow\; \text{pre } ReclaimID$

(b) $IDRep \wedge (\text{pre } AbsReclaimID) \wedge ReclaimID$
$\Rightarrow\; \exists AbsID' \bullet AbsReclaimID \wedge IDRep'$
$\iff$
$used \subseteq (0\,..\,nextID-1) \wedge nextID' = nextID$
$\Rightarrow\; \exists used' \bullet used' = used \setminus \{freeID?\} \wedge used' \subseteq (0\,..\,nextID'-1)$

Q.E.D.

Eine Zustandsfolge $s_0, s_1, s_2, \ldots$ heißt *Ablauf* einer Z-Spezifikation $\langle State, Init, Ops\rangle$, wenn s_0 die Initialisierungsbedingung *Init* erfüllt und für alle $i \geq 0$ eine Operation $Op \in Ops$ existiert, so dass das Zustandspaar $\langle s_i, s_{i+1}\rangle$ ein Modell von *Op* ist.

Folgerung. Sei *Conc* eine Verfeinerung von *Abs* und $c_0, c_1, \ldots$ ein Ablauf von *Conc*, in dem konkrete Operationen *COp* nur angewandt wurden, falls in jedem korrespondierenden abstrakten Zustand die entsprechende abstrakte Operation *AOp* anwendbar ist.

Dann existiert ein Ablauf $a_0, a_1, \ldots$ von *Abs*, so dass a_i Repräsentant von c_i ist (für alle i), vgl. Abbildung 11. Ist in a_k die Operation *AOp* anwendbar und *COp* eine Verfeinerung von *AOp*, so ist *COp* in c_k anwendbar.

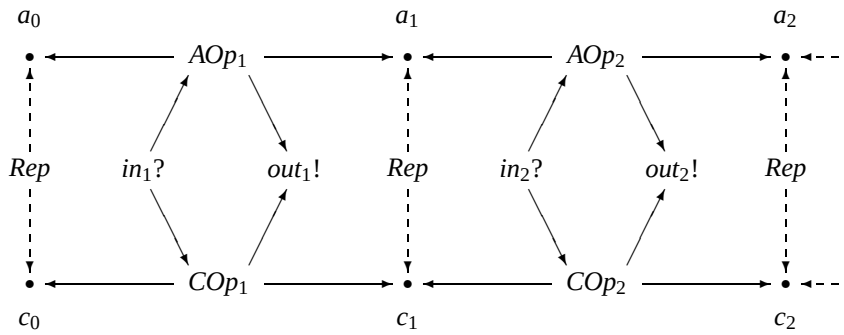


Abbildung 11: Abläufe der Verfeinerung simulieren Abläufe der abstrakten Spezifikation.

Vorsicht: Es muss nicht zu jedem Ablauf von *Abs* ein entsprechender Ablauf von *Conc* existieren!

8.4 Übergang zu imperativen Programmen

Ziele

- Zusammenhang zwischen Z-Schemata und imperativen Programmen erkennen
- Formale Entwicklung von Programmen aus Zustands- und Operationsbeschreibungen
- Grundlagen des Verfeinerungskalküls

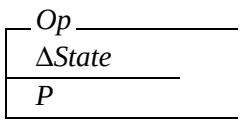
Literatur

R. Back, J. von Wright: *Refinement calculus—A Systematic Introduction*. Springer-Verlag, 1998.

C. Morgan: *Programming from Specifications*. Prentice-Hall, 3. Auflage 1998.

H. Partsch: *Specification and Transformation of Programs*. Springer-Verlag, 1990.

Programme als Relationen über Zuständen. Ein Z-Schema der Form



bzw. $Op \;\widehat{=}\; [\Delta State \mid P]$

beschreibt eine Relation zwischen Zuständen. Ebenso kann ein Programm als Relation zwischen zwei (Speicher-)Zuständen interpretiert werden.

Beispiel 93: Jedes der folgenden Programme überführt einen Zustand mit $x = 0 \wedge y = 1$ in einen Zustand mit $x = 1 \wedge y = 1$.

- `x := 1`
- `x := y`
- `x := x+1`
- `while x<1 do x := x+1 end` ◀

Beispiel-Programmiersprache in Anlehnung an „guarded commands“

(Mehrfach-)Zuweisung	`x,y := x+y, y-x`
Hintereinanderausführung	P `;` Q

bedingte Anweisung (nichtdeterministisch)	`if`
	 `[]` b_1 `->` P_1
	 $\vdots$
	 `[]` b_n `->` P_n
	 `else` P_{n+1} (optional)
	 `fi`

Schleife	`do` b `->` P `od`
lokaler Block	`|[ x:T ;` Π `]|`

Während der Entwicklung entstehen „gemischte Programme“, d.h. Programmgerüste, die Z-Schemata an Stellen noch zu entwickelnder Programmteile enthalten.

Allgemeiner Rahmen. Ein Schematupel $\langle State, Init, Ops\rangle$ wird überführt in ein Programm der Gestalt

```
|[ StateDecls ;
   Init ;
   type Choice = Quit | Op1 | ... | OpN ;
   |[ choice : Choice ; MakeChoice ;
      do  choice ≠ Quit ->
          if  choice = Op1 ->
              |[ InOutDecls1 ; GetInputs1 ; Op_1 ; SendOutputs1 ]|
          ...
          []  choice = OpN ->
              |[ InOutDeclsN ; GetInputsN ; Op_n ; SendOutputsN ]|
          fi ;
          MakeChoice
      od
    ]|
]|
```

Dabei werden durch `StateDecls` die im Schema *State* deklarierten Schemakomponenten als Programmvariablen deklariert und in `InOutDeclsi` die Eingabe- und Ausgabeparameter der Operation Op_i. Die Schemainvariante von *State* muss im Programm nicht berücksichtigt werden, da sie durch die Initialisierung sichergestellt wird und von allen Operationen erhalten bleibt.

Vereinfachende Annahmen:

- Alle Operationen der Ausgangsspezifikation sind total (ggf. Einführung von Fehlercodes).
- Prädikate aller Schemata enthalten nur Variablen, die im Deklarationsteil vorkommen (keine globalen Variablen).
- die Typen der deklarierten Variablen sind in der Programmiersprache vorhanden (ggf. zuvor Datenverfeinerung, siehe Definition 47 in Abschnitt 8.3).

Wir schreiben im folgenden (abweichend von strenger Z-Syntax)

$$\begin{array}{|l} \hline Op \\ \Delta[\vec{x} : \vec{T}];\ \Xi[\vec{y} : \vec{U}] \\ \hline P(\vec{x},\vec{y}) \wedge Q(\vec{y}) \\ \hline \end{array}$$

Implementierungen von *Op* dürfen nur Zuweisungen an die Variablen in $\vec{x}$ enthalten. Die Variablen in $\vec{y}$ dürfen nur lesend benutzt werden.

Grundprinzipien der Verfeinerungsregeln:

- Z-Schemata stehen nur an solchen Stellen im Programm, an denen ihre Vorbedingung erfüllt ist (Anwendbarkeitsprinzip).
- Alle im Deklarationsteil eines Schemas genannten Variablen sind an der entsprechenden Stelle im Programm deklariert.
- Nur die explizit in $\Delta[\vec{x} : \vec{T}]$ genannten Variablen dürfen verändert werden („frame rule“).

Das Anwendbarkeitsprinzip gilt im Ausgangsprogramm, da alle Operationen als total angenommen werden. Es wird von allen folgenden Verfeinerungsregeln vorausgesetzt und für das verfeinerte Programm sichergestellt. Die „frame rule“ bezieht sich auf die aktuell deklarierten Variablen. Später eingeführte lokale Variablen dürfen beliebig verändert werden, da sie außerhalb des Blocks nicht sichtbar sind.

Implementierungsbegriff. Seien Φ, Ψ (gemischte) Programme. Es gilt

$$\Phi \sqsubseteq \Psi \qquad (\Psi \textit{ implementiert } \Phi)$$

genau dann, wenn

1. Ψ anwendbar ist, wann immer Φ anwendbar ist und
2. jeder mögliche Zustandsübergang gemäß Ψ auch von Φ erlaubt wird, vorausgesetzt, dass Φ im Ausgangszustand anwendbar ist.

Werden Φ und Ψ als Relationen über Zuständen aufgefasst, gilt also wieder

$$\Phi \sqsubseteq \Psi \quad \Longleftrightarrow \quad \mathrm{dom}\,\Phi \subseteq \mathrm{dom}\,\Psi \quad \wedge \quad (\mathrm{dom}\,\Phi \lhd \Psi) \subseteq \Phi$$

Sind Φ und Ψ Schemata und ist Ψ operationelle Verfeinerung von Φ (siehe 8.1), so gilt $\Phi \sqsubseteq \Psi$.

Dies erlaubt u.a. prädikatenlogische Umformungen während der Programmentwicklung.

Die folgenden Regeln dienen zum schrittweisen Übergang von Operationsschemata zu Programmen. Wir begründen jeweil informell ihre Korrektheit.

Regel 1: Einführung lokaler Variablen

$$\boxed{\begin{array}{l} Op \\ \Delta[\vec{x} : \vec{T}];\ \Xi[\vec{y} : \vec{U}] \\ \hline P \end{array}} \quad \sqsubseteq \quad |[\ \vec{z} : \vec{V}\ ; \quad \boxed{\begin{array}{l} Op1 \\ \Delta[\vec{x} : \vec{T};\ \vec{z} : \vec{V}];\ \Xi[\vec{y} : \vec{U}] \\ \hline P \end{array}} \quad]|$$

für „neue“ Variablen $\vec{z}$

Die Variablen $\vec{z}$ kommen konventionsgemäß im Prädikat P nicht vor. Sie können aber in späteren Verfeinerungsschritten verändert werden.

Dagegen dürfen keine Annahmen über die Anfangswerte von $\vec{z}$ getroffen werden, da dies die Anwendbarkeit von $Op1$ einschränken würde.

Regel 2: Weglassen unbenutzter Variablen

$$\boxed{\begin{array}{l} Op \\ \Delta[\vec{x} : \vec{T}];\ \Xi[\vec{y} : \vec{U};\ \vec{z} : \vec{V}] \\ \hline P(\vec{x},\vec{y}) \wedge Q(\vec{y},\vec{z}) \end{array}} \quad \sqsubseteq \quad \boxed{\begin{array}{l} Op1 \\ \Delta[\vec{x} : \vec{T}];\ \Xi[\vec{y} : \vec{U}] \\ \hline P(\vec{x},\vec{y}) \end{array}}$$

Begründung.

- Die „frame rule“ garantiert, dass die Variablen $\vec{z}$ nicht verändert werden.
- Die Anwendbarkeitsbedingung garantiert, dass $Q(\vec{y},\vec{z})$ vor der Ausführung von Op, und daher auch von $Op1$ gilt. Da $\vec{y}$ und $\vec{z}$ von $Op1$ nicht verändert werden, gilt $Q(\vec{y},\vec{z})$ auch noch nach Terminierung von $Op1$.

Beispiel 94:

SwapIfGreater		*Swap*
$\Delta[x,y:\mathbb{Z}];\ \Xi[z:\mathbb{N}]$	$\sqsubseteq$	$\Delta[x,y:\mathbb{Z}]$
$z>5 \wedge x'=y \wedge y'=x$		$x'=y \wedge y'=x$

◀

Regel 3: Zuweisungsregel (für einfache Variablen)

Op		
$\Delta[x_1:T_1;\ \ldots;\ x_n:T_n];\ \Xi[\vec{y}]$	$\sqsubseteq$	`x1,...,xn := e1,...,en`
$x_1'=e_1 \wedge \ldots \wedge x_n'=e_n$		
P		

Voraussetzungen:

- Die Variablen $x_1,\ldots,x_n$ sind paarweise verschieden.
- Die Ausdrücke $e_1,\ldots,e_n$ enthalten keine gestrichenen Variablen und entsprechen „unmittelbar“ den Ausdrücken `e1,...,en` der Programmiersprache.

Begründung. Für die Vorbedingung von Op gilt

$$\text{pre } Op \equiv P[e_1/x_1',\ldots,e_n/x_n',\vec{y}/\vec{y}']$$

und die Anwendbarkeitsbedingung garantiert, dass dieses Prädikat vor Ausführung der Zuweisung erfüllt ist. Daher erfüllt die Zuweisung insgesamt die Formel P.

Beispiel 95:

SwapIfGreater		
$\Delta[x,y:\mathbb{Z}];\ \Xi[z:\mathbb{N}]$	$\sqsubseteq$	`x,y := y,x`
$z>5 \wedge x'=y \wedge y'=x$		

◀

Verallgemeinerung: Zuweisungen an Arraykomponenten

Arrays können in Z durch Sequenzen modelliert werden, Zuweisungen entsprechen dem Überschreiben des Arrays an bestimmten Indexpositionen.

$$\begin{array}{|l} \textit{Op} \\ \hline \Delta[x_1 : T_1; \ldots; x_n : T_n];\ \Xi[\vec{y}] \\ \hline \ldots \wedge x_i' = x_i \oplus \{j \mapsto e\} \wedge \ldots \\ P \\ \hline \end{array} \quad \sqsubseteq \quad \texttt{...,xi[j],...:= ...,e,...}$$

falls gilt: $P \Rightarrow 1 \leq j \leq \#x_i$

Beispiel 96:

$$\begin{array}{|l} \textit{SwapIJ} \\ \hline \Delta[x : \mathrm{seq}\,\mathbb{Z}];\ \Xi[i,j : \mathbb{N}] \\ \hline x' = x \oplus \{i \mapsto x(j), j \mapsto x(i)\} \\ 1 \leq i < j \leq \#x \\ \hline \end{array} \quad \sqsubseteq \quad \texttt{x[i],x[j] := x[j],x[i]}$$

◀

Ähnlich: Zuweisungen an record-Komponenten.

Regel 4: Sequenzialisierung

$$\frac{\mathrm{pre}\ \textit{Op} \Rightarrow \mathrm{pre}\ \textit{Op}_1 \qquad \mathrm{pre}\ \textit{Op} \wedge \textit{Op}_1 \Rightarrow (\mathrm{pre}\ \textit{Op}_2)' \qquad \mathrm{pre}\ \textit{Op} \wedge (\textit{Op}_1 \fatsemi \textit{Op}_2) \Rightarrow \textit{Op}}{\textit{Op} \quad \sqsubseteq \quad \textit{Op}_1 \ ;\ \textit{Op}_2}$$

Begründung.

- Ist Op in einem Zustand s anwendbar, dann ist auch Op_1 anwendbar. Daher gilt das Anwendbarkeitsprinzip für Op_1, wenn es zuvor für Op galt.
- Jede Ausführung von Op_1, ausgehend von einem solchen Zustand s, endet in einem Zustand, in dem Op_2 anwendbar ist. Dies garantiert die Erfülltheit des Anwendbarkeitsprinzips für Op_2.
- Werden Op_1 und Op_2 in einem solchen Zustand s hintereinander ausgeführt, so genügt das entstehende Ergebnis der Spezifikation Op.

Beispiel 97: Suche nach Indexposition in Array mit maximalem Element

$$\begin{array}{|l} \textit{FindMaxPos} \\ \hline \Delta[\mathit{maxel} : \mathbb{N}];\ \Xi[a : \mathrm{seq}\,\mathbb{Z}] \\ \hline \#a > 0 \\ 1 \leq \mathit{maxel}' \leq \#a \\ \forall n : 1\,..\,\#a \bullet a(\mathit{maxel}') \geq a(n) \\ \hline \end{array}$$

1. Schritt: Einführung einer lokalen Variablen für spätere Iteration

$$FindMaxPos \sqsubseteq \texttt{|[seen:int;}\ \begin{array}{|l} FindMaxSeen \\ \Delta[maxel:\mathbb{N};\ seen:\mathbb{Z}];\ \Xi[a:\mathrm{seq}\,\mathbb{Z}] \\ \hline \#a > 0 \\ 1 \le maxel' \le \#a \\ \forall n:1\,..\,\#a \bullet a(maxel') \ge a(n) \end{array}\ \texttt{]|}$$

2. Schritt: Aufspaltung von *FindMaxSeen* in Initialisierung und weitere Berechnung

$$FindMaxSeen \sqsubseteq \begin{array}{|l} Initialise \\ \Delta[maxel:\mathbb{N};\ seen:\mathbb{Z}] \\ \Xi[a:\mathrm{seq}\,\mathbb{Z}] \\ \hline MaxelInv' \end{array}\ ;\ \begin{array}{|l} Complete \\ \Delta[maxel:\mathbb{N};\ seen:\mathbb{Z}] \\ \Xi[a:\mathrm{seq}\,\mathbb{Z}] \\ \hline seen' = \#a \\ \Delta MaxelInv \end{array}$$

mit dem Hilfsschema

$$\begin{array}{|l} MaxelInv \\ maxel:\mathbb{N};\ seen:\mathbb{Z};\ a:\mathrm{seq}\,\mathbb{Z} \\ \hline \#a > 0 \\ 1 \le maxel \le seen \le \#a \\ \forall n:1\,..\,seen \bullet a(maxel) \ge a(n) \end{array}$$

Beweis der Verfeinerungsbedingungen.

1. pre *FindMaxSeen* $\Rightarrow$ pre *Initialise*
   reduziert auf $\#a > 0 \Rightarrow \#a > 0$
2. pre *FindMaxSeen* $\wedge$ *Initialise* $\Rightarrow$ (pre *Complete*)$'$
   gilt, da *Initialise* die Bedingung *MaxelInv*$'$ impliziert
3. pre *FindMaxSeen* $\wedge$ (*Initialise* $\fatsemi$ *Complete*) $\Rightarrow$ *FindMaxSeen*
   Complete garantiert $seen' = \#a \wedge MaxelInv'$

Q.E.D.

3. Schritt: Implementierung von *Initialise*

$$\begin{array}{lll} Initialise & \sqsubseteq & \begin{array}{|l} InitRefined \\ \Delta[maxel:\mathbb{N};\ seen:\mathbb{Z}];\ \Xi[a:\mathrm{seq}\,\mathbb{Z}] \\ \hline \#a > 0 \\ seen' = maxel' = 1 \end{array} \quad \text{(operationelle Verfeinerung)} \\ & \sqsubseteq & \texttt{seen,maxel := 1,1} \quad \text{(Zuweisungsregel)} \end{array}$$

◀

Regel 5: Fallunterscheidung

Sei *Op* ein Schema, $b_1, \ldots, b_n$ Bedingungen, in denen nur ungestrichene Variablen aus *Op* vorkommen, und die „unmittelbar“ Bedingungen `b1`, ..., `bn` in der Programmiersprache entsprechen.

Gilt pre $Op \Rightarrow b_1 \vee \ldots \vee b_n$, so folgt

$$
\begin{array}{lll}
Op & \sqsubseteq & \texttt{if b1 -> } b_1 \wedge Op \\
 & & \texttt{[] b2 -> } b_2 \wedge Op \\
 & & \ldots \\
 & & \texttt{[] bn -> } b_n \wedge Op \\
 & & \texttt{fi}
\end{array}
$$

Idee: Wähle b_i so, dass sich die einzelnen Alternativen in den folgenden Schritten vereinfachen lassen.

Begründung.

- Die Anwendbarkeitsbedingungen für die Schemata $b_i \wedge Op$ gelten wegen Voraussetzung

  $$\text{pre } Op \Rightarrow b_1 \vee \ldots \vee b_n$$

- Korrektheit folgt durch aussagenlogische Vereinfachung.

Beispiel 98: Iterationsschritt für Suche nach größtem Element (vgl. Beispiel 97)

StepSeen

$\Delta[maxel : \mathbb{N};\ seen : \mathbb{Z}];\ \Xi[a : \text{seq}\,\mathbb{Z}]$
$\Delta MaxelInv$
$seen' = seen + 1$

$\sqsubseteq$

MoveSeen

$\Delta[seen : \mathbb{Z}];\ \Xi[a : \text{seq}\,\mathbb{Z};\ maxel : \mathbb{N}]$
$MaxelInv$
$1 \leq seen < \#a$
$seen' = seen + 1$

;

AdjustMaxel

$\Delta[maxel : \mathbb{N}];\ \Xi[a : \text{seq}\,\mathbb{Z};\ seen : \mathbb{Z}]$
$1 \leq maxel < seen \leq \#a$
$\forall n : 1\,..\,(seen - 1) \bullet a(maxel) \geq a(n)$
$MaxelInv'$

$\sqsubseteq$

```
seen := seen+1 ;
if a[seen] > a[maxel]    ->  a(seen) > a(maxel) ∧ AdjustMaxel
[] a[seen] <= a[maxel]   ->  a(seen) ≤ a(maxel) ∧ AdjustMaxel
fi
```

$\sqsubseteq \ldots \sqsubseteq$

```
seen := seen+1 ;
if a[seen] > a[maxel]    ->  maxel := seen
[] a[seen] <= a[maxel]   ->  maxel := maxel
fi
```

◀

Regel 6: Iterationsregel Implementierung durch Schleife $Op \sqsubseteq$ `do b ->` *Body* `od`

Idee: Invariante *Inv* und Variante *v*

- Invariante gilt am Anfang und Ende jeder Ausführung von *Body*
- Invariante und Bedingung garantiert Ausführbarkeit von *Body*
- Invariante und Negation der Bedingung garantiert Korrektheit
- Variante wird bei jeder Ausführung von *Body* kleiner und sichert Terminierung

Formal: Es seien

Inv, *b*, *Goal*	Prädikate ohne gestrichene Variablen
`b`	„unmittelbare“ Übersetzung von *b* in die Programmiersprache
v	ganzzahliger arithmetischer Ausdruck ohne gestrichene Variablen.
Iterate	Operationsschema für den Iterationsschritt
$\vec{x}, \vec{y}$	Tupel aller vorkommenden Variablen

Die Verfeinerungsbeziehung

Op
$\Delta[\vec{x}]; \Xi[\vec{y}]$
ΔInv
$Goal'$

$\sqsubseteq$ `do b ->`

Body
$\Delta[\vec{x}]; \Xi[\vec{y}]$
ΔInv
Iterate

`od`

gilt, falls alle folgenden Bedingungen erfüllt sind:

$Inv \wedge b \Rightarrow \text{pre } Body$
$Inv \wedge \neg b \Rightarrow Goal$
$b \wedge Body \Rightarrow 0 \leq v' < v$

Beispiel 99: Implementierung von *Complete* durch Schleife (vgl. Bsp. 97)

Complete $\sqsubseteq$ `do seen < #a ->` *StepSeen* `od`

Instanziierungen der Iterationsregel

Inv	*MaxelInv*	*b*	$seen < \#a$
Goal	$seen = \#a$	*Iterate*	$seen' = seen + 1$
v	$\#a - seen$		

Entstandener Code in den Beispielen 97–99:

```
|[ seen : int ;
  seen, maxel := 1,1 ;
  do  seen < #a ->
          seen := seen+1 ;
          if a(seen) > a(maxel)  ->  maxel := seen
          []  a(seen) <= a(maxel)  ->  maxel := maxel
          fi
  od ]|
```

◀

8.5 Zusammenfassung

- Die Wirkungen von Operationen zustandsbasierter Systeme sind nicht allein durch deren Eingabeparameter bestimmt, sondern auch abhängig vom aktuellen Systemzustand.
- Z-Spezifikationen ermöglichen modellorientierte Beschreibungen zustandsbasierter, interaktiver Systeme.

- Eine typische Systemspezifikation hat die Form $\langle State, Init, Ops \rangle$.

  Dabei definiert das Schema *State* die Komponenten (mit ihren Typen) zur Darstellung eines Zustands und legt durch Schema-Invarianten die Beziehungen zwischen den Zustandskomponenten fest.

  Das Schema *Init* beschreibt die Teilmenge möglicher Anfangszustände.

  Jede Operation wird durch ein Schema $Op \in Ops$ mit Hilfe eines Prädikats über Vor- und Nachzustand sowie Ein- und Ausgaben beschrieben.

- Im Unterschied zu algebraischen Spezifikationssprachen wie CASL werden die grundlegenden Datenstrukturen nicht durch charakteristische Eigenschaften beschrieben, sondern (i.w.) aus fest vorgegebenen Basisstrukturen konstruiert.

  Z basiert auf Mengenlehre (nach Zermelo-Fraenkel) und abgeleiteten Konstrukten wie Relationen, Funktionen und Folgen.

  Wie CASL erlaubt Z freie Typkonstruktionen, kennt aber keine rekursiven Funktionsdefinitionen.

- Z-Schemata können durch „Dekorationen“ umbenannt und mittels logischer Operatoren einschließlich Quantoren miteinander verknüpft werden.

  Dadurch werden Konzepte wie Hiding und sequenzielle Komposition im „Schemakalkül“ darstellbar.

- Wie bei algebraischen Spezifikationen beruht die Spezifikationsentwicklung in Z auf einem Verfeinerungsbegriff.

- Für eine Operationsverfeinerung von *AOp* durch *COp* müssen

Anwendbarkeitsbedingung	$\text{pre}\ AOp \Rightarrow \text{pre}\ COp$	und
Korrektheitsbedingung	$(\text{pre}\ AOp) \wedge COp \Rightarrow AOp$	

  erfüllt sein.

- Das Konzept der Datenverfeinerung verallgemeinert den entsprechenden Begriff bei algebraischen Spezifikationen, indem der Zusammenhang zwischen dem Zustand der abstrakten Spezifikation und dem Zustand der Implementierung durch eine Relation ausgedrückt wird. Dadurch kann das Problem der Überspezifikation auf der abstrakten Ebene (d.h. einer zu konkreten Zustandsdefinition bei der Anforderungsspezifikation) abgemildert werden.

- Die Entwicklung imperativer Programme aus Z-Spezifikationen von Operationen kann formal durch den Verfeinerungskalkül abgestützt werden.

Teil IV

Reaktive Systeme

Bisher standen folgende Aspekte der Beschreibung von Systemen im Vordergrund:

Datenstrukturen und Funktionen: Axiomatische Beschreibung von Datenstrukturen und zugehöriger Operationen durch Angabe ihres Ein-/Ausgabeverhaltens

zustandsbasierte Systeme: Wirkungen von (Benutzer-)Aktionen auf den Zustand eines „interaktiven" Systems

Der abschließende Teil der Vorlesung befasst sich mit *reaktiven Systemen*. Charakteristika reaktiver Systeme sind:

- kontinuierliche Interaktion mit der Umgebung
- Terminierung unerwünscht bzw. unwesentlich

Typische Beispiele für reaktive Systeme sind Prozesssteuerungen oder Kommunikations- oder Krypto-Protokolle. Bei der formalen Beschreibung reaktiver Systeme dominiert meist der „Kontrollanteil", während der Berechnungsaspekt in den Hintergrund tritt.

9 Abläufe von Transitionssystemen

9.1 Begriff, Beispiele

Transitionssysteme bildeten bereits die Grundlage zur Beschreibung zustandsbasierter Systeme in Teil III der Vorlesung. Während dort die Beschreibung einzelner Systemzustände oder Übergänge im Blickpunkt stand, konzentrieren wir uns bei der Beschreibung reaktiver Systeme auf die Abläufe von Transitionssystemen. Dazu wiederholen bzw. erweitern wir die Definition 44 von Transitionssystemen.

Definition 48 *Ein markiertes Transitionssystem* $\Gamma = (Z, I, \mathcal{A}, \delta)$ *mit Anfangszuständen ist gegeben durch*

- *eine Menge* Z *von Zuständen,*
- *eine nichtleere Menge* $I \subseteq Z$ *von Anfangszuständen,*
- *eine Menge* $\mathcal{A}$ *von Aktionen und*
- *eine Zustandsübergangsrelation* $\delta \subseteq Z \times \mathcal{A} \times Z$.

Die Aktion A *heißt* ausführbar *(„enabled") im Zustand* $s \in Z$*, wenn ein* $t \in Z$ *existiert mit* $(s, A, t) \in \delta$.

Im folgenden setzen wir voraus, dass in jedem Zustand $s \in Z$ *mindestens eine Aktion* A *ausführbar ist.*

Ein Ablauf *von* Γ *ist eine unendliche Folge* $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ *mit* $s_0 \in I$ *und* $(s_i, A_i, s_{i+1}) \in \delta$ *für alle* $i \geq 0$.

Bemerkungen:

- Transitionssysteme haben eine ähnliche Struktur wie endliche Automaten, aber keine Endzustände.
- In der Literatur findet man uneinheitliche Definitionen von Transitionssystemen. Typische Varianten sind:
  - Beschränkung auf einen Anfangszustand $s_0 \in Z$ statt einer Menge $I \subseteq Z$.
  - Aktionen werden oft nicht benannt, d.h. man definiert $\delta \subseteq Z \times Z$.
  - Die Totalität von δ wird nicht immer gefordert; neben unendlichen Abläufen sind dann auch endliche Abläufe möglich.
  - Einführung einer „Stotteraktion“ $\tau \in \mathcal{A}$ mit $(s,\tau,t) \in \delta$ gdw. $s = t$.
- Die Zustandsmenge Z darf unendlich, sogar überabzählbar sein.
- Transitionssysteme modellieren *diskrete Systeme*, da Zustandsübergänge als atomare Aktionen ohne Zeitdauer modelliert werden. Für die Modellierung von Realzeitaspekten oder von *hybriden Systemen* mit analogen Zustandsanteilen (wie z.B. Steuerungen physikalischer Prozesse) muss die Definition von Transitionssystemen geeignet erweitert werden.
- Die Zustandsmenge Z wird meist durch Belegungen einer Menge $\mathcal{V}$ von (Zustands-)Variablen angegeben, evtl. eingeschränkt durch eine Zustandsinvariante.
- Jede Z-Spezifikation der Form $\langle State, Init, Ops\rangle$ definiert ein Transitionssystem mit Aktionenmenge *Ops* (vorausgesetzt, in jedem Zustand ist mindestens eine Operation ausführbar).

  Die Aktion *Op* ist ausführbar im Zustand s genau dann, wenn $s \models \text{pre } Op$ gilt.

Beispiel 100: Eisenbahnsteuerung als Transitionssystem

Die Eisenbahnstrecke in Abb. 12 besteht aus zwei Ringen, auf denen je ein Zug in bzw. entgegen dem Uhrzeigersinn fährt. Eine Brücke wird von beiden Zügen benutzt und durch Signale abgesichert. Die Aufgabe besteht im Entwurf einer Signalsteuerung zur Vermeidung von Kollisionen auf der Brücke, wobei Züge nicht unnötig aufgehalten werden sollen.

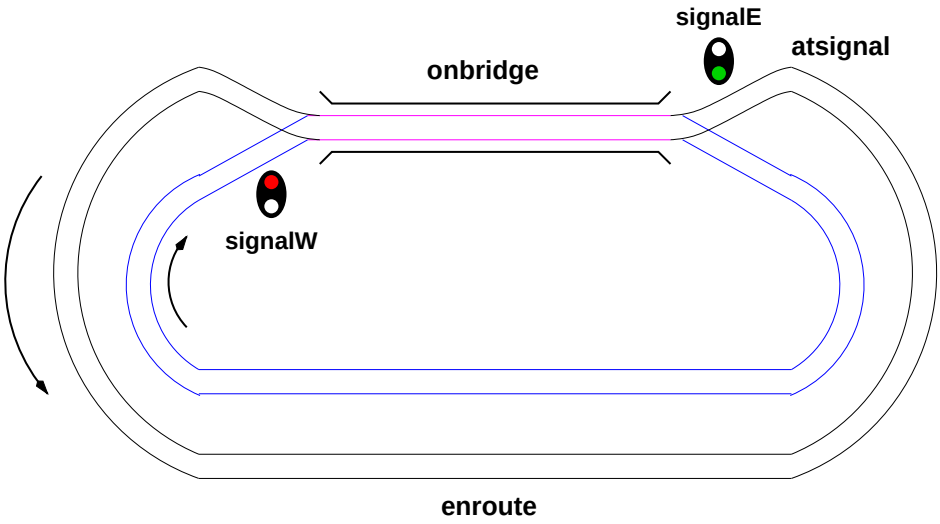


Abbildung 12: Eisenbahnsteuerung.

Die Zustandsinformation für die Steuerung beinhaltet Angaben über:

- die aktuelle Signalstellung
- die Position, Geschwindigkeit und Beschleunigung der Züge

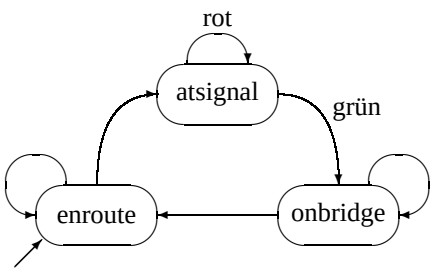
Zustandsübergänge sind von zweierlei Art:

Umgebungsaktionen. Sensoren aktualisieren die Information der Steuerung über den Zustand der Züge.

Systemaktionen. Durch Schalten der Signale verändert sich der aktuelle Zustand der Signale.

Für einen ersten Entwurf genügt eine abstraktere Modellierung, bei dem die Strecken in Abschnitte eingeteilt werden, wie in Abb. 12 angedeutet. Dadurch wird ein endlicher Zustandsraum erreicht. Die Zustandsübergänge der Züge und Signale wird in Abb. 13 dargestellt. (Die mit „?“ markierten Einträge können z.B. durch Priorisierung von trainW vervollständigt werden, so dass Kollisionen vermieden werden).

Züge



Signale (kombinatorische Schaltung)

trainW	trainE	signalW	signalE
enroute	atsignal	rot	grün
atsignal	enroute	grün	rot
atsignal	atsignal	?	?
sonst		rot	rot

Abbildung 13: Zustandsübergänge der Eisenbahnsteuerung („abstraktes Modell“).

Die abstrakte Modellierung führt zu Nichtdeterminismus: zum Beispiel kann ein Zug aus dem Zustand „onbridge“ in die Zustände „onbridge“ und „enroute“ übergehen. ◀

Beispiel 101: einfaches Kommunikationsprotokoll nach Lynch

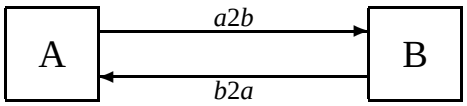


Abbildung 14: Kommunikationsprotokoll nach Lynch.

Im folgenden Beispiel modellieren wir ein einfaches Protokoll zur Datenübertragung zwischen zwei Partnern A und B (vgl. Abb. 14). Dabei werden folgende Annahmen getroffen:

- Ziel ist eine kontinuierliche Duplex-Übertragung zwischen A und B über die Kanäle *a*2*b* und *b*2*a*; es sei vorausgesetzt, dass ständig zu übertragende Nachrichten bereitstehen.
- Die Übertragungskanäle sind unzuverlässig: Nachrichten können bei der Übertragung verfälscht werden. Es werde vorausgesetzt, dass Übertragungsfehler erkannt werden können (z.B. durch Versehen der Nachrichten mit redundanter Information auf einer niedrigeren Protokollebene).
- Die Daten tragen daher *tags*, über die das Protokoll gesteuert wird. Mögliche tags sind `ack` (acknowledgement, positive Bestätigung der vorangegangenen Übertragung), `nak` (negative acknowledgement, kennzeichnet eine fehlerhafte Übertragung der vorangegangenen Nachricht) und `err` (Kennzeichnung einer fehlerhaft übertragenen Nachricht).

Das **Protokoll** (für beide Partner) wird informell folgendermaßen beschrieben:

1. Nach einer fehlerfreien Übertragung wird die nächste Nachricht auf dem Rückkanal mit `ack` versehen, ansonsten mit `nak`.
2. Nach einem Übertragungsfehler bzw. einer mit `nak` markierten Nachricht wird das letzte Datum nochmals übertragen, ansonsten wird die nächste zu sendende Nachricht geholt.
3. Fehlerfrei übertragene Nachrichten werden an die Umgebung ausgegeben.

Das Protokoll wird gestartet durch eine `err`-Nachricht von B an A.

Modellierung durch Transitionssystem. Wir modellieren zwei Kanäle *a*2*b* und *b*2*a* der Kapazität 1. Die Sender- und Empfängerseite jedes Kanals wird durch folgende drei Zustandskomponenten repräsentiert:

- ein Bit *valid*, das angibt, ob die Nachricht auf der jeweiligen Seite frisch oder veraltet ist; dient zur Synchronisation zwischen Prozess und Kanal,
- das tag *type* $\in \{\texttt{ack}, \texttt{nak}, \texttt{err}\}$ der aktuellen Nachricht und
- die eigentliche Nachricht *data* $\in$ *Data*; die Menge *Data* enthalte alle möglichen Nachrichten, die von den Prozessen versandt werden.

Die folgende Tabelle stellt dies kompakt dar; der Zustandsraum besteht aus den sortenrichtigen Belegungen der Zustandskomponenten *a*2*b*.*svalid*, ..., *b*2*a*.*rdata*.

Senderseite		Empfängerseite	
svalid	$\{0,1\}$	*rvalid*	$\{0,1\}$
stype	$\{\texttt{ack}, \texttt{nak}, \texttt{err}\}$	*rtype*	$\{\texttt{ack}, \texttt{nak}, \texttt{err}\}$
sdata	*Data*	*rdata*	*Data*

Die **Anfangszustände** entsprechen einer fehlerhaften Übertragung von B an A, sie sind daher formal gegeben durch die Menge aller Zustände mit

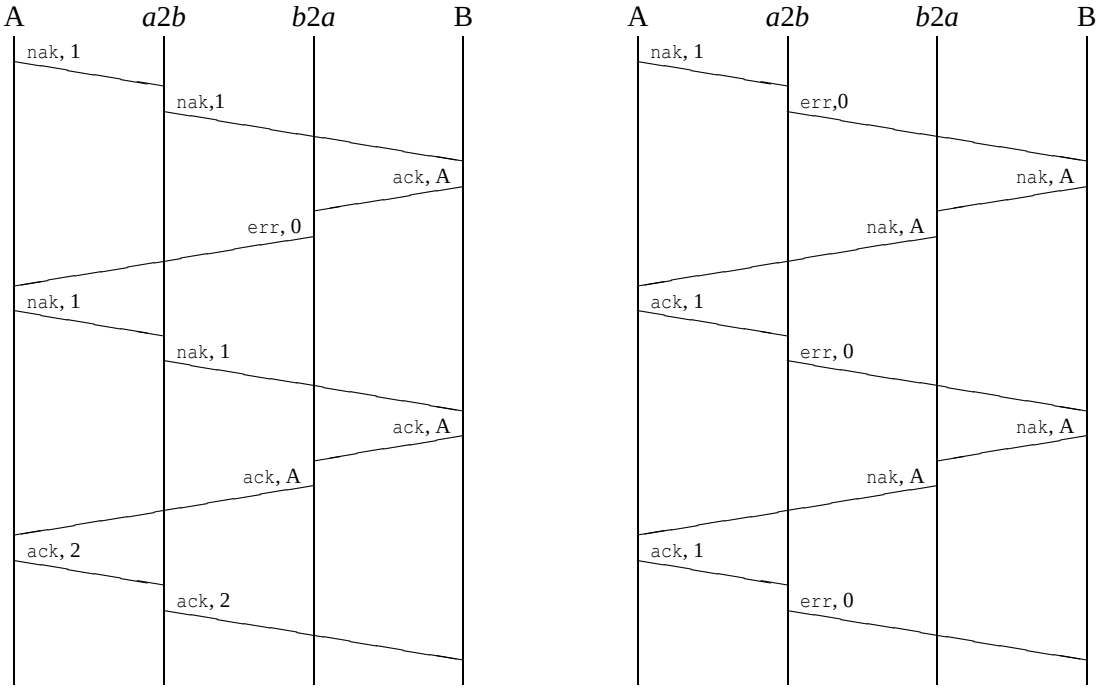
$b2a.rvalid = 1$, $b2a.rtype = \texttt{err}$, $b2a.svalid = 0$, $a2b.svalid = 0$

Die Belegung der übrigen Zustandskomponenten ist irrelevant.

Die möglichen **Aktionen** werden in der folgenden Tabelle in Pseudocode-Notation dargestellt. (Wir zeigen nur die Aktionen, die den Kanal *a*2*b* betreffen, die übrigen Aktionen sind symmetrisch.)

Aktion	Vorbedingung	Wirkung
Prozess A	$a2b.svalid = 0$, $b2a.rvalid = 1$	$b2a.rvalid := 0$, $outA :=$ **if** $rtype = \texttt{err}$ **then** $outA$ **else** append$(outA, b2a.rdata)$, $a2b.svalid := 1$, $a2b.stype :=$ **if** $rtype = \texttt{err}$ **then** `nak` **else** `ack`, $a2b.sdata :=$ **if** $rtype = \texttt{ack}$ **then** $nextAInput()$ **else** $a2b.sdata$
fehlerfreie Übertragung *a*2*b*	$a2b.svalid = 1$, $a2b.rvalid = 0$	$a2b.rvalid := 1$, $a2b.rtype := a2b.stype$, $a2b.rdata := a2b.sdata$, $a2b.svalid := 0$
fehlerhafte Übertragung *a*2*b*	$a2b.svalid = 1$, $a2b.rvalid = 0$	$a2b.rvalid := 1$, $a2b.rtype := \texttt{err}$, $a2b.svalid := 0$
Idle	**true**	**skip**

Die „Idle“-Aktion ist immer ausführbar und stellt sicher, dass es in jedem Zustand des Transitionssystems eine ausführbare Aktion gibt.



(a) einzelne Übertragungsfehler (b) Zusammenbruch eines Kanals

Abbildung 15: Abläufe des Kommunikationsprotokolls.

Abbildung 15 stellt zwei mögliche Abläufe des Protokolls durch Sequenzdiagramme dar, dabei sei angenommen, dass von A nach B die Folge 1, 2, 3, ... und umgekehrt die Folge A, B, C, ... gesandt wird. Bei Ablauf (a) wird eine Nachricht fehlerhaft übertragen, danach stabilisiert sich das Protokoll wieder. Ablauf (b) entspricht dem kompletten Zusammenbruch des Kanals *a*2*b*. Bei einem solchen Ablauf versuchen beide Partner immer wieder, dieselbe Nachricht zu senden; das Protokoll macht keinen Fortschritt mehr. ◀

Programme als Transitionssysteme

Bei der Modellierung von Abläufen (paralleler) Programme durch Transitionssysteme entsprechen die Zustände möglichen Belegungen der Programmvariablen. Außerdem gibt ein „Programmzähler“ (ggf. pro paralleler Komponente) den Kontrollflusszustand des Programms an.

Beispiel 102: „Stoppuhr“

var x, y : integer = $0, 0$;
cobegin
α : **while** $y = 0$ **do** $\beta : x := x + 1$ **end** $\quad\|\quad \gamma : y := 1$
coend

Zustände: Werte von x und y, außerdem „Programmzähler“ $\pi_1 \in \{\alpha, \beta, \tau_1\}$ und $\pi_2 \in \{\gamma, \tau_2\}$ (dabei stehen τ_1 und τ_2 für die Terminierung der beiden Komponenten).

Aktionen: entsprechen Programmanweisungen,„Stottern“ am Programmende

mögliche Abläufe

Ablauf 1

Aktion	—	α	β	α	β	α	γ	β	α	τ	⋯
x	0	0	1	1	2	2	2	3	3	3	⋯
y	0	0	0	0	0	0	1	1	1	1	⋯
π_1	α	β	α	β	α	β	β	α	τ_1	τ_1	⋯
π_2	γ	γ	γ	γ	γ	γ	τ_2	τ_2	τ_2	τ_2	⋯

Ablauf 2

Aktion	—	α	β	α	β	α	β	α	β	α	⋯
x	0	0	1	1	2	2	3	3	4	4	⋯
y	0	0	0	0	0	0	0	0	0	0	⋯
π_1	α	β	α	β	α	β	α	β	α	β	⋯
π_2	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ	⋯

◀

9.2 Fairnessbedingungen

Transitionssysteme enthalten häufig „unfaire“ Abläufe, die im modellierten System nicht vorkommen können. Typische Ursachen unfairer Abläufe sind:

- Abstraktion bei Modellierung führt zu Nichtdeterminismus im Modell

  vgl. Beispiel 100: Transitionssystem erlaubt Zyklen, in denen ein Zug immer auf der Brücke bleibt.
- Modellierung von Parallelität durch Nichtdeterminismus

  vgl. Beispiel 102: In Ablauf 2 wird Aktion γ nie ausgeführt.
- Zwei Aktionen stehen in Konflikt, d.h. sind im selben Zustand ausführbar

  vgl. Beispiel 101: korrekte und fehlerhafte Nachrichtenübertragung

Sollen unfaire Abläufe ausgeschlossen werden, muss die Menge der erlaubten Abläufe durch Zusatzbedingungen eingeschränkt werden.

- Beispiel 100: Enthält ein Ablauf unendlich viele Bewegungen eines Zugs ausgehend vom Abschnitt „onbridge“, so befindet sich der Zug unendlich oft nicht im Abschnitt „onbridge“.
- Beispiel 101: Werden unendlich viele Nachrichten entlang eines Kanals übertragen, so erfolgt die Übertragung unendlich oft fehlerfrei.
- Beispiel 102: Ein Prozess, der (ab einem gegebenem Zeitpunkt) persistent ausführbereit ist, führt irgendwann eine Aktion aus.

Fairnessbedingungen fordern intuitiv, dass Aktionen, die „genügend häufig“ ausführbar sind, irgendwann ausgeführt werden. Wir betrachten zwei Fairnessbegriffe:

schwache Fairness (weak fairness, justice): Ein Ablauf $s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ heißt schwach fair bezüglich der Aktion A, falls gilt:

Existiert ein $n \in \mathbb{N}$, so dass A in allen Zuständen s_m mit $m \geq n$ ausführbar ist, so ist $A_i = A$ für unendlich viele $i \in \mathbb{N}$.

Äquivalente Formulierung: Wird A nur endlich oft ausgeführt, so ist A im Ablauf unendlich oft nicht ausführbar.

starke Fairness (strong fairness, compassion): Ein Ablauf $s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ heißt stark fair bezüglich der Aktion A, falls gilt:

Existieren unendlich viele $m \in \mathbb{N}$, so dass A im Zustand s_m ausführbar ist, so ist $A_i = A$ für unendlich viele $i \in \mathbb{N}$.

Äquivalente Formulierung: Wird A nur endlich oft ausgeführt, so ist A im Ablauf nur endlich oft ausführbar.

Bemerkung: Starke Fairness impliziert schwache Fairness.

Ist ein Ablauf stark fair bezüglich A, so ist er auch schwach fair bezüglich A.

Definition 49 *Ein* Transitionssystem mit Fairness *(fair transition system, FTS)* $\Gamma_f = (Z, I, \mathcal{A}, \delta, W, S)$ *erweitert ein Transitionssystem* $\Gamma = (Z, I, \mathcal{A}, \delta)$ *um Mengen* $W, S \subseteq \mathcal{A}$.

Die Abläufe von Γ_f *sind diejenigen Abläufe von* Γ*, die schwach fair sind bezüglich der Aktionen* $A \in W$ *und stark fair bezüglich der Aktionen* $A \in S$.

Für die betrachteten Beispiele sind folgende Fairnessbedingungen sinnvoll:

Beispiel 100: schwache Fairness bzgl. der Aktionen „Brücke verlassen“ mit der Vorbedingung „Zug im Abschnitt onbridge“ und der Nachbedingung „Zug im Abschnitt enroute“ (für beide Züge).

Beispiel 101: starke Fairness bzgl. der Aktionen „Übertragung $a2b$“ und „Übertragung $b2a$“.

Beispiel 102: schwache Fairness für jeden der beiden Prozesse, d.h. für die Aktionen $P1$ und $P2$ mit

$$\begin{aligned} (s, P1, t) \in \delta &\quad\Leftrightarrow\quad (s, \alpha, t) \in \delta \text{ oder } (s, \beta, t) \in \delta \\ (s, P2, t) \in \delta &\quad\Leftrightarrow\quad (s, \gamma, t) \in \delta \end{aligned}$$

Die Wahl adäquater Fairnessbedingungen hängt vom jeweils modellierten System ab und ist häufig der schwierigste Teil der Modellierung.

Implementierung von Fairnessbedingungen

Fairnessbedingungen können durch Scheduler implementiert werden. Für schwache Fairness genügt ein „round-robin-Scheduler“, bei dem die Aktionen zyklisch auf Ausführbarkeit untersucht und ggf. ausgeführt werden.

Satz 50 *Es sei* $\Gamma_f = (Z, I, \mathcal{A}, \delta, \{B_0, \ldots, B_{m-1}\}, \emptyset)$ *ein FTS ohne starke Fairnessbedingungen, ferner sei* $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ *ein endlicher Ablauf von* Γ.

Dann ist jede Folge $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots$ *ein Ablauf von* Γ_f*, falls für alle* $k \geq n$ *die folgenden Bedingungen erfüllt sind:*

1. $A_k = B_{k \bmod m}$*, falls die Aktion* $B_{k \bmod m}$ *im Zustand* s_k *ausführbar ist.*
2. $(s_k, A_k, s_{k+1}) \in \delta$.

Bemerkung: Wegen der angenommenen Totalität von δ kann jeder endliche Ablauf von Γ_f zu einem Ablauf erweitert werden, der die Bedingungen von Satz 50 erfüllt. Das heißt, es reicht aus, den Scheduler erst ab einem beliebigen Zustand zu benutzen.

Beweis von Satz 50

Idee: Für den Nachweis der schwachen Fairness reicht es zu zeigen, dass jede Aktion B_i, die permanent ausführbar ist, irgendwann ausgeführt wird. Da der round-robin-Scheduler alle Aktionen zyklisch berücksichtigt, wird die Aktion B_i sicher ausgeführt, falls sie permanent ausführbar ist.

Formal: Angenommen, $\sigma = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots$ erfülle die Bedingungen (1) und (2), sei aber nicht schwach fair bezüglich der Aktion B_i.

Da $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ endlicher Ablauf von Γ ist und σ die Bedingung (2) erfüllt, ist σ sicher ein Ablauf von $\Gamma = (Z, I, \mathcal{A}, \delta)$.

Es sei $p \in \mathbb{N}$ so gewählt, dass B_i in allen Zuständen s_j mit $j \geq p$ ausführbar ist. Da σ die Fairnessbedingung für B_i verletzt, ist $A_k = B_i$ nur für endlich viele $k \in \mathbb{N}$.

Wähle $q \geq p$ so, dass $q \bmod m = i$ gilt und $A_k \neq B_i$ ist für alle $k \geq q$. Dann ist B_i ausführbar im Zustand s_q, also folgt nach Bedingung (1), dass $A_q = B_i$ gilt — Widerspruch. Q.E.D.

Ein round-robin-Scheduler ist ungeeignet zur Implementierung starker Fairnessbedingungen, da es möglich ist, dass eine Aktion, die immer wieder ausführbar ist, genau dann nicht ausführbar ist, wenn der Scheduler die Ausführbarkeit überprüft. Stattdessen kann man einen Scheduler mit „dynamischen Prioritäten“ verwenden, der Aktionen um so stärker priorisiert, je länger sie nicht ausgeführt wurden.

Satz 51 *Es sei $\Gamma_f = (Z, I, \mathcal{A}, \delta, \emptyset, \{B_0, \ldots, B_{m-1}\})$ ein FTS, das nur starke Fairnessbedingungen enthält, und $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ sei ein endlicher Ablauf von Γ.*

Dann ist jede Folge $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots$ ein Ablauf von Γ_f, falls eine Folge $\pi_n, \pi_{n+1}, \ldots$ von Permutationen von $\{0, \ldots, m-1\}$ existiert, so dass für alle $k \geq n$ die folgenden Bedingungen erfüllt sind:

1a. Falls $i \in \{0, \ldots, m-1\}$ existiert, so dass B_i im Zustand s_k ausführbar ist:

Sei $j \in \{0, \ldots, m-1\}$ minimal, so dass $B_{\pi_k(j)}$ in s_k ausführbar ist, dann gilt

$$A_k = B_{\pi_k(j)} \quad \textit{und} \quad \pi_{k+1}(i) = \left\{ \begin{array}{ll} \pi_k(i) & \textit{falls } i < j \\ \pi_k(i+1) & \textit{falls } j \leq i < m-1 \\ \pi_k(j) & \textit{falls } i = m-1 \end{array} \right\} \textit{ für alle } i \in \{0, \ldots, m-1\}$$

1b. Sonst ist A_k beliebige Aktion, die in s_k ausführbar ist, und $\pi_{k+1} = \pi_k$.

2. $(s_k, A_k, s_{k+1}) \in \delta$.

Bemerkung: Wieder kann jeder endliche Ablauf von Γ zu einem Ablauf erweitert werden, der die Bedingungen von Satz 51 erfüllt. Die intuitive Interpretation der etwas kompliziert formulierten Bedingungen ist, die Aktionen mit starker Fairness in einer Liste zu verwalten. Die Aktionen werden in der Reihenfolge, in der sie in der Liste stehen, auf ihre Ausführbarkeit überprüft. Sobald eine Aktion ausgeführt wird, erhält sie den letzten Listenplatz (und damit niedrigste Priorität in der folgenden Runde).

Beweis von Satz 51

Idee: Aktionen, die nicht ausgeführt werden, bewegen sich (falls überhaupt) auf der Liste nach vorne. Wird eine Aktion B_i nur endlich oft ausgeführt, muss sie daher irgendwann, etwa ab Zustand s_j, einen festen Listenplatz haben, und dies gilt dann auch für alle weiter vorne auf der Liste stehenden Aktionen. Aber dies kann nur dann eintreten, falls alle diese Aktionen ab s_j nie mehr ausführbar sind. Insbesondere ist B_i nur endlich oft ausführbar, und daher erfüllt der Ablauf die Bedingung der starken Fairness.

Formal: Angenommen, $\sigma = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots$ erfülle die Bedingungen (1a), (1b) und (2), sei aber kein Ablauf von Γ_f.

Da $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ endlicher Ablauf von Γ ist und σ die Bedingung (2) erfüllt, ist σ sicher ein Ablauf von $\Gamma = (Z, I, \mathcal{A}, \delta)$.

Angenommen, σ ist nicht stark fair bezüglich der Aktion B_i. Dann gibt es unendlich viele $m \in \mathbb{N}$, so dass B_i in s_m ausführbar ist, aber $A_k = B_i$ gilt nur für endlich viele k. Sei $p \geq n$ so gewählt, dass $A_k \neq B_i$ gilt für alle $k \geq p$. Betrachte die Folge $\pi_p, \pi_{p+1}, \ldots$. Da $A_k \neq B_i$ gilt, ist die Folge $j_p, j_{p+1}, \ldots$ mit $\pi_k(j_k) = i$ (schwach) monoton abnehmend, wird also schließlich stabil.

Wähle $q \geq p$ so, dass $j_k = j_q$ für alle $k \geq q$. Dann folgt $\pi_k(j) = \pi_q(j)$ für alle $j \leq j_q$ und $k \geq q$, und die Aktionen $B_{\pi_q(0)}, \ldots, B_{\pi_q(j_q)}$ sind in keinem Zustand s_k (für $k \geq q$) ausführbar — denn sonst wäre gemäß Bedingung (1a) $A_k = B_{\pi_k(j)}$ für ein $j \leq j_q$, und damit $j_{k+1} \neq j_k$.

Insbesondere ist $B_{\pi_q(j_q)} = B_i$ in keinem Zustand s_k (für $k \geq q$) ausführbar — Widerspruch. Q.E.D.

Interpretation. Die Sätze 50 und 51 besagen intuitiv:

- Ist Γ_f ein FTS, dessen zu Grunde liegendes Transitionssystem Γ ohne Fairnessbedingungen ausführbar ist (d.h. die Menge der Nachfolgerzustände zu jeder Aktion ist berechenbar), so können auch faire Abläufe von Γ_f systematisch erzeugt werden.

  Dabei reicht es sogar, den Scheduler erst ab einem beliebigen Zeitpunkt nach Beginn des Ablaufs zu verwenden.

- Allerdings werden auf diese Weise nicht alle fairen Abläufe von Γ_f generiert (selbst endliche FTSe haben i.a. überabzählbar unendlich viele verschiedene faire Abläufe).

- Der prioritätsbasierte Scheduler kann auch für FTSe verwendet werden, die sowohl starke wie schwache Fairnessbedingungen enthalten, da starke Fairness schwache Fairness impliziert.

9.3 Eigenschaften von Abläufen

Bei der Analyse von Transitionssystemen interessiert man sich in der Regel primär für Aussagen über ihre Abläufe, z.B.:

- Die Züge können zu keinem Zeitpunkt beide im Abschnitt onbridge sein.
- Wartet ein Zug vor dem Signal, so wird er zu einem späteren Zeitpunkt auf der Brücke sein.
- Wird ein Datum fehlerfrei empfangen, so wurde es zuvor vom Sender verschickt.
- Die Reihenfolge der empfangenen Daten entspricht der Reihenfolge beim Senden.
- Jedes gesendete Datum trifft irgendwann fehlerfrei beim Empfänger ein.

Außerdem interessiert man sich gelegentlich für Aussagen wie:

- Die Aktionen A und B stehen in Konflikt bzw. sie sind unabhängig.
- Von jedem Zustand aus kann ein Anfangszustand erreicht werden.
- Zwei Prozesse können kooperieren, um einen dritten Prozess auszusperren.

Solche Aussagen über die Struktur einer Transitionssystems bzw. mögliche alternative Abläufe klammern wir hier aus; sie werden z.B. in der Prozessalgebra und in Varianten temporaler Logik betrachtet.

Wir identifizieren im folgenden Eigenschaften mit der Menge der Zustands-Aktions-Folgen, welche die Eigenschaft erfüllen.

Definition 52 *Sei $\Gamma = (Z, I, \mathcal{A}, \delta)$ ein Transitionssystem. Eine* (Γ-)Eigenschaft *ist eine Menge von Folgen $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ mit $s_i \in Z$ und $A_i \in \mathcal{A}$. Das Transitionssystem Γ* erfüllt *die Eigenschaft P, wenn jeder Ablauf von Γ in P liegt.*

Eine „Eigenschaft“ P ist also eine Aussage, von der es sinnvoll ist zu fragen, ob sie auf einen Ablauf σ zutrifft ($\sigma \in P$) oder nicht.

Aussagen über die Existenz von Abläufen sind keine „Eigenschaften“ im Sinne von Definition 52.

Beispiele:

- Die Menge der Abläufe eines Transitionssystems Γ ist eine Γ-Eigenschaft.
- Menge der Zustands-Aktions-Folgen, die stark fair sind bzgl. Aktion A.
- Menge aller Folgen $s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$, so dass $s_n(y) = 1$ gilt für ein $n \in \mathbb{N}$.

Sicherheits- und Lebendigkeitseigenschaften

Man unterscheidet zwei grundlegende Klassen von Eigenschaften von Transitionssystemen. *Sicherheitseigenschaften* drücken intuitiv aus, dass zu keinem Zeitpunkt während des Ablaufs ein unerwünschtes Ereignis eintritt, während *Lebendigkeitseigenschaften* verlangen, dass (erwünschte) Ereignisse irgendwann bzw. immer wieder eintreten. Formal können diese Klassen, unabhängig von einer konkreten Beschreibungssprache, topologisch charakterisiert werden. Sie zeichnen sich durch unterschiedliche, jeweils charakteristische Beweisprinzipien aus (dazu mehr im Abschnitt 10). Im Bereich sequenzieller Programme entsprechen Sicherheitseigenschaften der partiellen Korrektheit (“es kommt nie vor, dass das Programm terminiert, ohne das gewünschte Ergebnis zu berechnen”), während Lebendigkeitseigenschaften als Verallgemeinerung der Terminierung von Programmen betrachtet werden können.

Informell wurden Sicherheits- und Lebendigkeitseigenschaften von Lamport 1980 folgendermaßen charakterisiert:

Sicherheitseigenschaft (safety property): *something bad never happens*

- Die Signalsteuerung gewährleistet, dass Züge nie auf der Brücke zusammenstoßen.
- jedes Datum wird entweder unverfälscht übertragen oder als fehlerhaft gekennzeichnet.

Lebendigkeitseigenschaft (liveness property): *something good eventually happens*

- Jeder wartende Zug wird die Brücke irgendwann befahren.
- Nachrichten werden unendlich oft fehlerfrei übertragen.
- Die Aktion γ wird irgendwann ausgeführt.

Formal werden Sicherheits- und Lebendigkeitseigenschaften folgendermaßen charakterisiert (nach Alpern und Schneider 1985).

Definition 53 *Sei Γ ein Transitionssystem.*

- *P ist eine* (Γ-)Sicherheitseigenschaft *genau dann, wenn für jede Folge $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ gilt:*

$$\sigma \in P \quad \textit{gdw.} \quad \textit{für jeden Präfix } s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \textit{ von } \sigma \textit{ gibt es eine Folge } \tau = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{B_n} t_{n+1} \xrightarrow{B_{n+1}} t_{n+2} \ldots \textit{ mit } \tau \in P$$

- *P ist eine* (Γ-)Lebendigkeitseigenschaft *genau dann, wenn sich jede endliche Folge $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ zu einer Folge $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \ldots \in P$ ergänzen lässt.*

Zusammenhang zu informeller Charakterisierung

- Eine Folge σ erfüllt eine Sicherheitseigenschaft P genau dann *nicht*, wenn es ein endliches Anfangsstück von σ gibt, das sich nicht zu einer Folge ergänzen lässt, die P erfüllt. Dieses endliche Anfangsstück endet mit dem Eintreffen des "unerwünschten Ereignisses", das nicht mehr behoben werden kann.
- Lebendigkeitseigenschaften schränken dagegen die Menge der endlichen Präfixe nicht ein: Was vor dem Eintreffen des "erwünschten Ereignisses" geschieht, ist irrelevant.

Vereinfachende Schreibweisen (Transitionssystem Γ sei vorausgesetzt)

- Für eine Folge $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ bezeichne $\sigma[..n]$ den Präfix $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$.
- Für Folgen $\rho = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ und $\sigma = s_n \xrightarrow{A_n} s_{n+1} \xrightarrow{A_{n+1}} s_{n+2} \ldots$ bezeichnet $\rho \circ \sigma$ die Konkatenation $s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_n} s_{n+1} \xrightarrow{A_{n+1}} s_{n+2} \ldots$.
- Für eine endliche Folge $\rho = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n$ schreiben wir $\rho \in P$, falls $\rho = \sigma[..n]$ gilt für ein $\sigma \in P$ und ein $n \in \mathbb{N}$.
- Σ^* und Σ^ω bezeichnen die Menge aller endlichen bzw. unendlichen Folgen von Zuständen und Aktionen.

Damit gilt: P ist Sicherheitseigenschaft genau dann, wenn für alle Folgen $\sigma \in \Sigma^\omega$ gilt:

Falls $\sigma[..n] \in P$ für alle $n \in \mathbb{N}$, so gilt auch $\sigma \in P$.

P ist Lebendigkeitseigenschaft genau dann, wenn $\sigma[..n] \in P$ gilt für alle Folgen $\sigma \in \Sigma^\omega$ und alle $n \in \mathbb{N}$.

Beispiel 103: Sicherheits- und Lebendigkeitseigenschaften

Die Menge der Abläufe eines Transitionssystems $\Gamma = (Z, I, \mathcal{A}, \delta)$ ohne Fairnessbedingungen ist eine Sicherheitseigenschaft.

Denn: Sei $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$

	σ Ablauf von Γ	
gdw.	$s_0 \in I$ und für alle $i \in \mathbb{N}$ gilt $(s_i, s_{i+1}) \in \delta$	[Definition 48]
gdw.	für alle $n \in \mathbb{N}$ gilt $s_0 \in I$ und für alle $i < n$ gilt $(s_i, s_{i+1}) \in \delta$	[Arithmetik]
gdw.	für alle $n \in \mathbb{N}$ gibt es $\tau \in \Sigma^\omega$, so dass $\sigma[..n] \circ \tau$ Ablauf von Γ	[wegen Totalität von δ]

Dagegen ist die Eigenschaft L „unendlich oft hat x den Wert 0“ eine Lebendigkeitseigenschaft.

Denn: Seien $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots \in \Sigma^\omega$ und $n \in \mathbb{N}$ beliebig, und sei s ein Zustand mit $s(x) = 0$. Betrachte die Folge $\tau = s_0 \xrightarrow{A_0} s_1 \ldots \xrightarrow{A_{n-1}} s_n \xrightarrow{A_0} s \xrightarrow{A_0} s \ldots$

Offenbar hat x in τ unendlich oft den Wert 0, also gilt $\sigma[..n] \in L$. ◀

Beachte: In Definition 53 wird nicht gefordert, dass die Zustands-Aktions-Folgen Abläufe von Γ sind.

Satz 54 (Sicherheits- und Lebendigkeitseigenschaften)

1. *Die Klasse der Sicherheitseigenschaften ist abgeschlossen unter beliebigen Durchschnitten.*
2. *Ist L eine Lebendigkeitseigenschaft, so ist auch jede Eigenschaft $M \supseteq L$ Lebendigkeitseigenschaft.*
3. *Für jede Eigenschaft P ist*

   $$\mathcal{C}(P) = \{\sigma \in \Sigma^\omega \mid \sigma[..n] \in P \text{ für alle } n \in \mathbb{N}\}$$

   die (bzgl. $\subseteq$) kleinste Sicherheitseigenschaft, die P enthält. $\mathcal{C}(P)$ heißt der Sicherheitsabschluss *(safety closure) von P.*
4. *P ist Sicherheitseigenschaft genau dann, wenn $\mathcal{C}(P) = P$ gilt.*
5. *P ist Lebendigkeitseigenschaft genau dann, wenn $\mathcal{C}(P) = \Sigma^\omega$ gilt.*
6. *Σ^ω ist die einzige Eigenschaft, die sowohl Sicherheits- als auch Lebendigkeitseigenschaft ist.*
7. *Jede Eigenschaft P lässt sich als Durchschnitt einer Sicherheits- und einer Lebendigkeitseigenschaft darstellen.*
8. *Ist S Sicherheitseigenschaft und P beliebige Eigenschaft, so gilt*

   $$P \subseteq S \quad \textit{gdw.} \quad \mathcal{C}(P) \subseteq S$$

Beweis.

1. Sei $\mathcal{S}$ eine Menge von Sicherheitseigenschaften, zu zeigen: $\bigcap \mathcal{S}$ ist Sicherheitseigenschaft.
   Sei $\sigma \in \Sigma^\omega$ beliebig mit $\sigma[..n] \in \bigcap \mathcal{S}$ für alle $n \in \mathbb{N}$.
   Also gilt $\sigma[..n] \in S$ für alle $n \in \mathbb{N}$ und alle $S \in \mathcal{S}$.
   Da jedes $S \in \mathcal{S}$ Sicherheitseigenschaft ist, folgt $\sigma \in S$ für alle $S \in \mathcal{S}$.
   Daher gilt $\sigma \in \bigcap \mathcal{S}$, und damit die Behauptung.

2. unmittelbar nach Definition.

3. Offenbar gilt $P \subseteq \mathcal{C}(P)$.

   - $\mathcal{C}(P)$ ist Sicherheitseigenschaft: Sei $\sigma \in \Sigma^\omega$ beliebig mit $\sigma[..n] \in \mathcal{C}(P)$ für alle $n \in \mathbb{N}$.
     Das heißt, für alle $n \in \mathbb{N}$ existiert $\tau_n \in \Sigma^\omega$ mit $\sigma[..n] \circ \tau_n \in \mathcal{C}(P)$.
     Nach Definition von $\mathcal{C}(P)$ folgt $\sigma[..n] \in P$ für alle $n \in \mathbb{N}$, also $\sigma \in \mathcal{C}(P)$.
   - Sei nun $S \supseteq P$ beliebige Sicherheitseigenschaft, zu zeigen ist $\mathcal{C}(P) \subseteq S$.
     Sei $\sigma \in \mathcal{C}(P)$ beliebig. Dann gilt $\sigma[..n] \in P$ für alle $n \in \mathbb{N}$, und wegen $P \subseteq S$ folgt $\sigma[..n] \in S$ für alle $n \in \mathbb{N}$.
     Da S Sicherheitseigenschaft ist, folgt $\sigma \in S$, was zu zeigen war.

4. "$\Rightarrow$": Sei P Sicherheitseigenschaft. Gemäß Aussage (3) reicht zu zeigen: dann gilt $\mathcal{C}(P) \subseteq P$.

   Sei also $\sigma \in \mathcal{C}(P)$, dann ist nach Definition $\sigma[..n] \in P$ für alle $n \in \mathbb{N}$. Da P Sicherheitseigenschaft ist, folgt $\sigma \in P$. Das genügt.

   "$\Leftarrow$": Gemäß (3) ist $\mathcal{C}(P)$ Sicherheitseigenschaft. Ist $\mathcal{C}(P) = P$, so ist also P ebenfalls Sicherheitseigenschaft.

5. P Lebendigkeitseigenschaft
   $\Leftrightarrow$ $\sigma[..n] \in P$ für alle $\sigma \in \Sigma^\omega$ und alle $n \in \mathbb{N}$ [Def. Lebendigkeitseigenschaft]
   $\Leftrightarrow$ $\sigma \in \mathcal{C}(P)$ für alle $\sigma \in \Sigma^\omega$ [Def. $\mathcal{C}(P)$]
   $\Leftrightarrow$ $\mathcal{C}(P) = \Sigma^\omega$

6. - Σ^ω ist offensichtlich Sicherheits- und Lebendigkeitseigenschaft.
   - Umgekehrt sei P Sicherheits- und Lebendigkeitseigenschaft, zu zeigen: $P = \Sigma^\omega$.
     Sei $\sigma \in \Sigma^\omega$ beliebig. Da P Lebendigkeitseigenschaft ist, gilt $\sigma[..n] \in P$ für alle $n \in \mathbb{N}$. Da P Sicherheitseigenschaft ist, folgt daraus $\sigma \in P$.

7. Setze

   $$L \;=_{\text{def}}\; P \cup (\Sigma^\omega \setminus \mathcal{C}(P))$$

   Offensichtlich gilt $P = \mathcal{C}(P) \cap L$.

   Zu zeigen bleibt: L ist Lebendigkeitseigenschaft.

   Sei $\sigma \in \Sigma^\omega$ beliebig, zu zeigen: $\sigma[..n] \in L$ gilt für alle $n \in \mathbb{N}$.

   Fall 1: $\sigma \notin \mathcal{C}(P)$. Dann ist $\sigma \in L$, also offenbar $\sigma[..n] \in L$ für alle $n \in \mathbb{N}$.

   Fall 2: $\sigma \in \mathcal{C}(P)$. Dann gilt für alle $n \in \mathbb{N}$, dass $\sigma[..n] \in P \subseteq L$.

8. "$\Rightarrow$": Gelte $P \subseteq S$ und sei $\sigma \in \mathcal{C}(P)$. Dann ist $\sigma[..n] \in P$ für alle $n \in \mathbb{N}$, und mit $P \subseteq S$ folgt $\sigma[..n] \in S$ für alle $n \in \mathbb{N}$. Da S nach Annahme Sicherheitseigenschaft ist, folgt $\sigma \in S$. Das genügt.

   "$\Leftarrow$": trivial wegen $P \subseteq \mathcal{C}(P)$ (gemäß 3). Q.E.D.

Beispiel 104: (vgl. Stoppuhr aus Beispiel 102)

Sei P die Menge aller Folgen $s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$, für die ein $n \geq 0$ existiert, so dass gilt:

- $s_i(y) = 0$ für alle $i \leq n$ und $s_i(y) = 1$ für alle $i > n$,
- $s_0(x) = 0$ und für alle $i < n$ gilt $s_{i+1}(x) \in \{s_i(x), s_i(x)+1\}$.

Intuitive Bedeutung: Eine Zeit lang gilt $y = 0$, danach $y = 1$. Solange $y = 0$ gilt, erhöht sich der Wert von x entweder um 1 oder er bleibt konstant.

Der Sicherheitsabschluss $\mathcal{C}(P)$ enthält genau die Folgen $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$, für die gilt:

- $\sigma \in P$ oder
- $s_0(x) = 0$ und für alle $i \in \mathbb{N}$ gilt $s_i(y) = 0$ und $s_{i+1}(x) \in \{s_i(x), s_i(x)+1\}$. ◀

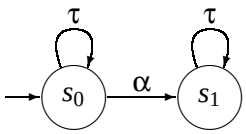
Satz 54.7 besagt, dass sich jede Spezifikation als Paar (S, L) aus einer Sicherheitseigenschaft S und einer Lebendigkeitseigenschaft L schreiben lässt. Eine natürliche Forderung besagt, dass L die gemäß S erlaubten endlichen Abläufe nicht einschränkt, dass sich also jede endliche Folge $\rho \in S$ zu einer Folge $\rho \circ \tau \in S \cap L$ ergänzen lässt.

Definition 55 *Sei S Sicherheitseigenschaft und L beliebige Eigenschaft.*

Das Paar (S, L) heißt maschinenabgeschlossen *(machine closed), wenn $\mathcal{C}(S \cap L) = S$ gilt.*

Beispiel 105: nicht maschinenabgeschlossene Spezifikation

Sei S die Menge aller Abläufe des folgenden Transitionssystems



und L die Menge aller Folgen, die unendlich oft den Zustand s_0 enthalten.

Der endliche Ablauf $s_0 \xrightarrow{\alpha} s_1$ lässt sich nicht zu einem Ablauf in $S \cap L$ ergänzen. Das Paar (S, L) ist also nicht maschinenabgeschlossen. ◀

Die Abläufe einer maschinenabgeschlossenen Spezifikation (S, L) erfüllen eine Sicherheitseigenschaft I genau dann, wenn $S \subseteq I$ gilt:

	$S \cap L \subseteq I$	
gdw.	$\mathcal{C}(S \cap L) \subseteq I$	[Satz 54.8]
gdw.	$S \subseteq I$	[(S, L) maschinenabgeschlossen]

Die zusätzliche Eigenschaft L wird also zum Beweis von Sicherheitseigenschaften nicht benötigt.

Nicht maschinenabgeschlossene Spezifikationen sind problematisch und werden in einigen Formalismen ganz ausgeschlossen.

Aus den Sätzen 50 und 51 folgt: FTSe mit endlich vielen Fairnessbedingungen sind maschinenabgeschlossen. Diese Aussage gilt sogar für abzählbar viele Fairnessbedingungen (ohne Beweis).

9.4 Zusammenfassung

- Reaktive Systeme werden formal durch Transitionssysteme modelliert.
- Abläufe eines Transitionssystems Γ sind Folgen $s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ von Zuständen und Aktionen, so dass s_0 ein Anfangszustand von Γ ist und alle Übergänge $s_i \xrightarrow{A_i} s_{i+1}$ der Zustandsübergangsrelation von Γ entsprechen.
- Zur Systemspezifikation werden Abläufe von Transitionssystemen meist weiter eingeschränkt.

  Insbesondere verwendet man Fairnessbedingungen, um unfaire Abläufe auszuschließen. Fairnessbedingungen besagen intuitiv, dass Aktionen, die genügend oft ausführbar sind, immer wieder ausgeführt werden.
- Bei der Analyse von Transitionssystemen ist man hauptsächlich an Aussagen über ihre Abläufe interessiert und identifiziert daher Eigenschaften mit Mengen von Abläufen.
- Eigenschaften können danach klassifiziert werden, wie viel Information durch die Betrachtung endlicher Abläufe gewonnen werden kann. Dabei treffe eine Eigenschaft P auf einen endlichen Ablauf ρ zu, falls es eine unendliche Folge σ mit $\rho \circ \sigma \in P$ gibt.
- Zwei grundlegende Klassen von Eigenschaften sind Sicherheitseigenschaften („*something bad never happens*“) und Lebendigkeitseigenschaften („*something good eventually happens*“).

  Eine Sicherheitseigenschaft trifft auf σ genau dann zu, wenn sie auf alle endlichen Anfangsstücke $\sigma[..n]$ von σ zutrifft. Sicherheitseigenschaften sind also durch endliche Abläufe vollständig bestimmt.

  Eine Lebendigkeitseigenschaft trifft auf alle endlichen Abläufe zu. Endliche Anfangsstücke geben also keinerlei Information auf das Zutreffen der Eigenschaft auf den Gesamtablauf.
- Zu jeder Eigenschaft P gibt es eine kleinste Sicherheitseigenschaft $\mathcal{C}(P) \supseteq P$, genannt der *Sicherheitsabschluss* von P.

  Jede Eigenschaft P kann als Durchschnitt einer Sicherheits- und einer Lebendigkeitseigenschaft dargestellt werden.
- Ein Paar (S,L) aus einer Sicherheitseigenschaft S und einer beliebigen Eigenschaft L heißt maschinenabgeschlossen, wenn $\mathcal{C}(S \cap L) = S$ gilt.

  In diesem Fall kann jeder endliche Ablauf, der S erfüllt, zu einem Ablauf von $S \cap L$ erweitert werden.
- Transitionssysteme mit schwachen und starken Fairnessbedingungen sind maschinenabgeschlossen.

10 Spezifikation von Transitionssystemen mit TLA

Temporale Logiken zur Beschreibung reaktiver Systeme und ihrer Eigenschaften wurden seit der 2. Hälfte der 1970er Jahre entwickelt (erste Veröffentlichungen 1977 von A. Pnueli und von F. Kröger).

Ab 1990 definierte L. Lamport die *Temporal Logic of Actions* (TLA), die sich durch folgende Charakteristika auszeichnet:

- natürliche Beschreibung von Transitionssystemen durch Formeln,

- Charakterisierung von Begriffen wie (Parallel-)Komposition, Implementierung, Hiding durch einen möglichst kleinen Satz logischer Operatoren,
- weitgehende Reduktion temporallogischer Verifikationsbedingungen auf prädikatenlogische Beweisverpflichtungen.

Ziele

- TLA als Beschreibungssprache für reaktive Systeme
- unterschiedliche Spezifikationsstile, z.B. asynchrone vs. synchrone Kommunikation, Interleaving vs. echte Parallelität
- grundlegende Regeln zur Systemverifikation in TLA
- Komposition, Verfeinerung und Hiding
- Implementierungsbeweise durch Verfeinerungsabbildungen

10.1 Die Logik TLA

Beispiel 106: Spezifikation einer Uhr mit Stunden- und Minutenanzeige

$$\begin{aligned} \mathit{IClk} &\equiv \mathit{hr} \in \{0,\ldots,23\} \wedge \mathit{min} \in \{0,\ldots,59\} \\ \mathit{Min} &\equiv \mathit{min} < 59 \wedge \mathit{min}' = \mathit{min}+1 \wedge \mathit{hr}' = \mathit{hr} \\ \mathit{Hr} &\equiv \mathit{min} = 59 \wedge \mathit{min}' = 0 \wedge \mathit{hr}' = (\mathit{hr}+1) \bmod 24 \\ \mathit{Tick} &\equiv \mathit{Min} \vee \mathit{Hr} \\ \mathit{Clock} &\equiv \mathit{IClk} \wedge \Box[\mathit{Tick}]_{\mathit{hr},\mathit{min}} \wedge \mathrm{WF}_{\mathit{hr},\mathit{min}}(\mathit{Tick}) \end{aligned}$$

Die Formel *Clock* beschreibt das Verhalten einer Uhr mit Stunden- und Minutenanzeige: die Anfangsbedingung wird durch die prädikatenlogische Formel *IClk* angegeben (in TLA auch *Zustandsformel* genannt, weil sie über einzelnen Zuständen ausgewertet wird). Die Teilformel $\Box[\mathit{Tick}]_{\mathit{hr},\mathit{min}}$ besagt, dass jeder Übergang, der die Variablen *hr* oder *min* verändert, die *Aktionsformel Tick* erfüllen muss. Aktionsformeln sind ebenfalls prädikatenlogische Formeln, die aber auch gestrichene Variablen enthalten können. Sie beschreiben Zustandsübergänge und werden daher über einem Paar von Zuständen ausgewertet. Dabei bezeichnen die ungestrichenen bzw. gestrichenen Variablen den Wert im Zustand vor bzw. nach dem Übergang. Die Formel *Tick* ist eine Disjunktion, dabei beschreibt *Min* den Übergang zur nächsten Minute und *Hr* den Stundenübergang.

Die Formel $\Box[\mathit{Tick}]_{\mathit{hr},\mathit{min}}$ verlangt nur, dass jeder Zustandsübergang, der *hr* oder *min* verändert, zulässig ist. Sie drückt nicht aus, dass solche Übergänge auch tatsächlich stattfinden. Sie wird daher ergänzt durch die Fairnessbedingung $\mathrm{WF}_{\mathit{hr},\mathit{min}}(\mathit{Tick})$, die erzwingt, dass "die Uhr nicht stehenbleibt". ◄

TLA-Spezifikationen haben gewöhnlich die Form

$$\mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L$$

Init ist eine prädikatenlogische Formel zur Beschreibung der Anfangszustände.

Next ist eine Formel mit gestrichenen und ungestrichenen Variablen zur Beschreibung der erlaubten Zustandsübergänge. *Next* hat meist die Form $A_1 \vee \ldots \vee A_n$, wobei die Formeln A_i die einzelnen Systemaktionen beschreiben.

v ist ein Tupel aller Variablen, die vom System verändert werden können.

L ist eine Konjunktion von Fairnessbedingungen $\mathrm{WF}_v(A_i)$ oder $\mathrm{SF}_v(A_i)$.

Man unterscheidet in TLA drei Klassen von Formeln:

- Zustandsformeln beschreiben Zustände,
- Aktionsformeln beschreiben Zustandsübergänge,
- temporallogische Formeln beschreiben Abläufe (Zustandsfolgen).

10.1.1 Zustandsformeln (state predicates)

Prädikatenlogische Formeln heißen in TLA Zustandsformeln.

Beispiele: $n > 0$, $q = \text{empty}$, $y = 0 \Leftrightarrow pc_1 = \text{“g”}$, $\exists k : n = m + k$

Dabei wird implizit eine (z.B. algebraisch beschriebene) Signatur *SIG* vorausgesetzt.

In TLA unterscheidet man zwischen *flexiblen* (zustandsabhängigen) und *rigiden* (zustandsunabhängigen) Variablen. Die rigiden Variablen entsprechen den üblichen Variablen der Prädikatenlogik. Die flexiblen Variablen modellieren Zustandskomponenten (z.B. Programmvariablen).

Formal: Die Menge X aller Variablen ist gegeben als disjunkte Vereinigung $X = X_f \cup X_r$ der Mengen X_f und X_r von flexiblen und rigiden Variablen.

Interpretation von Zustandsformeln

Vorausgesetzt sei eine gegebene *SIG*-Algebra A zur Interpretation der Sorten sowie der Funktions- und Prädikatensymbole.

Ein *Zustand* s ist eine (sortenrichtige) Belegung der flexiblen Variablen mit Werten.

Zustandsformeln werden interpretiert relativ zu einem Zustand s und einer Belegung ξ der rigiden Variablen.

Interpretation der Terme:

$$[\![x]\!]_{s,\xi} = \xi(x) \qquad \text{für rigide Variablen } x \in X_r$$
$$[\![v]\!]_{s,\xi} = s(v) \qquad \text{für flexible Variablen } v \in X_f$$
$$[\![f(t_1,\ldots,t_n)]\!]_{s,\xi} = f^A([\![t_1]\!]_{s,\xi},\ldots,[\![t_n]\!]_{s,\xi})$$

Interpretation der Formeln:

$$[\![p(t_1,\ldots,t_n)]\!]_{s,\xi} = \mathrm{T} \quad \text{gdw.} \quad ([\![t_1]\!]_{s,\xi},\ldots,[\![t_n]\!]_{s,\xi}) \in p^A$$
$$[\![\neg P]\!]_{s,\xi} = \mathrm{T} \quad \text{gdw.} \quad [\![P]\!]_{s,\xi} = \mathrm{F}$$
$$[\![P \wedge Q]\!]_{s,\xi} = \mathrm{T} \quad \text{gdw.} \quad [\![P]\!]_{s,\xi} = \mathrm{T} \text{ und } [\![Q]\!]_{s,\xi} = \mathrm{T}$$
$$[\![\exists x : P]\!]_{s,\xi} = \mathrm{T} \quad \text{gdw.} \quad [\![P]\!]_{s,\eta} = \mathrm{T} \text{ für ein } \eta \text{ mit } \eta(y) = \xi(y) \text{ für alle } y \in X_r \setminus \{x\}$$
$$[\![\exists v : P]\!]_{s,\xi} = \mathrm{T} \quad \text{gdw.} \quad [\![P]\!]_{t,\xi} = \mathrm{T} \text{ für ein } t \text{ mit } t(w) = s(w) \text{ für alle } w \in X_f \setminus \{v\}$$

Für eine gegebene Belegung ξ beschreiben Zustandsformeln also Mengen von Zuständen, z.B. die Anfangszustände oder die erreichbaren Zustände eines Transitionssystems.

10.1.2 Aktionsformeln (actions, transition predicates)

Aktionsformeln sind prädikatenlogische Formeln, die gestrichene und ungestrichene flexible Variablen frei enthalten dürfen.

Beispiele: $n' = n+1, \quad q' = \text{cons}(i', q), \quad \exists x : n = x + m'$

Interpretation von Aktionsformeln Aktionsformeln werden interpretiert relativ zu einem Paar (s,t) von Zuständen und einer Belegung ξ von X_r. Dabei werden ungestrichene flexible Variablen durch s und gestrichene flexible Variablen durch t interpretiert:

$$[\![v]\!]_{s,t,\xi} = s(v) \quad \text{für } v \in X_f$$
$$[\![v']\!]_{s,t,\xi} = t(v) \quad \text{für } v \in X_f$$
$$[\![x]\!]_{s,t,\xi} = \xi(x) \quad \text{für } x \in X_r$$

Die Fortsetzung der Semantik auf komplexe Terme und Formeln erfolgt analog zur Semantik der Zustandsformeln.

Für eine gegebene Belegung ξ von X_r beschreiben Aktionsformeln eine Menge von Zustandspaaren, z.B. die erlaubten Übergänge eines Transitionssystems.

Abkürzungen:

- Für einen Term t bzw. eine Zustandsformel P bezeichnen t' bzw. P' das Ergebnis der Ersetzung aller ungestrichenen freien Variablen $v \in X_f$ durch die entsprechenden gestrichenen Variablen v'. Dabei werden gebundene Variablen ggf. umbenannt, um Namenskonflikte zu vermeiden.

  Beispiele: $(v+1)' \equiv v' + 1$
  $(\exists x : n = x + m)' \equiv \exists x : n' = x + m'$
  $(\exists n' : n = n' + m)' \equiv \exists n1 : n' = n1 + m'$

- Für eine Aktionsformel A und Terme $t_1, \ldots, t_n$ (ohne gestrichene Variablen):

  $$[A]_{t_1,\ldots,t_n} \equiv A \vee (t_1' = t_1 \wedge \ldots \wedge t_n' = t_n)$$
  $$\langle A \rangle_{t_1,\ldots,t_n} \equiv A \wedge \neg (t_1' = t_1 \wedge \ldots \wedge t_n' = t_n)$$

  Ein Paar (s,t) von Zuständen erfüllt die Formel $[A]_{t_1,\ldots,t_n}$, wenn es A erfüllt oder wenn die Werte aller Terme $t_1, \ldots, t_n$ unverändert bleiben.

  Ein Paar (s,t) von Zuständen erfüllt die Formel $\langle A \rangle_{t_1,\ldots,t_n}$, wenn es A erfüllt und wenn sich der Wert mindestens eines Terms $t_1, \ldots, t_n$ ändert.

  Es gilt:

  $$\langle A \rangle_{t_1,\ldots,t_n} \Leftrightarrow \neg\, [\neg A]_{t_1,\ldots,t_n} \qquad \text{und} \qquad [A]_{t_1,\ldots,t_n} \Leftrightarrow \neg\, \langle \neg A \rangle_{t_1,\ldots,t_n}$$

- Für eine Aktionsformel A ist

  $$\text{ENABLED}\, A \equiv \exists v_1', \ldots, v_n' : A$$

  wobei $\{v_1', \ldots, v_n'\}$ die Menge der freien gestrichenen Variablen in A ist.

  Die Formel ENABLEDA ist (äquivalent zu einer) Zustandsformel; sie gilt in einem Zustand s genau dann, wenn es einen Zustand t gibt, so dass A im Zustandspaar (s,t) erfüllt ist.

Beispiele:

$$\begin{aligned}
\text{ENABLED}(n' = n+1) &\equiv \exists n' : n' = n+1 \\
\text{ENABLED}(q = \text{cons}(o', q')) &\equiv \exists o', q' : q = \text{cons}(o', q') \\
\text{ENABLED}(\exists z' : w = v' + z') &\equiv \exists v', z' : w = v' + z' \\
(\text{ENABLED}(q = \text{cons}(o', q')))' &\equiv \exists o1, q1 : q' = \text{cons}(o1, q1)
\end{aligned}$$

10.1.3 Temporallogische Formeln (temporal formulas)

Definition 56

- *Jede Zustandsformel P ist eine temporale Formel.*
- *Ist A Aktionsformel und sind $t_1, \ldots, t_m$ Terme (ohne gestrichene Variablen), so ist $\Box[A]_{t_1,\ldots,t_m}$ temporale Formel.*
- *Ist F temporale Formel, so ist $\Box F$* ("always *F*") *temporale Formel.*
- *Aussagenlogische Kombinationen temporaler Formeln sind temporale Formeln.*

Interpretation temporaler Formeln definiert über Folgen $\sigma = s_0 s_1 \ldots$ von Zuständen

$$\begin{aligned}
[\![P]\!]_{\sigma,\xi} = \text{T} \quad &\text{gdw.} \quad [\![P]\!]_{s_0,\xi} = \text{T} \qquad (P \text{ Zustandsformel}) \\
[\![\Box[A]_t]\!]_{\sigma,\xi} = \text{T} \quad &\text{gdw.} \quad [\![[A]_t]\!]_{s_n,s_{n+1},\xi} = \text{T} \text{ für alle } n \in \mathbb{N} \\
[\![\Box F]\!]_{\sigma,\xi} = \text{T} \quad &\text{gdw.} \quad [\![F]\!]_{\sigma[n..],\xi} = \text{T} \text{ für alle } n \in \mathbb{N} \\
[\![\neg F]\!]_{\sigma,\xi} = \text{T} \quad &\text{gdw.} \quad [\![F]\!]_{\sigma,\xi} = \text{F} \\
[\![F \wedge G]\!]_{\sigma,\xi} = \text{T} \quad &\text{gdw.} \quad [\![F]\!]_{\sigma,\xi} = \text{T} \text{ und } [\![G]\!]_{\sigma,\xi} = \text{T}
\end{aligned}$$

Dabei bezeichnet $\sigma[n..]$ die Restfolge $s_n s_{n+1} \ldots$ von σ ab dem Zustand s_n.

Abkürzungen:

- Ist *F* temporale Formel, so bezeichnet $\Diamond F$ („*eventually F*", „*sometime F*") die Formel

  $$\Diamond F \equiv \neg \Box \neg F$$

  Es ist $[\![\Diamond F]\!]_{\sigma,\xi} = \text{T}$ gdw. $[\![F]\!]_{\sigma[n..],\xi} = \text{T}$ für ein $n \in \mathbb{N}$.

- Analog schreiben wir

  $$\Diamond\langle A\rangle_{t_1,\ldots,t_m} \equiv \neg \Box[\neg A]_{t_1,\ldots,t_m}$$

  Die Zustandsfolge σ erfüllt $\Diamond\langle A\rangle_t$, wenn $[\![\langle A\rangle_{t_1,\ldots,t_m}]\!]_{s_n,s_{n+1},\xi} = \text{T}$ für ein $n \in \mathbb{N}$ gilt, also mindestens ein Paar aufeinanderfolgender Zustände die Formel *A* erfüllt und *t* verändert.

- Für temporale Formeln F, G definieren wir $F \rightsquigarrow G$ („*F leadsto G*") durch

  $$F \rightsquigarrow G \equiv \Box(F \Rightarrow \Diamond G)$$

  Die Formel $F \rightsquigarrow G$ ist wahr in σ, wenn für jeden Suffix $\sigma[n..]$, der *F* erfüllt, ein Suffix $\sigma[m..]$ mit $m \geq n$ existiert, der *G* erfüllt, also jedes Eintreffen von *F* von einem Eintreffen von *G* gefolgt wird.

- Klammerersparnis: $\Box$ und $\Diamond$ binden stärker als binäre aussagenlogische Operatoren.

  Zum Beispiel steht $\Box F \wedge \Diamond G$ für $(\Box F) \wedge (\Diamond G)$.

Umformungsregeln:

$$\neg\Diamond F \Leftrightarrow \neg\neg\Box\neg F \Leftrightarrow \Box\neg F \qquad \neg\Box F \Leftrightarrow \neg\Box\neg\neg F \Leftrightarrow \Diamond\neg F$$
$$\neg\Diamond\langle A\rangle_t \Leftrightarrow \Box[\neg A]_t \qquad \neg\Box[A]_t \Leftrightarrow \Diamond\langle\neg A\rangle_t$$
$$\Diamond\Diamond F \Leftrightarrow \Diamond F \qquad \Box\Box F \Leftrightarrow \Box F$$
$$\Diamond\Diamond\langle A\rangle_t \Leftrightarrow \Diamond\langle A\rangle_t \qquad \Box\Box[A]_t \Leftrightarrow \Box[A]_t$$
$$\Diamond F \vee \Diamond G \Leftrightarrow \Diamond(F \vee G) \qquad \Box F \wedge \Box G \Leftrightarrow \Box(F \wedge G)$$

„Unendlich oft“ und „irgendwann immer“

Es gilt: $[\![\Box\Diamond F]\!]_{\sigma,\xi} = \mathrm{T}$ gdw. für alle $m \in \mathbb{N}$ gibt es $n \geq m$ mit $[\![F]\!]_{\sigma[n..],\xi} = \mathrm{T}$.

Die Formel $\Box\Diamond F$ besagt also, dass F in der Zustandsfolge σ unendlich oft gilt. Analog fordert die Formel $\Box\Diamond\langle A\rangle_t$, dass die Aktion $\langle A\rangle_t$ unendlich oft ausgeführt wird.

Dagegen ist $[\![\Diamond\Box F]\!]_{\sigma,\xi} = \mathrm{T}$ gdw. ein $m \in \mathbb{N}$ existiert, so dass für alle $n \geq m$ gilt: $[\![F]\!]_{\sigma[n..],\xi} = \mathrm{T}$.

Die Formel $\Diamond\Box F$ besagt also, dass F in σ ab einem gewissen Zeitpunkt immer gilt. Analog fordert die Formel $\Diamond\Box[A]_t$, dass schließlich nur noch die Aktion $[A]_t$ stattfindet.

Für eine erfüllbare Zustandsformel P sind $\Box\Diamond P$ und $\Diamond\Box P$ Lebendigkeitseigenschaften.

Umformungsregeln:

$$\neg\Box\Diamond F \Leftrightarrow \Diamond\Box\neg F \qquad \Diamond\Box\Diamond F \Leftrightarrow \Box\Diamond F$$
$$\neg\Diamond\Box F \Leftrightarrow \Box\Diamond\neg F \qquad \Box\Diamond\Box F \Leftrightarrow \Diamond\Box F$$
$$\Box\Diamond F \vee \Box\Diamond G \Leftrightarrow \Box\Diamond(F \vee G) \qquad \Diamond\Box F \wedge \Diamond\Box G \Leftrightarrow \Diamond\Box(F \wedge G)$$
$$\Diamond\Box F \Rightarrow \Box\Diamond F$$

Fairness in TLA

In Abschnitt 9.2 hatten wir definiert:

- Ein Ablauf ist schwach fair bezüglich A, falls gilt: Ist A ab einem gewissen Zeitpunkt immer ausführbar, so wird A unendlich oft ausgeführt.
- Ein Ablauf ist stark fair bezüglich A, falls gilt: Ist A unendlich oft ausführbar, so wird A unendlich oft ausgeführt.

Für Aktionen der Form $\langle A\rangle_t$ können wir dies durch TLA-Formeln ausdrücken:

$$\mathrm{WF}_t(A) \equiv \Diamond\Box\,\mathrm{ENABLED}\langle A\rangle_t \Rightarrow \Box\Diamond\langle A\rangle_t$$
$$\mathrm{SF}_t(A) \equiv \Box\Diamond\,\mathrm{ENABLED}\langle A\rangle_t \Rightarrow \Box\Diamond\langle A\rangle_t$$

Äquivalente Formulierungen sind:

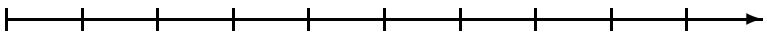
$$\mathrm{WF}_t(A) \Leftrightarrow \Box\Diamond\neg\,\mathrm{ENABLED}\langle A\rangle_t \vee \Box\Diamond\langle A\rangle_t \qquad \mathrm{SF}_t(A) \Leftrightarrow \Diamond\Box\neg\,\mathrm{ENABLED}\langle A\rangle_t \vee \Box\Diamond\langle A\rangle_t$$
$$\mathrm{WF}_t(A) \Leftrightarrow \Box\Diamond(\mathrm{ENABLED}\langle A\rangle_t \Rightarrow \Diamond\langle A\rangle_t) \qquad \mathrm{SF}_t(A) \Leftrightarrow \Diamond\Box(\mathrm{ENABLED}\langle A\rangle_t \Rightarrow \Diamond\langle A\rangle_t)$$

Beweis für $\mathrm{WF}_t(A)$

$$\begin{aligned}
\mathrm{WF}_t(A) &\Leftrightarrow \neg\,\Diamond\Box\,\mathrm{ENABLED}\langle A\rangle_t \vee \Box\Diamond\langle A\rangle_t \\
&\Leftrightarrow \Box\Diamond\neg\,\mathrm{ENABLED}\langle A\rangle_t \vee \Box\Diamond\langle A\rangle_t \\
&\Leftrightarrow \Box\Diamond\neg\,\mathrm{ENABLED}\langle A\rangle_t \vee \Box\Diamond\Diamond\langle A\rangle_t \\
&\Leftrightarrow \Box\Diamond(\neg\,\mathrm{ENABLED}\langle A\rangle_t \vee \Diamond\langle A\rangle_t) \\
&\Leftrightarrow \Box\Diamond(\mathrm{ENABLED}\langle A\rangle_t \Rightarrow \Diamond\langle A\rangle_t)
\end{aligned}$$

Q.E.D.

Beispiel 107: Semantik temporaler Formeln



x	0	0	3	7	0	1	1	0	2	$\cdots$	(immer $\neq 0$)
y	1	1	0	0	0	0	3	4	0	$\cdots$	(immer $= 0$)

Welche der folgenden Formeln gelten in diesem Ablauf?

- $\Box\neg\,(x = 0 \wedge y = 0)$
- $\Box[x = 0 \Rightarrow y' = 0]_{x,y}$
- $\Diamond(x = 7 \wedge y = 0)$
- $\Diamond\langle y = 0 \wedge x' = 0\rangle_y$
- $\Box\Diamond(y \neq 0)$
- $\Diamond\Box(x = 0 \Rightarrow y \neq 0)$
- $\Diamond\Box[\mathbf{false}]_y$ ◄

10.2 Spezifikationsstile in TLA

Ein reaktives System wird in TLA durch eine temporale Formel beschrieben, meist von der Form

$$\mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L$$

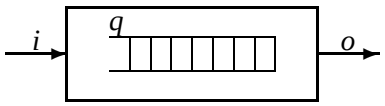
was der Modellierung durch ein Transitionssystem entspricht.

Zustandskomponenten (z.B. Programmvariablen, Kanäle) werden dabei durch flexible Variablen modelliert. Das Zusammenspiel zwischen Zustandskomponenten (inkl. Synchronisierung bzw. Parallelität von Aktionen) muss explizit durch geeignete Beschreibung der Aktionen codiert werden.

Für unterschiedliche Arten von Systemen existieren jeweils typische Spezifikationsstile. Die folgenden Beispiele illustrieren diese exemplarisch anhand der Modellierung eines FIFO-Puffers.

10.2.1 Einfache Interleaving-Spezifikationen

Beispiel 108: FIFO mit möglichem Datenverlust



$$
\begin{aligned}
\mathit{LQInit} &\equiv q = \text{empty} \wedge i = o \\
\mathit{LQEnq} &\equiv q' = \text{append}(q, \langle i' \rangle) \wedge o' = o \\
\mathit{LQDeq} &\equiv q \neq \text{empty} \wedge o' = \text{first}(q) \wedge q' = \text{rest}(q) \wedge i' = i \\
\mathit{LQNext} &\equiv \mathit{LQEnq} \vee \mathit{LQDeq} \\
\mathit{LQLive} &\equiv \text{WF}_{q,o}(\mathit{LQDeq}) \\
\mathit{LQSpec} &\equiv \mathit{LQInit} \wedge \Box[\mathit{LQNext}]_{q,o} \wedge \mathit{LQLive}
\end{aligned}
$$

- Die Variablen i und o repräsentieren die Ein- und Ausgabekanäle des FIFO-Puffers, q modelliert den (hier unbeschränkten) internen Puffer.
- Die Aktionen *LQEnq* und *LQDeq* modellieren Eingabe bzw. Ausgabe eines Datenwertes aus Sicht des Kanals. Sie schließen sich wechselseitig aus („Interleaving-Modell“).
- Die Variable i kommt nicht im Index von $\Box[\mathit{LQNext}]_{q,o}$ vor. Daher muss der Puffer nicht auf Änderungen von i reagieren. Gesendete Daten können also verloren gehen oder auch dupliziert werden.
- Die Fairnessbedingung für *LQDeq* garantiert, dass im Puffer gespeicherte Daten wieder ausgegeben werden. ◀

Einfache Interleaving-Spezifikationen haben i.a. die Form

$$\mathit{Init} \wedge \Box[\mathit{Next}]_{v,o} \wedge L$$

Dabei sind:

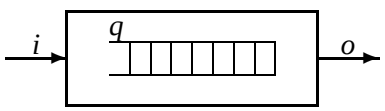
i, o, v : die Eingabe-, Ausgabe- und internen Variablen der Spezifikation.

Next : eine Aktionsformel, welche die erlaubten Übergänge beschreibt. Nur o und v kommen im Index von $\Box[\mathit{Next}]_{v,o}$ vor, daher sind beliebige Änderungen an den Eingabevariablen erlaubt, die o und v unverändert lassen („Umgebungsaktionen“). Umgekehrt sollte $\mathit{Next} \Rightarrow i' = i$ gelten, d.h. Umgebungs- und Systemaktionen schließen sich gegenseitig aus.

L : eine Konjunktion von Fairnessbedingungen der Form $\text{WF}_{v,o}(A)$ oder $\text{SF}_{v,o}(A)$. Meist ist *Next* eine Disjunktion $A_1 \vee \ldots \vee A_n$, und die Aktionen A sind unter den A_i.

10.2.2 Interleaving-Spezifikationen mit synchroner Kommunikation

Beispiel 109: FIFO mit synchroner Kommunikation



$$
\begin{aligned}
\mathit{SIQInit} &\equiv q = \text{empty} \wedge i = o \\
\mathit{SIQEnq} &\equiv i' \neq i \wedge q' = \text{append}(q, \langle i' \rangle) \wedge o' = o \\
\mathit{SIQDeq} &\equiv q \neq \text{empty} \wedge o' = \text{first}(q) \wedge q' = \text{rest}(q) \wedge i' = i \\
\mathit{SIQNext} &\equiv \mathit{SIQEnq} \vee \mathit{SIQDeq} \\
\mathit{SIQLive} &\equiv \text{WF}_{i,q,o}(\mathit{SIQDeq}) \\
\mathit{SIQSpec} &\equiv \mathit{SIQInit} \wedge \Box[\mathit{SIQNext}]_{i,q,o} \wedge \mathit{SIQLive}
\end{aligned}
$$

- Da i im Index der Übergangsrelation vorkommt, muss die Spezifikation auf Änderungen von i reagieren.
- Die Definitionen von *SIEnq* und *SIDeq* stellen sicher, dass der Kanal genau dann eine Eingabeaktion ausführt, wenn sich i ändert. In diesem Sinne modelliert *SIQSpec* synchrone Kommunikation zwischen Umgebung und Kanal.
- *SIEnq* und *SIDeq* schließen sich wieder aus (Interleaving-Modellierung).
- Jeder Ablauf, der *SIQSpec* erfüllt, erfüllt auch *LQSpec*. ◀

Interleaving-Spezifikationen mit synchroner Kommunikation koppeln Umgebungs- und Systemaktionen. Sie sind typischerweise von der Form

$$Init \wedge \Box[Next]_{i,v,o} \wedge L$$

Dabei gilt:

Next : lässt sich darstellen in der Form $Env \vee Sys$, wobei *Env* Umgebungsaktionen und *Sys* Systemaktionen beschreibt. Dabei sollte gelten

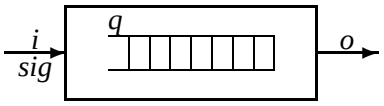
$$Env \Rightarrow o' = o \qquad \text{und} \qquad Sys \Rightarrow i' = i$$

d.h. Eingabe- und Ausgabeaktionen schließen sich gegenseitig aus. Die Eingabevariablen i kommen im Index von $\Box[Next]_{i,v,o}$ vor, um sicherzustellen, dass die Spezifikation auf Änderungen der Eingaben reagiert.

L : enthält Fairnessbedingungen an einzelne Systemaktionen.

10.2.3 Interleaving-Spezifikationen mit asynchroner Kommunikation

Beispiel 110: FIFO mit asynchroner Kommunikation



$$\begin{aligned}
AQInit &\equiv q = \text{empty} \wedge i = o \wedge sig = 0\\
AQEnv &\equiv sig = 0 \wedge sig' = 1 \wedge q' = q \wedge o' = o\\
AQEnq &\equiv sig = 1 \wedge sig' = 0 \wedge q' = \text{append}(q, \langle i' \rangle) \wedge o' = o \wedge i' = i\\
AQDeq &\equiv q \neq \text{empty} \wedge o' = \text{first}(q) \wedge q' = \text{rest}(q) \wedge sig' = sig \wedge i' = i\\
AQNext &\equiv AQEnv \vee AQEnq \vee AQDeq\\
AQLive &\equiv \text{WF}_{q,i,o,sig}(AQEnq) \wedge \text{WF}_{q,i,o,sig}(AQDeq)\\
AQSpec &\equiv AQInit \wedge \Box[AQNext]_{q,i,o,sig} \wedge AQLive
\end{aligned}$$

- Ein zusätzliches Bit *sig* modelliert „Handshake-Protokoll“ auf Eingabekanal.
- Die Aktion *Enq* wurde in zwei Teilaktionen aufgespalten: die „Umgebungsaktion“ *AQEnv* erlaubt das Senden eines neuen Werts, falls $sig = 0$ gilt. Die „Systemaktion“ *AQEnq* übernimmt den gesendeten Wert und übergibt die Kontrolle wieder an den Sender.
- Die Fairnessbedingung an *AQEnq* garantiert, dass jeder gesendete Wert vom Kanal übernommen wird.

- Jeder Ablauf, der *AQSpec* erfüllt, erfüllt auch *LQSpec*. ◄

Asynchrone Kommunikation muss explizit modelliert werden. Dazu werden „Interface-Variablen“ wie *sig* eingeführt, die sowohl von Umgebungs- wie von Systemaktionen verändert werden.

Gemeinsame Aktionen A von Umgebung und System (wie das Senden einer Nachricht) zerfallen in zwei Aktionen A_{env} und A_{sys}. Dabei modelliert A_{env} den Umgebungsschritt, es gilt daher

$$A_{env} \Rightarrow v' = v \wedge o' = o$$

und A_{sys} modelliert die Reaktion des Systems auf den Umgebungsschritt, es gilt daher in der Regel

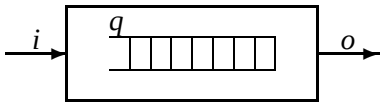
$$A_{sys} \Rightarrow i' = i \wedge o' = o$$

Die Interface-Variablen stellen sicher, dass sich A_{env} und A_{sys} abwechseln.

Fairnessbedingungen an A_{sys} garantieren, dass das System auf jeden Umgebungsschritt reagiert.

10.2.4 Noninterleaving-Spezifikationen

Beispiel 111: FIFO mit gleichzeitiger Ein- und Ausgabe, synchrone Kommunikation



$$\begin{array}{lcl}
SNQInit & \equiv & q = \text{empty} \wedge i = o \\
di & \equiv & \textbf{if } i' = i \textbf{ then empty else } \langle i' \rangle \\
do & \equiv & \textbf{if } o' = o \textbf{ then empty else } \langle o' \rangle \\
SNQEnq & \equiv & i' \neq i \wedge \text{append}(q, di) = \text{append}(do, q') \\
SNQDeq & \equiv & q \neq \text{empty} \wedge o' = \text{first}(q) \wedge \text{append}(q, di) = \text{append}(do, q') \\
SNQLive & \equiv & \text{WF}_o(SNQDeq) \\
SNQSpec & \equiv & SNQInit \ \wedge \Box[SNQEnq]_i \wedge \Box[SNQEnq \vee SNQDeq]_q \\
 & & \qquad \wedge \Box[SNQDeq]_o \wedge SNQLive
\end{array}$$

- Die Spezifikation gibt eine Übergangsrelation pro Variable (bzw. pro Tupel zusammengehörender Variablen) an.
- Ein- und Ausgabe können im selben Schritt stattfinden („Noninterleaving-Modell“), falls der Puffer nicht leer ist. Die dadurch bewirkten Änderungen werden über eine „Transitionsinvariante“ zusammengefasst.
- Jeder Ablauf, der *SIQSpec* erfüllt, erfüllt auch *SNQSpec*: gleichzeitige Ein- und Ausgabe wird nicht gefordert, aber zugelassen. ◄

Noninterleaving-Spezifikationen (Spezifikationen mit echter Parallelität) erlauben gleichzeitige Schritte von System und Umgebung. Sie können in der Form

$$Init \wedge \Box[Env]_i \wedge \Box[Mod]_v \wedge \Box[Out]_o \wedge L$$

geschrieben werden.

Env, *Mod*, *Out* : modellieren Eingabe-, interne und Ausgabeaktionen. Die Indizes *i*, *v* und *o* garantieren, dass alle Änderungen an den betreffenden Systemkomponenten diesen Aktionen entsprechen. Diese Aktionen können miteinander über gemeinsame Variablen synchronisiert werden.

Bedingungen wie $Env \Rightarrow o' = o$ werden nicht verlangt, daher sind gleichzeitige Aktionen von System und Umgebung möglich.

L : enthält Fairnessbedingungen an Teilaktionen von *Mod* oder *Out*.

10.2.5 Zusammenfassung

- TLA erlaubt die Beschreibung von Systemen auf unterschiedliche Weise. Programmierkonzepte (z.B. Kommunikation über gemeinsame Variablen oder durch Nachrichtenaustausch) werden nicht fest vorgegeben, sondern müssen explizit modelliert werden.
- Für häufige Klassen von Systemen existieren typische Spezifikationsstile, die Anhaltspunkte zum Schreiben eigener Spezifikationen bieten.
- Interleaving-Spezifikationen sind in der Regel einfacher zu schreiben. Noninterleaving-Spezifikationen spiegeln manchmal besser die Realität wider und sind einfacher zusammenzusetzen (vgl. Kapitel 11.2).

10.3 Verifikationsregeln

TLA dient nicht nur zur Spezifikation reaktiver Systeme, sondern erlaubt auch die Formulierung und den Nachweis von Systemeigenschaften. Werden das (faire) Transitionssystem Γ_f und die Eigenschaft *P* durch TLA-Formeln *Spec* und *Prop* beschrieben, so erfüllt das Transitionssystem die Eigenschaft genau dann, wenn

$$[\![Spec \Rightarrow Prop]\!]_{\sigma,\xi} = \mathrm{T}$$

gilt für alle Zustandsfolgen σ (über der fest gewählten Algebra oder einer Klasse von Algebren, die z.B. durch eine algebraische Spezifikation beschrieben wird) und alle Belegungen ξ.

Im folgenden Kapitel werden Beweisregeln zum Nachweis von in TLA formulierten Sicherheits- und Lebendigkeitseigenschaften vorgestellt. Diese ermöglichen Beweise von Aussagen der Form

$$Spec \Rightarrow Prop$$

für Systemspezifikationen *Spec* und typische Klassen von Eigenschaften *Prop*.

10.3.1 Beweise von Invarianten

Invarianten sind Formeln der Gestalt $\Box P$ mit einer Zustandsformel *P*. Ein Ablauf erfüllt $\Box P$ genau dann, wenn *P* in jedem Zustand des Ablaufs erfüllt ist. Ein Transitionssystem erfüllt eine Invariante $\Box P$ genau dann, wenn alle erreichbaren Zustände *P* erfüllen.

Invarianten sind die Basis für alle weiteren Verifikationsschritte und drücken die intuitive Korrektheitsidee des Algorithmus formal aus. Die grundlegende Regel zum Beweis von Invarianten für TLA-Spezifikationen ist

$$\text{(INV1)} \quad \frac{P \wedge Next \Rightarrow P' \qquad P \wedge v' = v \Rightarrow P'}{P \wedge \Box[Next]_v \Rightarrow \Box P}$$

Eine nützliche Verallgemeinerung (für Noninterleaving-Spezifikationen) ist

$$\text{(INV1)}_m \quad \frac{P \wedge [N_1]_{v_1} \Rightarrow P' \qquad \cdots \qquad P \wedge [N_m]_{v_m} \Rightarrow P'}{P \wedge \Box[N_1]_{v_1} \wedge \ldots \wedge \Box[N_m]_{v_m} \Rightarrow \Box P}$$

Beweis der Korrektheit von INV1. Seien $P \wedge \mathit{Next} \Rightarrow P'$ sowie $P \wedge v' = v \Rightarrow P'$ gültig.

D.h. für alle Zustände (der gegebenen Algebra oder Klasse von Algebren) s, t und alle Belegungen ξ gilt

$$[\![P \wedge \mathit{Next} \Rightarrow P']\!]_{s,t,\xi} = \mathrm{T} \qquad \text{und} \qquad [\![P \wedge v' = v \Rightarrow P']\!]_{s,t,\xi} = \mathrm{T}$$

Sei $\sigma = s_0 s_1 \ldots$ eine Zustandsfolge und ξ Belegung von X_r. Zu zeigen ist

$$[\![P \wedge \Box[\mathit{Next}]_v \Rightarrow \Box P]\!]_{\sigma,\xi} = \mathrm{T}$$

Sei also $[\![P \wedge \Box[\mathit{Next}]_v]\!]_{\sigma,\xi} = \mathrm{T}$. Wir zeigen, dass $[\![P]\!]_{s_n,\xi} = \mathrm{T}$ gilt für alle $n \in \mathbb{N}$.

$n = 0$: $[\![P]\!]_{s_0,\xi} = \mathrm{T}$ folgt unmittelbar nach Annahme.

$n \to n+1$: Nach Induktionsvoraussetzung gilt $[\![P]\!]_{s_n,\xi} = \mathrm{T}$.

Ferner ist nach Annahme $[\![[\mathit{Next}]_v]\!]_{s_n,s_{n+1},\xi} = \mathrm{T}$.

Gilt $[\![\mathit{Next}]\!]_{s_n,s_{n+1},\xi} = \mathrm{T}$, so folgt $[\![P]\!]_{s_{n+1},\xi} = \mathrm{T}$ aus der Annahme $P \wedge \mathit{Next} \Rightarrow P'$.

Ansonsten muss $[\![v' = v]\!]_{s_n,s_{n+1},\xi} = \mathrm{T}$ gelten, und die Behauptung folgt aus $P \wedge v' = v \Rightarrow P'$. Q.E.D.

Beispiel 112: Invariantenbeweis für den Puffer mit synchroner Eingabe (vgl. Bsp. 109)

Im Puffer der synchronen Queue sind aufeinanderfolgende Elemente verschieden.

Sei $\mathit{Diff} \equiv \wedge \forall k : 1 \leq k < \mathrm{length}(q) \Rightarrow q[k] \neq q[k+1]$
$\wedge\ \mathrm{length}(q) > 0 \Rightarrow q[\mathrm{length}(q)] = i$

(mit Selektorfunktion $q[k]$ zum Zugriff auf die einzelnen Elemente der Folge).

Dann gelten (Beweis durch Nachrechnen mit Hilfe von Datenaxiomen):

$\mathit{Diff} \wedge \mathit{SIQEnq} \Rightarrow \mathit{Diff}'$
$\mathit{Diff} \wedge \mathit{SIQDeq} \Rightarrow \mathit{Diff}'$
$\mathit{Diff} \wedge (i,q,o)' = (i,q,o) \Rightarrow \mathit{Diff}'$

Also folgt mit (INV1) auch

$$\mathit{Diff} \wedge \Box[\mathit{SIQNext}]_{i,q,o} \Rightarrow \Box \mathit{Diff}$$

Wegen $\mathit{SIQInit} \Rightarrow \mathit{Diff}$ folgt daher $\mathit{SIQSpec} \Rightarrow \Box \mathit{Diff}$. ◀

Bemerkung: Die Regel (INV1) erlaubt den Beweis induktiver Invarianten. In der Praxis muss zum Beweis einer Invariante P eine stärkere induktive Invariante Q gefunden werden.

$$\text{(INV)} \quad \frac{\mathit{Init} \Rightarrow Q \qquad Q \wedge [\mathit{Next}]_v \Rightarrow Q' \qquad Q \Rightarrow P}{\mathit{Init} \wedge \Box[\mathit{Next}]_v \Rightarrow \Box P}$$

Das Finden induktiver Invarianten erfordert Kreativität. Der eigentliche Beweis erfolgt dagegen schematisch und erfordert kein temporallogisches Schließen.

Ein Vorteil bei der Systementwicklung durch schrittweise Verfeinerung besteht darin, dass die Invariante (für den jeweiligen Abstraktionsgrad) mit dokumentiert wird—vgl. etwa die Schema-Invarianten in Z. Das nachträgliche Auffinden der Invariante wird so vermieden.

Exkurs. Verstärkung von Invarianten durch Berechnung schwächster Vorbedingungen.

Zu einer Aktionsformel A und einer Zustandsformel P bezeichne $\mathrm{wp}(P,A)$ die Zustandsformel

$$\mathrm{wp}(P,A) \equiv \forall v_1', \ldots, v_n' : A \Rightarrow P'$$

wobei $v_1', \ldots, v_n'$ alle in A und P' frei vorkommenden gestrichenen Variablen seien. $\mathrm{wp}(P,A)$ heißt *schwächste Vorbedingung (weakest precondition) von P bezüglich A.*

Es ist $[\![\mathrm{wp}(P,A)]\!]_{s,\xi} = \mathrm{T}$ genau dann, wenn für alle Zustände t mit $[\![A]\!]_{s,t,\xi} = \mathrm{T}$ folgt, dass $[\![P]\!]_{t,\xi} = \mathrm{T}$ ist. $\mathrm{wp}(P,A)$ charakterisiert also die Menge der Zustände, deren sämtliche A-Nachfolger die Formel P erfüllen.

Beispiele:

$$\begin{aligned}
\mathrm{wp}(x = 5, x' = x+1) &\equiv \forall x' : x' = x+1 \Rightarrow x' = 5 \\
&\Leftrightarrow x = 4 \\
\mathrm{wp}(y \in S, S' = S \cup T \wedge y' = y) &\equiv \forall y', S' : S' = S \cup T \wedge y' = y \Rightarrow y' \in S' \\
&\Leftrightarrow y \in S \vee y \in T \\
\mathrm{wp}(x > 0, x' = 0) &\equiv \forall x' : x' = 0 \Rightarrow x' > 0 \\
&\Leftrightarrow \textbf{false}
\end{aligned}$$

Mit dieser Schreibweise kann (INV) umformuliert werden:

$$\frac{\mathit{Init} \Rightarrow Q \qquad Q \Rightarrow \mathrm{wp}(Q, [\mathit{Next}]_v) \qquad Q \Rightarrow P}{\mathit{Init} \wedge \Box[\mathit{Next}]_v \Rightarrow \Box P}$$

Eine nützliche Heuristik zum Finden induktiver Invarianten ist folgende:

1. Start mit der zu beweisenden Invariante: $Q_0 \equiv P$.
2. Versuche, die Formeln $Q_i \wedge A \Rightarrow Q_i'$ für jede Teilaktion (Disjunktionsglied) A von $[\mathit{Next}]_v$ zu beweisen.

   Schlägt ein Beweis fehl, setze $Q_{i+1} \equiv Q_i \wedge \mathrm{wp}(Q_i, A)$.
3. Wiederhole Schritt 2, bis
   - entweder alle Teilbeweise gelingen und außerdem $\mathit{Init} \Rightarrow Q_i$ gilt; dann ist Q_i eine induktive Invariante, die P impliziert, oder
   - Q_i widersprüchlich geworden ist oder nicht aus *Init* folgt; dann ist P keine Invariante des Systems.

10.3.2 Lebendigkeit 1: Ausnutzen von Fairnessbedingungen

Fairnessbedingungen garantieren, dass Aktionen irgendwann ausgeführt werden. Sie sind die Basis für Lebendigkeitsbeweise.

Beispiel: Die schwache Fairnessbedingung an die Aktion *SIQDeq* garantiert, dass das erste Element eines nichtleeren FIFO-Puffers irgendwann ausgegeben wird. Es gilt also

$$SIQSpec \Rightarrow ((q \neq \text{empty} \wedge \text{first}(q) = x) \rightsquigarrow o = x)$$

Dabei ist x eine rigide Variable. Die Aussage gilt für beliebige Belegungen von x (implizite Allquantifizierung).

Diese Überlegung (für Zustandsformeln P, Q) wird formalisiert durch folgende Regel:

$$\text{(WF1)} \quad \frac{\begin{array}{c} P \wedge [Next]_v \Rightarrow P' \vee Q' \\ P \wedge \langle Next \wedge A \rangle_v \Rightarrow Q' \\ P \Rightarrow \text{ENABLED}\langle A \rangle_v \end{array}}{\Box[Next]_v \wedge \text{WF}_v(A) \Rightarrow (P \rightsquigarrow Q)}$$

Wie bei der Invariantenregel (INV1) sind die Prämissen von (WF1) wieder nicht-temporale Formeln.

Beweis der Korrektheit von (WF1). Seien die Prämissen von (WF1) gültig, sei $\sigma = s_0 s_1 \ldots$ eine Zustandsfolge und gelte

$$[\![\Box[Next]_v \wedge \text{WF}_v(A)]\!]_{\sigma,\xi} = \text{T}$$

Zu zeigen ist

$$[\![P \rightsquigarrow Q]\!]_{\sigma,\xi} = \text{T}$$

Sei also $n \in \mathbb{N}$ beliebig und gelte $[\![P]\!]_{s_n,\xi} = \text{T}$. Angenommen, $[\![Q]\!]_{s_m,\xi} = \text{F}$ für alle $m \geq n$.

1. Für alle $m \geq n$ ist $[\![P]\!]_{s_m,\xi} = \text{T}$.
   Beweis durch Induktion nach $m \geq n$:
   - Für $m = n$ gilt die Aussage nach Voraussetzung.
   - Gelte $[\![P]\!]_{s_m,\xi} = \text{T}$. Nach Voraussetzung ist $[\![[Next]_v]\!]_{s_m,s_{m+1},\xi} = \text{T}$, also folgt mit Annahme
     $$P \wedge [Next]_v \Rightarrow P' \vee Q'$$
     dass $[\![P]\!]_{s_{m+1},\xi} = \text{T}$ oder $[\![Q]\!]_{s_{m+1},\xi} = \text{T}$ gilt. Mit der Widerspruchsannahme folgt $[\![P]\!]_{s_{m+1},\xi} = \text{T}$.
2. Für alle $m \geq n$ ist $[\![\text{ENABLED}\langle A \rangle_v]\!]_{s_m,\xi} = \text{T}$.
   Dies folgt aus Aussage (1) und der Prämisse $P \Rightarrow \text{ENABLED}\langle A \rangle_v$.
3. Für ein $m \geq n$ ist $[\![\langle A \rangle_v]\!]_{s_m,s_{m+1},\xi} = \text{T}$.
   Gilt wegen Aussage (2) und Fairnessannahme $[\![\text{WF}_v(A)]\!]_{\sigma,\xi} = \text{T}$.
4. Für jedes m wie in (3) folgt $[\![Q]\!]_{s_{m+1},\xi} = \text{T}$.
   Denn es ist $[\![P \wedge [Next]_v \wedge \langle A \rangle_v]\!]_{s_m,s_{m+1},\xi} = \text{T}$, und mit der Prämisse
   $$P \wedge \langle Next \wedge A \rangle_v \Rightarrow Q'$$
   folgt die Behauptung.
5. Widerspruch, also muss ein $m \geq n$ mit $[\![Q]\!]_{s_m,\xi} = \text{T}$ existieren. Q.E.D.

Beispiel 113: zur Anwendung von (WF1)

Für die Spezifikation *SIQSpec* aus Beispiel 109 zeigen wir

$$SIQSpec \Rightarrow (\underbrace{(q \neq \text{empty} \wedge \text{first}(q) = x)}_{P} \rightsquigarrow \underbrace{o = x}_{Q})$$

Gemäß der Regel (WF1) ist zu beweisen

(1a) $P \wedge SIQEnq \Rightarrow P'$
(1b) $P \wedge SIQDeq \Rightarrow Q'$
(1c) $P \wedge (i,o,q)' = (i,o,q) \Rightarrow P'$
(2) $P \wedge \langle SIQNext \wedge SIQDeq \rangle_{i,q,o} \Rightarrow Q'$
(3) $P \Rightarrow \text{ENABLED}\langle SIQDeq \rangle_{i,q,o}$

Alle diese Aussagen folgen unmittelbar auf Grund der Definitionen der Aktionsformeln mit Hilfe von Datenaxiomen. Mit (WF1) folgt daher

$$\Box[SIQNext]_{i,q,o} \wedge \text{WF}_{i,q,o}(SIQDeq) \Rightarrow (P \rightsquigarrow Q)$$

und damit die Aussage. ◄

Beweisregel für starke Fairness

Eine starke Fairnessannahme für eine Aktion $\langle A \rangle_v$ garantiert, dass die Aktion irgendwann ausgeführt wird, falls sie immer wieder (aber nicht unbedingt kontinuierlich) ausführbar ist. Die dritte Prämisse

$$P \Rightarrow \text{ENABLED}\langle A \rangle_v$$

der Regel (WF1) ist daher zu stark, falls starke Fairness für $\langle A \rangle_v$ angenommen wird.

Beispiel: Gegeben sei das folgende Programm (in Pseudocode) zum gegenseitigen Ausschluss zweier Prozesse durch ein Semaphor *s*:

```
semaphore s = 1;
cobegin
   loop                                  loop
     ncs1:  (* nichtkrit. Abschnitt *)     ncs2:  (* nichtkrit. Abschnitt *)
     try1:  P(s);                          try2:  P(s);
     cs1:   (* kritischer Abschnitt *)  ||   cs2:   (* kritischer Abschnitt *)
     exit1: V(s)                           exit2: V(s)
   end                                   end
coend
```

Wird starke Fairness für die *P*-Operationen angenommen, so gilt

$$pc_1 = \text{try} \rightsquigarrow pc_1 = \text{cs}$$

obwohl $pc_1 = \text{try}$ nicht die Ausführbarkeit der *P*-Operation impliziert.

Diese Intuition wird formalisiert durch folgende Regel:

$$\text{(SF1)} \quad \frac{\begin{array}{c} P \wedge [Next]_v \Rightarrow P' \vee Q' \\ P \wedge \langle Next \wedge A \rangle_v \Rightarrow Q' \\ \Box P \wedge \Box[Next]_v \wedge \Box F \Rightarrow \Diamond \,\text{ENABLED}\langle A \rangle_v \end{array}}{\Box[Next]_v \wedge \text{SF}_v(A) \wedge \Box F \Rightarrow (P \rightsquigarrow Q)}$$

- Die ersten beiden Prämissen sind wie in (WF1).
- Die dritte Prämisse ist hier eine temporale Formel (und i.w. von derselben Form wie die Konklusion). Ihr Beweis benötigt Anwendung weiterer temporaler Verifikationsregeln.
- Die Formel F ist eine beliebige temporale Formel, z.B. Konjunktion von
  - Invarianten
  - weiteren Fairnessbedingungen: es gilt
    $\mathrm{WF}_v(B) \Leftrightarrow \Box\mathrm{WF}_v(B)$ und $\mathrm{SF}_v(B) \Leftrightarrow \Box\mathrm{SF}_v(B)$
  - bereits zuvor bewiesenen „leadsto“-Aussagen

Beweis der Korrektheit von (SF1). Seien die Prämissen von (SF1) gültig, sei $\sigma = s_0 s_1 \ldots$ eine Zustandsfolge und gelte

$$[\![\Box[Next]_v \wedge \mathrm{SF}_v(A) \wedge \Box F]\!]_{\sigma,\xi} = \mathrm{T}$$

Zu zeigen ist

$$[\![P \rightsquigarrow Q]\!]_{\sigma,\xi} = \mathrm{T}$$

Sei also $n \in \mathbb{N}$ beliebig und gelte $[\![P]\!]_{s_n,\xi} = \mathrm{T}$. Angenommen, $[\![Q]\!]_{s_m,\xi} = \mathrm{F}$ für alle $m \geq n$.

1. Für alle $m \geq n$ ist $[\![P]\!]_{s_m,\xi} = \mathrm{T}$.
   Folgt wie im Beweis der Korrektheit von (WF1).
2. Für alle $m \geq n$ gilt
   $$[\![\Box P \wedge \Box[Next]_v \wedge \Box F]\!]_{\sigma[m..],\xi} = \mathrm{T}$$
   Folgt unmittelbar aus (1) und Voraussetzung gemäß TLA-Semantik
3. Für alle $m \geq n$ gibt es ein $k \geq m$ mit $[\![\mathrm{ENABLED}\langle A\rangle_v]\!]_{s_k,\xi} = \mathrm{T}$.
   Aus (2) mit Prämisse
   $$\Box P \wedge \Box[Next]_v \wedge \Box F \Rightarrow \Diamond\,\mathrm{ENABLED}\langle A\rangle_v$$
4. Es gilt $[\![\Box\Diamond\,\mathrm{ENABLED}\langle A\rangle_v]\!]_{\sigma,\xi} = \mathrm{T}$.
   Folgt unmittelbar aus (3).
5. Für ein $m \geq n$ ist $[\![\langle A\rangle_v]\!]_{s_m,s_{m+1},\xi} = \mathrm{T}$.
   Gilt wegen Aussage (4) und Fairnessannahme $[\![\mathrm{SF}_v(A)]\!]_{\sigma,\xi} = \mathrm{T}$.
6. Für jedes m wie in (5) folgt $[\![Q]\!]_{s_{m+1},\xi} = \mathrm{T}$.
   Denn es ist $[\![P \wedge [Next]_v \wedge \langle A\rangle_v]\!]_{s_m,s_{m+1},\xi} = \mathrm{T}$, und mit der Prämisse
   $$P \wedge \langle Next \wedge A\rangle_v \Rightarrow Q'$$
   folgt die Behauptung.
7. Widerspruch, also muss ein $m \geq n$ mit $[\![Q]\!]_{s_m,\xi} = \mathrm{T}$ existieren. Q.E.D.

10.3.3 Lebendigkeit 2: fundierte Ordnungen

Die Regeln (WF1) und (SF1) bilden die Basis für Beweise von Lebendigkeitseigenschaften. Komplexere Aussagen folgen durch Induktion über fundierte Ordnungen (vgl. die Rolle der „Variante“ in der Iterationsregel aus Abschnitt 8.4).

Beispiel: Die Regel (WF1) erlaubt zu beweisen, dass das erste Element im Puffer irgendwann ausgegeben wird. Wie beweist man die analoge Aussage für beliebige Elemente im Puffer?

$$SIQSpec \Rightarrow ((k < \text{length}(q) \wedge q[k] = x) \rightsquigarrow o = x)$$

(mit rigiden Variablen k und x)

Informelle Argumentation: Jeder Ausgabeschritt *SIQDeq* bewirkt, dass das Element im Puffer weiter nach vorne rückt. Daher muss es sich irgendwann am Kopf des Puffers befinden, und mit dem nächsten *SIQDeq*-Schritt wird es ausgegeben.

Formalisierung: fundierte Relationen

Definition 57 *Eine Relation $\prec \subseteq D \times D$ auf einer Menge D heißt* fundiert, *wenn es keine unendliche Folge*

$$d_0 \succ d_1 \succ d_2 \succ \ldots$$

von Elementen $d_i \in D$ gibt.

Bemerkung: Fundierte Relationen sind irreflexiv und asymmetrisch:

- Gälte $d \prec d$ für ein $d \in D$, so gäbe es die unendliche absteigende Kette $d \succ d \succ d \succ \ldots$
- Gälte $d \prec e$ und $e \prec d$ für $d, e \in D$, so folgte $d \succ e \succ d \succ e \succ \ldots$, also wäre $\prec$ nicht fundiert.

In der Praxis benutzt man meist fundierte (Halb-)Ordnungen, also transitive fundierte Relationen $\prec$.

Beispiele: Fundierte Relationen sind $(\mathbb{N}, <)$, die lexikographische Ordnung auf Listen fester Länge und die strikte Teilmengenrelation auf endlichen Mengen.

Dagegen sind $(\mathbb{Z}, <)$ und die lexikographische Ordnung auf $\{a, b\}^*$ keine fundierten Ordnungen. Es gilt z.B.

$$b \succ ab \succ aab \succ aaab \succ \ldots$$

Die folgende Regel erlaubt den Beweis von „leadsto“-Eigenschaften unter Verwendung einer fundierten Relation.

$$\text{(WFO)} \quad \frac{\begin{array}{c}\prec \subseteq D \times D \text{ fundierte Relation} \\ F \wedge d \in D \Rightarrow (H(d) \rightsquigarrow (G \vee \exists e \in D : e \prec d \wedge H(e)))\end{array}}{F \Rightarrow ((\exists d \in D : H(d)) \rightsquigarrow G)}$$

Dabei seien $d, e \in X_r$ rigide Variablen, und d komme in der Formel G nicht frei vor.

Die temporale Prämisse von (WFO) erfordert selbst den Beweis einer „leadsto“-Eigenschaft. Dieser kann mit Hilfe von (WF1) bzw. (SF1) oder wiederum mit einer Anwendung von (WFO) erfolgen.

Beweis der Korrektheit von (WFO). Sei $\prec \subseteq D \times D$ eine fundierte Relation und gelte die temporale Prämisse von (WFO). Sei ferner σ eine Zustandsfolge und gelte $[\![F]\!]_{\sigma,\xi} = \mathrm{T}$.

Zu zeigen ist: $[\![(\exists\, d \in D : H(d)) \rightsquigarrow G]\!]_{\sigma,\xi} = \mathrm{T}$.

Sei $n \geq 0$ beliebig und gelte $[\![\exists\, d \in D : H(d)]\!]_{\sigma[n..],\xi} = \mathrm{T}$. Angenommen, für alle $m \geq n$ ist $[\![G]\!]_{\sigma[m..],\xi} = \mathrm{F}$.

Dann existiert ein $d_0 \in D$ mit $[\![H(d)]\!]_{\sigma[n..],\xi[d:=d_0]} = \mathrm{T}$, und mit der Prämisse von (WFO) und der Widerspruchsannahme folgt, dass für ein $n_1 \geq n$ gilt

$$[\![\exists\, e \in D : e \prec d \wedge H(e)]\!]_{\sigma[n_1..],\xi[d:=d_0]} = \mathrm{T}$$

Also existiert ein $d_1 \in D$ mit $d_1 \prec d_0$ und $[\![H(d)]\!]_{\sigma[n_1..],\xi[d:=d_1]} = \mathrm{T}$.

Induktiv folgt die Existenz einer Folge $d_0, d_1, \ldots \in D$ und einer Folge $n = n_0 \leq n_1 \leq \ldots$ mit

$$d_0 \succ d_1 \succ \ldots \qquad \text{und} \qquad [\![H(d)]\!]_{\sigma[n_i..],\xi[d:=d_i]} = \mathrm{T}$$

im Widerspruch zur Fundiertheit von $(D, \prec)$. Also muss ein $m \geq n$ existieren mit $[\![G]\!]_{\sigma[m..],\xi} = \mathrm{T}$. Q.E.D.

Beispiel 114: zur Anwendung von (WFO)

Für die Spezifikation *SIQSpec* aus Beispiel 109 zeigen wir

$$\mathit{SIQSpec} \Rightarrow \big((\exists k \in \mathbb{N} : 1 \leq k \leq \mathrm{length}(q) \wedge q[k] = x) \rightsquigarrow o = x\big)$$

Als fundierte Relation wählen wir $(\mathbb{N}, <)$. Gemäß Regel (WFO) ist zu beweisen:

$$\mathit{SIQSpec} \wedge k \in \mathbb{N} \Rightarrow \big(\underbrace{(1 \leq k \leq \mathrm{length}(q) \wedge q[k] = x)}_{H(k)} \rightsquigarrow (o = x \vee \exists m \in \mathbb{N} : m < k \wedge 1 \leq m \leq \mathrm{length}(q) \wedge q[m] = x)\big)$$

Dies folgt mit Hilfe von (WF1) aus den folgenden Aussagen:

1a. $H(k) \wedge \mathit{SIQEnq} \Rightarrow H(k)'$

1b. $H(k) \wedge \mathit{SIQDeq} \Rightarrow o' = x \vee \exists m \in \mathbb{N} : m < k \wedge H(m)'$

1c. $H(k) \wedge i' = i \wedge q' = q \wedge o' = o \Rightarrow H(k)'$

2. $H(k) \wedge \langle \mathit{SIQNext} \wedge \mathit{SIQDeq} \rangle_{i,q,o} \Rightarrow o' = x \vee \exists m \in \mathbb{N} : m < k \wedge H(m)'$

3. $H(k) \Rightarrow \mathrm{ENABLED} \langle \mathit{SIQDeq} \rangle_{i,q,o}$ ◀

10.3.4 Temporallogische Gesetze

Die bisher betrachteten Gesetze dienen hauptsächlich der Verifikation von Systemspezifikationen. Daneben gibt es in TLA auch rein logische Gesetze, die zur Umformulierung und Vereinfachung von Beweisverpflichtungen nützlich sein können. Diese bereiten die Anwendung von Verifikationsregeln wie (INV1), (WF1) und (WFO) vor.

Temporallogische Basisregeln betreffen die Operatoren $\Box$ und $\Diamond$ (*„simple temporal logic“*).

(STL1) Jede gültige prädikatenlogische Formel.

(STL2) $\Box F \Rightarrow F$

(STL3) $\Box F \Rightarrow \Box\Box F$

(STL4) $\Box(F \Rightarrow G) \Rightarrow (\Box F \Rightarrow \Box G)$

(STL5) $\Box(F \wedge G) \Leftrightarrow \Box F \wedge \Box G$

(STL6) $\Diamond\Box F \wedge \Diamond\Box G \Leftrightarrow \Diamond\Box(F \wedge G)$

(GEN) $\dfrac{F}{\Box F}$

(MP) $\dfrac{F \quad F \Rightarrow G}{G}$

Bemerkungen:

- Das Axiom (STL1) erlaubt es, jede Zustandsformel, die in der gegebenen Algebra bzw. Klasse von Algebren gilt, in Herleitungen zu benutzen. Diese Klasse kann z.B. durch eine prädikatenlogische Theorie beschrieben sein.
- Aus (STL1) und (MP) folgt, dass beliebige aussagenlogische Schlüsse in TLA-Herleitungen zulässig sind.
- (GEN) muss als Regel formuliert werden: die Formel $F \Rightarrow \Box F$ ist nicht gültig.

Aus diesen Basisregeln sind viele weitere Gesetze beweisbar.

Beispiel 115: Herleitung des temporallogischen Gesetzes $\Box F \wedge \Diamond G \Rightarrow \Diamond(F \wedge G)$

(1)	$F \wedge \neg(F \wedge G) \Rightarrow \neg G$	(STL1)
(2)	$\Box(F \wedge \neg(F \wedge G) \Rightarrow \neg G)$	(GEN)(1)
(3)	$\Box(F \wedge \neg(F \wedge G) \Rightarrow \neg G) \Rightarrow (\Box(F \wedge \neg(F \wedge G)) \Rightarrow \Box\neg G)$	(STL4)
(4)	$\Box(F \wedge \neg(F \wedge G)) \Rightarrow \Box\neg G$	(MP)(2)(3)
(5)	$\Box F \wedge \Box\neg(F \wedge G) \Rightarrow \Box(F \wedge \neg(F \wedge G))$	(STL5)
(6)	$\Box F \wedge \Box\neg(F \wedge G) \Rightarrow \Box\neg G$	(prop)(4)(5)
(7)	$\Box F \wedge \Diamond G \Rightarrow \Diamond(F \wedge G)$	(prop)(6) ◀

Tabelle 5 enthält eine Auswahl weiterer abgeleiteter temporallogischer Gesetze.

Bemerkung: In der Praxis müssen temporallogische Gesetze nicht aus den Regeln hergeleitet werden, da temporale Aussagenlogik entscheidbar ist. Ein solches Entscheidungsverfahren ist das Programm `ptl`, das unter `http://www.ics.ele.tue.nl/es/research/fv/` erhältlich ist. (`ptl` basiert nicht auf TLA-Syntax und kennt keine Formeln wie $\Box[A]_v$.)

Beispiele:

```
> ptl
PTL V4.2 Copyright (C) 1994-97 G. Janssen, Eindhoven University.

([](F & []!G -> <>G)) <-> [](F -> <>G);
True

[](F -> <>G) & [](F -> <>H) -> [](F -> <>(G & H));
False
```

(T1) $\neg \Box F \Leftrightarrow \Diamond \neg F$

(T2) $\neg \Diamond F \Leftrightarrow \Box \neg F$

(T3) $F \Rightarrow \Diamond F$

(T4) $\Diamond \Diamond F \Rightarrow \Diamond F$

(T5) $\dfrac{F \Rightarrow G}{\Box F \Rightarrow \Box G}$

(T6) $\dfrac{F \Rightarrow G}{\Diamond F \Rightarrow \Diamond G}$

(T7) $\Diamond(F \vee G) \Leftrightarrow \Diamond F \vee \Diamond G$

(T8) $\Box\Diamond F \vee \Box\Diamond G \Leftrightarrow \Box\Diamond(F \vee G)$

(T9) $\Box F \wedge \Diamond G \Rightarrow \Diamond(\Box F \wedge G)$

(T10) $\Box F \wedge \Diamond G \Rightarrow \Diamond(F \wedge G)$

(T11) $\Diamond\Box\Diamond F \Leftrightarrow \Box\Diamond F$

(T12) $\Box\Diamond\Box F \Leftrightarrow \Diamond\Box F$

(T13) $\Diamond\Box F \Rightarrow \Box\Diamond F$

(T14) $\Box\Diamond F \wedge \Diamond\Box G \Rightarrow \Box\Diamond(F \wedge G)$

(T15) $\Box \mathrm{WF}_v(A) \Leftrightarrow \mathrm{WF}_v(A)$

(T16) $\Box \mathrm{SF}_v(A) \Leftrightarrow \mathrm{SF}_v(A)$

(T17) $\dfrac{F \Rightarrow G}{F \leadsto G}$

(T18) $\dfrac{F \leadsto G \qquad G \leadsto H}{F \leadsto H}$

(T19) $((F \vee G) \leadsto H) \Leftrightarrow (F \leadsto H) \wedge (G \leadsto H)$

(T20) $((F \wedge \Box\neg G) \leadsto G) \Leftrightarrow (F \leadsto G)$

(WF1-Inv) $\dfrac{\begin{array}{c} I \wedge P \wedge [\mathit{Next}]_v \Rightarrow P' \vee Q' \\ I \wedge P \wedge \langle \mathit{Next} \wedge A \rangle_v \Rightarrow Q' \\ I \wedge P \Rightarrow \mathrm{ENABLED}\langle A \rangle_v \end{array}}{\Box I \wedge \Box[\mathit{Next}]_v \wedge \mathrm{WF}_v(A) \Rightarrow (P \leadsto Q)}$

Tabelle 5: Weitere temporallogische Gesetze.

```
<>[]F & []<>G -> []<>(F & G);
True
```

TLA-spezifische Gesetze kombinieren Aktionen und temporale Formeln

(TLA1) $\Box P \Leftrightarrow P \wedge \Box[P \Rightarrow P']_{\mathit{vars}(P)}$

(TLA2) $\dfrac{P \wedge [A]_v \Rightarrow [B]_w}{\Box P \wedge \Box[A]_v \Rightarrow \Box[B]_w}$

(INV2) $\Box P \wedge \Box[A]_v \Rightarrow \Box[A \wedge P \wedge P']_v$

(TLA3) $\Box[A]_v \wedge \Box[B]_w \Leftrightarrow \Box\big[[A]_v \wedge [B]_w\big]_{v,w}$

Dabei seien P Zustandsformel, A, B Aktionsformeln, v, w Tupel von Termen, $\mathit{vars}(P)$ ein Tupel aller freien Variablen in P. Diese Gesetze ermöglichen insbesondere „Simulationsbeweise“, z.B. gilt

$$\mathit{SIQSpec} \Rightarrow \mathit{LQSpec}$$

Beweis.

(1) $\mathit{SIQInit} \Rightarrow \mathit{LQInit}$ (prop)

(2) $\mathit{SIQEnq} \Rightarrow \mathit{LQEnq}$ (prop)

(3)	$SIQDeq \Rightarrow LQDeq$	(prop)
(4)	$[SIQNext]_{i,q,o} \Rightarrow [LQNext]_{q,o}$	(prop)(2)(3)
(5)	$\Box[SIQNext]_{i,q,o} \Rightarrow \Box[LQNext]_{q,o}$	(TLA2)(4)
(6)	$\langle SIQDeq\rangle_{i,q,o} \Leftrightarrow \langle LQDeq\rangle_{q,o}$	(data)
(7)	$\mathrm{WF}_{i,q,o}(SIQDeq) \Leftrightarrow \mathrm{WF}_{q,o}(LQDeq)$	(6)(TLA2)(STL)
(8)	$SIQSpec \Rightarrow LQSpec$	(prop)(1)(5)(7)

Q.E.D.

10.4 Zusammenfassung

- Die Logik TLA erweitert klassische Prädikatenlogik um gestrichene Variablen und temporale Operatoren. Man unterscheidet Zustands-, Aktions- und temporallogische Formeln, die über Zuständen, Zustandspaaren und unendlichen Zustandsfolgen interpretiert werden.
- Temporale Formeln enthalten Aktionsformeln nur in Teilformeln $\Box[A]_v$ bzw. $\Diamond\langle A\rangle_v$.
- Spezifikationen und Eigenschaften reaktiver Systeme werden durch TLA-Formeln ausgedrückt. Systemspezifikationen sind i.a. von der Form

  $$Init \wedge \Box[Next]_v \wedge L$$

  wobei L eine Konjunktion von Fairnessbedingungen ist. Flexible Variablen repräsentieren Zustandskomponenten des Systems.
- Ein System *Spec* erfüllt eine Eigenschaft *Prop*, wenn die Formel

  $$Spec \Rightarrow Prop$$

  gültig ist.
- Zum Beweis von Eigenschaften dienen logische Beweisregeln. Typische Verifikationsregeln reduzieren temporallogische Aussagen auf nicht-temporale Beweisverpflichtungen. Temporallogische Gesetze dienen zur Vereinfachung und Umformung von Aussagen.

11 Verfeinerung und Strukturierung

Bisher haben wir einfache („flache“) TLA-Spezifikationen und ihre Eigenschaften betrachtet. Im folgenden Kapitel studieren wir die Verfeinerung von TLA-Spezifikationen und strukturierte Spezifikationen. Ein Charakteristikum von TLA ist es, dass strukturelle Beziehungen wie Verfeinerung, Komposition und Kapselung durch logische Operatoren ausgedrückt werden.

Ziele

- Ablaufverfeinerung als Implikation von Spezifikationen beschreiben.
- Komposition (mit Synchronisierung über gemeinsame Komponenten) als Konjunktion darstellen.
- Kapselung durch Quantifizierung über flexible Variablen modellieren.
- Verfeinerungsabbildungen als technische Grundlage für Implementierungsbeweise kennenlernen.

11.1 Ablaufverfeinerung

Beispiel 116: Uhr mit Stunden-, Minuten- und Sekundenanzeige

Ähnlich wie in Beispiel 106 kann eine Uhr mit Stunden-, Minuten und Sekundenanzeige durch die folgende TLA-Spezifikation beschrieben werden:

$$\begin{aligned}
\mathit{IClk2} &\equiv hr \in \{0,\ldots,23\} \wedge min \in \{0,\ldots,59\} \wedge sec \in \{0,\ldots,59\} \\
\mathit{Sec2} &\equiv sec < 59 \wedge sec' = sec + 1 \wedge hr' = hr \wedge min' = min \\
\mathit{Min2} &\equiv sec = 59 \wedge min < 59 \wedge sec' = 0 \wedge min' = min + 1 \wedge hr' = hr \\
\mathit{Hr2} &\equiv sec = 59 \wedge min = 59 \wedge sec' = 0 \wedge min' = 0 \wedge hr' = (hr + 1) \bmod 24 \\
\mathit{Tick2} &\equiv \mathit{Sec2} \vee \mathit{Min2} \vee \mathit{Hr2} \\
\mathit{Clock2} &\equiv \mathit{IClk2} \wedge \Box[\mathit{Tick2}]_{hr,min,sec} \wedge \mathrm{WF}_{hr,min,sec}(\mathit{Tick2})
\end{aligned}$$

Jeder Ablauf von *Clock*2 erfüllt auch die Spezifikation *Clock* aus Beispiel 106:

- Die Anfangsbedingung *IClk*2 impliziert die Bedingung *IClk*.
- Jeder Übergang gemäß *Tick*2 lässt entweder (hr, min) unverändert oder erfüllt die Aktionsformel *Tick*.
- Die Fairnessbedingung $\mathrm{WF}_{hr,min,sec}(\mathit{Tick2})$ erzwingt, dass die Uhr unendlich oft tickt—aber dann finden auch unendlich viele $\langle \mathit{Tick} \rangle_{hr,min}$-Übergänge statt. Daher ist auch $\mathrm{WF}_{hr,min}(\mathit{Tick})$ erfüllt.

Also gilt die Implikation $\mathit{Clock2} \Rightarrow \mathit{Clock}$.

Hierbei ist wesentlich, dass TLA-Spezifikationen „Stotterschritte“ erlauben, welche die Systemvariablen nicht verändern. Bei Verfeinerungen dürfen daher neue Zustandskomponenten („Implementierungsdetails“) eingeführt werden und neue Zustandsübergänge definiert werden, die auf der Ebene der abstrakten Spezifikation nicht sichtbar sind. ◀

Definition 58 *Eine TLA-Spezifikation Impl heißt* (Ablauf-)Verfeinerung *einer Spezifikation Spec, wenn die Implikation*

$$\mathit{Impl} \Rightarrow \mathit{Spec}$$

gültig ist, d.h. wenn jede Zustandsfolge, die Impl erfüllt, auch Spec erfüllt.

Vergleich mit früher vorgestellten Verfeinerungsbegriffen:

- Der TLA-Verfeinerungsbegriff entspricht dem Begriff in algebraischen Spezifikationen insofern, als in beiden Fällen die Inklusion der Modellklassen (Algebren bzw. erfüllende Abläufe) gefordert wird. Allerdings wurde bei algebraischen Spezifikationen Gleichheit der Signaturen verlangt, während die Verfeinerung einer TLA-Spezifikation zusätzliche flexible Variablen („Implementierungsdetails“) enthalten darf.
- In Z waren zwei Bedingungen gefordert:

  $$(\mathrm{pre}\ A) \wedge C \Rightarrow A \qquad \text{und} \qquad \mathrm{pre}\ A \Rightarrow \mathrm{pre}\ C$$

  Die zweite Bedingung drückt aus, dass die implementierende Operation in jedem Zustand anwendbar ist, in dem die abstrakte Operation anwendbar ist. Eine analoge Bedingung fehlt in TLA:

die Implementierung darf zusätzliche Schritte benötigen, bevor ein Übergang, der einem Schritt der abstrakten Spezifikation entspricht, ausführbar wird. Die Uhrspezifikation aus Beispiel 116 benötigt etwa nach einem *Tick*-Übergang 59 *Tick*2-Schritte, bevor ein weiterer *Tick*-Schritt ausgeführt werden kann.

Andererseits führt das Fehlen einer solchen Bedingung in TLA dazu, dass gewisse Aktionen der Spezifikation in der Implementierung evtl. überhaupt nicht ausgeführt werden können—im Extremfall kann die Implementierung inkonsistent werden, denn **false** implementiert jede Spezifikation. Solche unerwünschten Implementierungen müssen in TLA durch Validierung ausgeschlossen werden.

Beweis von Verfeinerungen. Zu beweisen ist eine Formel der Art $Impl \Rightarrow Spec$ bzw. genauer

$$\underbrace{CInit \wedge \Box[CNext]_{cv} \wedge CLive}_{Impl} \quad \Rightarrow \quad \underbrace{AInit \wedge \Box[ANext]_{av} \wedge ALive}_{Spec}$$

Verfeinerungsbeweise erfolgen in 4 Schritten:

1. Beweis einer geeigneten Hilfsinvariante $Impl \Rightarrow \Box Inv$.
2. Initialisierungsbedingung $CInit \Rightarrow AInit$.
3. Simulationsbeweis $Inv \wedge [CNext]_{cv} \Rightarrow [ANext]_{av}$.
4. Lebendigkeitsbedingungen $\Box Inv \wedge \Box[CNext]_{cv} \wedge CLive \Rightarrow ALive$.

Nur der Beweis von Schritt 4 benötigt temporale Logik. Die folgende Verifikationsregel dient zum Beweis einer schwachen Fairnessbedingung unter Ausnutzung schwacher Fairness auf der Ebene der Implementierung:

$$\text{(WF2)} \quad \frac{\begin{array}{c}\langle N \wedge P \wedge P' \wedge A\rangle_v \Rightarrow \langle B\rangle_w \\ P \wedge \text{ENABLED}\langle B\rangle_w \Rightarrow \text{ENABLED}\langle A\rangle_v \\ \Box[N \wedge [\neg B]_w]_v \wedge \text{WF}_v(A) \wedge \Box F \wedge \Diamond\Box\,\text{ENABLED}\langle B\rangle_w \Rightarrow \Diamond\Box P\end{array}}{\Box[N]_v \wedge \text{WF}_v(A) \wedge \Box F \Rightarrow \text{WF}_w(B)}$$

Idee:

- Die Aktion A der Implementierung simuliert die Aktion B der Spezifikation, falls die Zusatzbedingung P gilt.

  In Beispiel 116: *Tick*2 simuliert *Tick*, falls $sec = 59$ gilt.
- A ist ausführbar, falls B ausführbar ist und P gilt.
- Wird B nie ausgeführt, wird P ab einem gewissen Zeitpunkt immer erfüllt sein.

Die analoge Regel zum Nachweis starker Fairness lautet

$$\text{(SF2)} \quad \frac{\begin{array}{c}\langle N \wedge P \wedge P' \wedge A\rangle_v \Rightarrow \langle B\rangle_w \\ P \wedge \text{ENABLED}\langle B\rangle_w \Rightarrow \text{ENABLED}\langle A\rangle_v \\ \Box[N \wedge [\neg B]_w]_v \wedge \text{SF}_v(A) \wedge \Box F \wedge \Box\Diamond\,\text{ENABLED}\langle B\rangle_w \Rightarrow \Diamond\Box P\end{array}}{\Box[N]_v \wedge \text{SF}_v(A) \wedge \Box F \Rightarrow \text{SF}_w(B)}$$

In komplizierteren Fällen (z.B. wenn die B simulierende Aktion nicht eindeutig bestimmt ist) müssen vor Anwendung von (WF2) oder (SF2) temporallogische Umformungen durchgeführt werden, oder die Aussage muss direkt aus der Definition von $\text{WF}_w(B)$ bzw. $\text{SF}_w(B)$ bewiesen werden.

Beweis der Korrektheit von (WF2). Seien die Prämissen von (WF2) gültig, sei $\sigma = s_0 s_1 \ldots$ Zustandsfolge und gelte

(1) $[\![\Box[N]_v \wedge \mathrm{WF}_v(A) \wedge \Box F]\!]_{\sigma,\xi} = \mathrm{T}$

Angenommen, es wäre $[\![\mathrm{WF}_w(B)]\!]_{\sigma,\xi} = \mathrm{F}$. Dann ist $\langle B\rangle_w$ fast immer ausführbar, wird aber nur endlich oft ausgeführt, also gibt es ein $n \in \mathbb{N}$, so dass für alle $k \geq n$ gilt:

(2) $[\![\mathrm{ENABLED}\langle B\rangle_w]\!]_{s_k,\xi} = \mathrm{T}$ aber (3) $[\![\langle B\rangle_w]\!]_{s_k,s_{k+1},\xi} = \mathrm{F}$

Aus (1) und (3) folgt

(4) $[\![\Box[N \wedge [\neg B]_w]_v \wedge \mathrm{WF}_v(A) \wedge \Box F]\!]_{\sigma[n..],\xi} = \mathrm{T}$

Mit (2) und der dritten Prämisse von (WF2) folgt: Es gibt ein $m \geq n$, so dass für alle $k \geq m$ gilt

(5) $[\![P]\!]_{s_k,\xi} = \mathrm{T}$

Aus (5) und (2) folgt mit der zweiten Prämisse, dass für alle $k \geq m$ gilt

(6) $[\![\mathrm{ENABLED}\langle A\rangle_v]\!]_{s_k,\xi} = \mathrm{T}$

Mit der Fairnessbedingung $\mathrm{WF}_v(A)$, vgl. (4), folgt: Es gibt ein $k \geq m$ mit

(7) $[\![\langle A\rangle_v]\!]_{s_k,s_{k+1},\xi} = \mathrm{T}$

Aus (4), (5), (7) und der ersten Prämisse ergibt sich

(8) $[\![\langle B\rangle_w]\!]_{s_k,s_{k+1},\xi} = \mathrm{T}$

Widerspruch zu (3). Q.E.D.

Beispiel 117: Beweis von $Clock2 \Rightarrow Clock$

1. Invariante: Wähle $Inv \equiv IClk2$.

Es gilt $Clock2 \Rightarrow \Box Inv$: Beweis wie üblich mit (INV1).

2. Initialisierung: Offensichtlich gilt $IClk2 \Rightarrow IClk$.

3. Simulation: Zu zeigen ist $Inv \wedge [Tick2]_{hr,min,sec} \Rightarrow [Tick]_{hr,min}$. Dazu zeigen wir:

1. $Sec2 \Rightarrow hr' = hr \wedge min' = min$
2. $Min2 \Rightarrow Min$
3. $Hr2 \Rightarrow Hr$
4. $hr' = hr \wedge min' = min \wedge sec' = sec \Rightarrow hr' = hr \wedge min' = min$

Mit (TLA2) folgt

$$\Box Inv \wedge \Box[Tick2]_{hr,min,sec} \Rightarrow \Box[Tick]_{hr,min}$$

4. Fairness: Zu zeigen ist

$$\Box Inv \wedge \Box[Tick2]_{hr,min,sec} \wedge \mathrm{WF}_{hr,min,sec}(Tick2) \Rightarrow \mathrm{WF}_{hr,min}(Tick)$$

Dazu zeigen wir:

1. $\langle Tick2 \wedge sec = 59 \wedge sec' = 59\rangle_{hr,min,sec} \Rightarrow \langle Tick\rangle_{hr,min}$.
   Diese Aussage folgt mit Datenaxiomen aus Definition von *Tick*2.
2. $sec = 59 \wedge \text{ENABLED}\langle Tick\rangle_{hr,min} \Rightarrow \text{ENABLED}\langle Tick2\rangle_{hr,min,sec}$.
   Trivial, denn $\text{ENABLED}\langle Tick2\rangle_{hr,min,sec}$ gilt immer.
3. $\Box[Tick2 \wedge [\neg\, Tick]_{hr,min}]_{hr,min,sec} \wedge \text{WF}_{hr,min,sec}(Tick2) \wedge \Box Inv \wedge \Diamond\Box\,\text{ENABLED}\langle Tick\rangle_{hr,min}$
   $\Rightarrow \Diamond\Box(sec = 59)$
   - Beweis von „$\ldots \Rightarrow \Diamond(sec = 59)$“: Regel (WFO) mit „Variante“ $59 - sec$.
   - Beweis von „$\ldots \wedge sec = 59 \Rightarrow \Box(sec = 59)$“: Standard-Invariantenbeweis, da *Tick*-Übergänge ausgeschlossen sind.
   - Die Gesamtaussage folgt mit „simple temporal logic“.

Die Anwendung von (WF2) ergibt

$$\Box[Tick2]_{hr,min,sec} \wedge \text{WF}_{hr,min,sec}(Tick2) \wedge \Box Inv \Rightarrow \text{WF}_{hr,min}(Tick)$$

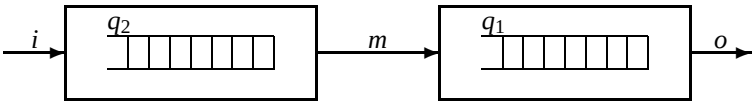
und damit ist die Verfeinerung von *Clock* durch *Clock*2 bewiesen. ◄

11.2 Komposition von Spezifikationen

Spezifikationen größerer Systeme können aus Spezifikationen ihrer Komponenten zusammengesetzt werden. Bei reaktiven Systemen bedeutet dies eine parallele Komposition der Komponenten. Dabei entsprechen gemeinsame flexible Variablen in den Komponentenspezifikationen Schnittstellen zwischen den Komponenten; ggf. müssen Variablen vor der Komposition geeignet umbenannt werden.

Wird das Verhalten der Komponenten A und B durch Formeln *ASpec* und *BSpec* beschrieben, so beschreibt *ASpec* $\wedge$ *BSpec* das Verhalten des Gesamtsystems. Zumindest bei zeit-asynchronen Systemen (d.h. kein gemeinsamer Takt) ist hier wieder essenziell, dass TLA „Stotterschritte“ erlaubt.

Beispiel 118: Komposition zweier FIFO-Puffer



Der linke und rechte Puffer können z.B. durch

$$SIQSpec[m/o, q_2/q] \qquad \text{und} \qquad SIQSpec[m/i, q_1/q]$$

beschrieben werden. Das System mit zwei hintereinandergeschalteten Puffern erfüllt gerade die Formel

$$SIQSpec[m/o, q_2/q] \wedge SIQSpec[m/i, q_1/q]$$

Die beiden Puffer werden über die gemeinsame Schnittstelle *m* synchronisiert.

Frage: Intuitiv stellt das Gesamtsystem wieder einen Puffer dar. Lässt sich das in TLA ausdrücken?

Anwort: Zu erwarten ist die Gültigkeit der Formel

$$SIQSpec[m/o, q_2/q] \wedge SIQSpec[m/i, q_1/q] \Rightarrow SIQSpec[\text{append}(q_1, q_2)/q]$$

Allerdings setzt *SIQSpec* eine Modellierung mit Interleaving voraus, d.h. es gilt

$$SIQSpec \Rightarrow \Box[i' = i \vee o' = o]_{i,o}$$

während die Komposition gleichzeitige Änderungen von Ein- und Ausgabe erlaubt.

Die Interleaving-Annahme wurde zur Vereinfachung der Modellierung getroffen. Wird sie auch für die Komposition angenommen, so ist die Implementierung eines Puffers durch zwei hintereinandergeschaltete Puffer beweisbar:

$$SIQSpec[m/o, q_2/q] \wedge SIQSpec[m/i, q_1/q] \wedge \Box[i' = i \vee o' = o]_{i,o} \\ \Rightarrow SIQSpec[\text{append}(q_1, q_2)/q]$$

Der Beweis erfolgt mit den bekannten Regeln und verbleibt zur Übung.

Die analoge Aussage für die Modellierung ohne Interleaving-Annahme (Formel *SNQSpec*) benötigt keine Zusatzannahme. ◀

11.3 Kapselung

Die Kapselung von Zustandskomponenten dient wie bei algebraischen Spezifikationen und bei Z dazu, interne Hilfsstrukturen nicht nach außen sichtbar zu machen. Logisch kann dies durch Existenzquantifizierung ausgedrückt werden; bereits in Z galt z.B.

$$[y : \mathbb{N};\ z : 1 \,..\, 10 \mid y = z * z] \setminus (z) \quad \equiv \quad [y : \mathbb{N} \mid \exists z : 1 \,..\, 10 \bullet y = z * z]$$

Analog wird Kapselung in TLA durch Existenzquantifizierung über flexible Variablen ausgedrückt.

Kapselung ermöglicht, dass Spezifikationen nur das an der Schnittstelle sichtbare Verhalten beschreiben. Daher verbleibt Freiheit für die spätere Implementierung der Komponente.

Beispiel 119: Spezifikation einer Speicherschnittstelle

op : gibt gewünschte Operation an und wird nach Ausführung der Operation zurückgesetzt.

adr : Adresse der Speicherzelle, die gelesen oder geschrieben werden soll.

val : enthält den zu schreibenden Wert bzw. das Ergebnis der Lese-Operation.

$$\begin{aligned}
MInit &\equiv op = \text{“none”} \wedge m \in (Adr \to Val) \\
MReqRd &\equiv op = \text{“none”} \wedge op' = \text{“read”} \wedge adr' \in Adr \wedge m' = m \\
MReqWr &\equiv op = \text{“none”} \wedge op' = \text{“write”} \wedge adr' \in Adr \wedge val' \in Val \wedge m' = m \\
MDoRd &\equiv op = \text{“read”} \wedge op' = \text{“none”} \wedge val' = m(adr) \wedge m' = m \\
MDoWr &\equiv op = \text{“write”} \wedge op' = \text{“none”} \wedge m' = m \oplus \{adr \mapsto val\} \\
MNext &\equiv MReqRd \vee MReqWr \vee MDoRd \vee MDoWr \\
v &\equiv (op, adr, val, m) \\
MISpec &\equiv MInit \wedge \Box[MNext]_v \wedge \mathrm{WF}_v(MDoRd) \wedge \mathrm{WF}_v(MDoWr) \\
MSpec &\equiv \exists m : MISpec
\end{aligned}$$

Die Formel *MISpec* enthält neben den Schnittstellenvariablen *op*, *adr* und *val* auch die interne Variable *m*, die den aktuellen Speicherzustand repräsentiert. Die Formel *MSpec* entsteht durch Quantifizierung über *m* und beschreibt daher das Verhalten der Schnittstelle, unabhängig von der konkreten Realisierung des Speichers—wie wir in Beispiel 120 auf Seite 167 sehen werden. ◀

11.3.1 Erweiterung von TLA um Quantoren

Wir erweitern Definition 56 wie folgt:

- Ist F temporale Formel und $x \in X_r$ rigide Variable, so ist auch $\exists x : F$ eine temporale Formel.
- Ist F temporale Formel und $x \in X_f$ flexible Variable, so ist auch $\mathbf{\exists} x : F$ eine temporale Formel.

Allquantifizierung kann als Abkürzung definiert werden:

$$\forall x : F \equiv \neg\, \exists x : \neg\, F \qquad \mathbf{\forall} x : F \equiv \neg\, \mathbf{\exists} x : \neg\, F$$

Die Semantik von Quantoren über rigide Variablen wird wie üblich definiert:

$$[\![\exists x : F]\!]_{\sigma,\xi} = \mathrm{T} \quad \text{gdw.} \quad [\![F]\!]_{\sigma,\eta} = \mathrm{T} \text{ für eine Belegung } \eta \text{ mit } \eta(y) = \xi(y) \text{ für alle } y \in X_r \setminus \{x\}$$

Mit dieser Definition gelten die üblichen Quantorenregeln aus der klassischen Prädikatenlogik:

$$(\exists\text{-I}) \quad F(t) \Rightarrow \exists x : F(x) \qquad\qquad (\exists\text{-E}) \quad \frac{F \Rightarrow G}{(\exists x : F) \Rightarrow G} \ (x \text{ nicht frei in } G)$$

Bei ($\exists$-I) darf der Term t nur rigide Variablen enthalten.

Beispiele: Es gilt z.B. $\Diamond(v = 2) \Rightarrow \exists x : \Diamond(v = x * x + 1)$.

Dagegen ist es nicht korrekt zu folgern $\Box(v = v) \Rightarrow \exists x : \Box(v = x)$. Denn die rechte Seite besagt, dass ein fester Wert x existiert, so dass immer $v = x$ gilt.

Beweis der Korrektheit von ($\exists$-E). Sei $F \Rightarrow G$ gültig, d.h $[\![F \Rightarrow G]\!]_{\sigma,\xi} = \mathrm{T}$ für alle σ, ξ.

Zu zeigen ist, dass auch $[\![(\exists x : F) \Rightarrow G]\!]_{\sigma,\xi} = \mathrm{T}$ für alle σ und ξ gilt. Seien also σ, ξ beliebig und gelte $[\![\exists x : F]\!]_{\sigma,\xi} = \mathrm{T}$.

Also existiert eine Belegung η mit $\eta(y) = \xi(y)$ für alle $y \in X_r \setminus \{x\}$, so dass $[\![F]\!]_{\sigma,\eta} = \mathrm{T}$ ist. Wegen der Gültigkeit von $F \Rightarrow G$ folgt daher auch $[\![G]\!]_{\sigma,\eta} = \mathrm{T}$, und weil x in G nicht frei vorkommt, folgt auch $[\![G]\!]_{\sigma,\xi} = \mathrm{T}$. Q.E.D.

Ferner gilt das Axiom

$$(\exists\text{-}\Diamond) \quad \Diamond(\exists x : F) \Rightarrow (\exists x : \Diamond F)$$

Die umgekehrte Implikation ist beweisbar:

(1)	$F \Rightarrow \exists x : F$	($\exists$-I)
(2)	$\Diamond F \Rightarrow \Diamond(\exists x : F)$	(T6)(1)
(3)	$(\exists x : \Diamond F) \Rightarrow \Diamond(\exists x : F)$	($\exists$-E)(2)

Für den Allquantor sind entsprechende Regeln beweisbar:

$$(\forall\text{-I}) \quad \frac{F \Rightarrow G}{F \Rightarrow \forall x : G} \ (x \text{ nicht frei in } F)$$

$$(\forall\text{-E}) \quad (\forall x : F(x)) \Rightarrow F(t) \ \ (t \text{ enthält nur rigide Variablen})$$

$$(\forall\text{-}\Box) \quad \Box(\forall x : F) \Leftrightarrow (\forall x : \Box F)$$

11.3.2 Stotterinvarianz von TLA-Formeln

Die Semantik von Quantoren über flexible Variablen ist etwas komplizierter, was an der gewünschten Stotterinvarianz von (auch quantifizierten) TLA-Formeln liegt. Wir holen zunächst eine formale Definition dieses Begriffs nach.

Definition 59

1. *Sei* $\sigma = s_0 s_1 s_2 \ldots$ *eine Zustandsfolge. Eine Zustandsfolge* $\tau = s_{i_0} s_{i_1} s_{i_2} \ldots$ *(mit* $i_0 < i_1 < i_2 < \ldots$*) heißt* stotterfreie Variante *von* σ*, geschrieben* $\tau = \natural(\sigma)$*, wenn gilt:*
   - $i_0 = 0$,
   - $s_j = s_{i_k}$ *für alle* j *mit* $i_k \leq j < i_{k+1}$ *und*
   - $s_{i_k} = s_{i_{k+1}}$ *genau dann, wenn* $s_j = s_{i_k}$ *für alle* $j \geq i_k$ *gilt.*
2. *Zwei Zustandsfolgen* σ *und* τ *heißen* stotteräquivalent*, geschrieben* $\sigma \sim \tau$*, wenn* $\natural(\sigma) = \natural(\tau)$ *gilt.*

Bemerkung:

- Intuitiv ist $\natural(\sigma)$ diejenige Zustandsfolge, die sich aus σ dadurch ergibt, dass maximale endliche Teilfolgen gleicher Zustände s durch ein einziges Vorkommen von s ersetzt werden.
- Zu gegebenem σ ist die stotterfreie Variante $\natural(\sigma)$ eindeutig bestimmt (aber nicht die Folge $i_0, i_1, \ldots$, falls σ mit unendlicher Zustandswiederholung endet). Daher ist die Schreibweise „$\tau = \natural(\sigma)$“ gerechtfertigt. Es gilt

$$\natural(s_0 s_1 s_2 \ldots) = \left\{ \begin{array}{ll} s_0 s_1 s_2 \ldots & \text{falls } s_0 = s_i \text{ für alle } i \geq 0 \text{ gilt} \\ \langle s_0 \rangle \circ \natural(s_k s_{k+1} \ldots) & \text{falls } k \text{ der kleinste Index mit } s_k \neq s_0 \text{ ist} \end{array} \right\}$$

- Gilt $\sigma \sim \tau$ (für $\sigma = s_0 s_1 \ldots$ und $\tau = t_0 t_1 \ldots$), so folgt:

$$s_0 = t_0 \quad \text{und} \quad \text{für alle } i \in \mathbb{N} \text{ gibt es } j \in \mathbb{N} \text{ mit } \sigma[i..] \sim \tau[j..]$$

Der folgende Satz besagt, dass TLA-Formeln (zunächst ohne Quantoren über flexible Variablen) zwischen stotteräquivalenten Zustandsfolgen nicht unterscheiden können, also „stotterinvariant“ sind. Nach Lamport ist die Stotterinvarianz eine wünschenswerte Eigenschaft für temporale Logiken: sie ermöglicht, Verfeinerung durch Implikation und Komposition durch Konjunktion auszudrücken.

Satz 60 *Sei* F *temporale Formel ohne Teilformeln* $\exists x : G$ *und seien* $\sigma \sim \tau$ *stotteräquivalente Zustandsfolgen. Dann ist*

$$[\![F]\!]_{\sigma,\xi} = \mathrm{T} \quad \textit{gdw.} \quad [\![F]\!]_{\tau,\xi} = \mathrm{T}.$$

Beweis. Seien $\sigma = s_0 s_1 \ldots$ und $\tau = t_0 t_1 \ldots$ Zustandsfolgen mit $\sigma \sim \tau$. Der Beweis der Aussage erfolgt durch Induktion über den Aufbau der Formel F.

- Ist F Zustandsformel, so gilt die Aussage wegen $s_0 = t_0$ (vgl. Bemerkung nach Definition 59).
- Für aussagenlogische Kombinationen und Quantoren über rigide Variablen folgt die Aussage unmittelbar aus der Induktionsvoraussetzung.

- $[\![\Box G]\!]_{\sigma,\xi} = \mathrm{T}$
  $\Rightarrow$ $[\![G]\!]_{\sigma[i..],\xi} = \mathrm{T}$ für alle $i \in \mathbb{N}$ [Def. $[\![\Box G]\!]_{\sigma,\xi}$]
  $\Rightarrow$ $[\![G]\!]_{\tau[j..],\xi} = \mathrm{T}$ für alle $j \in \mathbb{N}$ [mit Ind.vor., denn f.a. $j \in \mathbb{N}$ ex. $i \in \mathbb{N}$ mit $\tau[j..] \sim \sigma[i..]$]
  $\Rightarrow$ $[\![\Box G]\!]_{\tau,\xi} = \mathrm{T}$ [Def. $[\![\Box G]\!]_{\tau,\xi}$]
  Die umgekehrte Implikation wird genauso bewiesen.

- Sei $F \equiv \Box[A]_v$. Es genügt zu zeigen: $[\![F]\!]_{\sigma,\xi} = [\![F]\!]_{\natural(\sigma),\xi}$; sei also $\natural(\sigma) = s_{i_0}s_{i_1}\ldots$
  Gilt $[\![F]\!]_{\sigma,\xi} = \mathrm{T}$, so folgt $[\![F]\!]_{\natural(\sigma),\xi} = \mathrm{T}$, da jeder Übergang $(s_{i_k}, s_{i_{k+1}})$ in $\natural(\sigma)$ auch als Übergang $(s_{i_{k+1}-1}, s_{i_{k+1}})$ in σ vorkommt.
  Ist umgekehrt $[\![F]\!]_{\natural(\sigma),\xi} = \mathrm{T}$, so folgt auch $[\![F]\!]_{\sigma,\xi} = \mathrm{T}$: Ist $s_j = s_{j+1}$, so folgt offenbar $[\![v]\!]_{s_j,\xi} = [\![v]\!]_{s_{j+1},\xi}$ und daher auch $[\![[A]_v]\!]_{s_j,s_{j+1},\xi} = \mathrm{T}$. Ansonsten ist $s_{j+1} = s_{i_{k+1}}$ für ein $k \in \mathbb{N}$, und es folgt $s_j = s_{i_k}$. Damit ist $[\![[A]_v]\!]_{s_j,s_{j+1},\xi} = [\![[A]_v]\!]_{s_{i_k},s_{i_{k+1}},\xi} = \mathrm{T}$. Q.E.D.

11.3.3 Semantik von Quantoren über flexible Variablen

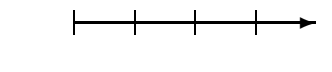
Es wäre naheliegend, die Semantik von $\exists x : F$ folgendermaßen zu definieren:

$[\![\exists x : F]\!]_{\sigma,\xi} = \mathrm{T}$ gdw.
$[\![F]\!]_{\tau,\xi} = \mathrm{T}$ für eine Zustandsfolge τ mit $\tau =_x \sigma$ (d.h. $\tau_i(y) = \sigma_i(y)$ für alle $y \in X_f \setminus \{x\}$, $i \in \mathbb{N}$)

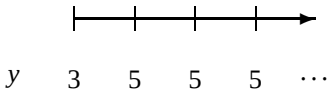
Diese Definition würde jedoch die Stotterinvarianz verletzen: die Formel

$$F \equiv x = 0 \wedge \Box[x = 1]_y$$

verlangt, dass sich x ändert, bevor sich y ändert. Daher folgt:



erfüllt die Formel $\exists x : F$.



erfüllt die Formel $\exists x : F$ nicht.

Die beiden Zustandsfolgen sind aber stotteräquivalent.

Die Semantik von $\exists x : F$ wird daher explizit so definiert, dass sie unter Stotterinvarianz abgeschlossen ist, indem neben der Änderung der Werte von x zusätzliche Stotterschritte eingefügt werden dürfen:

$[\![\exists x : F]\!]_{\sigma,\xi} = \mathrm{T}$ gdw.
es gibt Zustandsfolgen ρ, τ, so dass gilt: $\sigma \sim \rho$, $\rho =_x \tau$ und $[\![F]\!]_{\tau,\xi} = \mathrm{T}$

- Beide oben gezeigte Zustandsformeln erfüllen die Formel $\exists x : x = 0 \wedge \Box[x = 1]_y$.
  Tatsächlich ist diese Formel allgemeingültig.

- Im Uhr-Beispiel (Bsp. 116) gilt $Clock \Rightarrow \exists sec : Clock2$.
  Intuitiv: Jede Uhr mit Stunden- und Minutenanzeige könnte so implementiert sein, dass intern Sekunden gezählt werden.

Obwohl die Semantik komplizierter ist, gelten die üblichen Beweisregeln:

$$(\exists\text{-I}) \quad F(t) \Rightarrow \exists x : F(x)$$

$$(\exists\text{-E}) \quad \frac{F \Rightarrow G}{(\exists x : F) \Rightarrow G} \ (x \text{ nicht frei in } G)$$

$$(\exists\text{-}\Diamond) \quad (\exists x : \Diamond F) \Leftrightarrow \Diamond(\exists x : F)$$

Der Term t in ($\exists$-I) darf rigide und flexible Variablen enthalten.

Beispiel 120: Implementierung eines Speichers (vgl. Bsp. 119)

Die Implementierung enthält einen Hauptspeicher *mm* und einen Cache *cc*. Interne Operationen transportieren Daten zwischen Hauptspeicher und Cache. Die Schnittstelle nach außen bleibt unverändert. (Wir benutzen hier wieder Z-artige Schreibweisen für Daten-Operationen.)

$$\begin{aligned}
CInit &\equiv op = \text{“none”} \wedge mm \in (Adr \to Val) \wedge cc = (\lambda a : Adr \bullet \bot) \\
CReqRd &\equiv op = \text{“none”} \wedge op' = \text{“read”} \wedge adr' \in Adr \wedge mm' = mm \wedge cc' = cc \\
CReqWr &\equiv op = \text{“none”} \wedge op' = \text{“write”} \wedge adr' \in Adr \wedge val' \in Val \wedge mm' = mm \wedge cc' = cc \\
CDoRd &\equiv op = \text{“read”} \wedge cc(adr) \neq \bot \wedge op' = \text{“none”} \wedge val' = cc(adr) \wedge mm' = mm \wedge cc' = cc \\
CDoWr &\equiv op = \text{“write”} \wedge op' = \text{“none”} \wedge cc' = cc \oplus \{adr \mapsto val\} \wedge mm' = mm \\
CLoad(a) &\equiv cc(a) = \bot \wedge cc' = cc \oplus \{a \mapsto mm(a)\} \wedge \text{UNCHANGED}(op, adr, val, mm) \\
CFlush(a) &\equiv cc(a) \neq \bot \wedge (op = \text{“read”} \Rightarrow a \neq adr) \wedge mm' = mm \oplus \{a \mapsto cc(a)\} \\
&\quad \wedge\ cc' = cc \oplus \{a \mapsto \bot\} \wedge \text{UNCHANGED}(op, adr, val) \\
CNext &\equiv CReqRd \vee CReqWr \vee CDoRd \vee CDoWr \vee \exists a \in Adr : CLoad(a) \vee CFlush(a) \\
w &\equiv (op, adr, val, mm, cc) \\
CISpec &\equiv CInit \wedge \Box[CNext]_w \wedge \text{WF}_w(CDoRd) \wedge \text{WF}_w(CDoWr) \\
&\quad \wedge\ \text{WF}_w(CLoad(adr) \wedge op = \text{“read”}) \\
CSpec &\equiv \exists mm, cc : CISpec
\end{aligned}$$

◀

11.3.4 Verfeinerungsabbildungen (refinement mappings)

Um die Implementierung des Speichers mit einem Cache aus Beispiel 120 als korrekte Verfeinerung der urpsrünglichen Spezifikation nachzuweisen, muss

$$(\exists mm, cc : CISpec) \Rightarrow (\exists m : MISpec)$$

bewiesen werden. Gemäß Regel ($\exists$-E) kann dies auf den Beweis von

$$CISpec \Rightarrow \exists m : MISpec$$

zurückgeführt werden, da *mm* und *cc* in $(\exists m : MISpec)$ nicht frei vorkommen.

Um die Regel ($\exists$-I) anwenden zu können, ist

$$CISpec \Rightarrow MISpec[t/m]$$

für einen geeigneten Term t zu zeigen.

Ein solcher Term t „berechnet“ passende Werte der internen Variablen *m* der Spezifikation aus den Variablen der Implementierung. Er wird üblicherweise als *Verfeinerungsabbildung* (refinement mapping) bezeichnet.

Beispiel 121: Verfeinerungsabbildung für das Speicherbeispiel

Wir setzen

$$t \;=_{\text{def}}\; \lambda a : Adr \bullet \textbf{if } cc(a) = \bot \textbf{ then } mm(a) \textbf{ else } cc(a)$$

Im folgenden sei $\bot \notin Val$ vorausgesetzt. Der Beweis von $CISpec \Rightarrow MISpec[t/m]$ erfolgt mit den üblichen Verifikationsregeln.

1. $CISpec \Rightarrow \Box(op = \text{“write”} \Rightarrow val \in Val)$: Anwendung von (INV1).
2. $CInit \Rightarrow MInit[t/m]$: offensichtlich.
3. $Inv \wedge [CNext]_w \Rightarrow ([MNext]_v)[t/m]$
   - $CReqRd \Rightarrow MReqRd[t/m]$, $CReqWr \Rightarrow MReqWr[t/m]$: unmittelbar aus Definition der Aktionen und von t
   - $CDoRd \Rightarrow MDoRd[t/m]$: die Definition von t impliziert insbesondere $val' = t(adr)$ und $t' = t$.
   - $(op = \text{“write”} \Rightarrow val \in Val) \wedge CDoWr \Rightarrow MDoWr[t/m]$: aus der Invariante und der Annahme $\bot \notin Val$ folgt $t' = t \oplus \{adr \mapsto val\}$.
   - $a \in Adr \wedge CLoad(a) \Rightarrow \text{UNCHANGED}(op, adr, val, t)$: unmittelbar aus der Definition von $CLoad(a)$ und von t.
   - $a \in Adr \wedge CFlush(a) \Rightarrow \text{UNCHANGED}(op, adr, val, t)$: unmittelbar aus der Definition von $CFlush(a)$ und von t.
   - $\text{UNCHANGED}\, w \Rightarrow \text{UNCHANGED}(op, adr, val, t)$: trivial.
4. $CISpec \Rightarrow (\text{WF}_v(MDoRd) \wedge \text{WF}_v(MDoWr))[t/m]$: Anwendung von (WF2), dabei garantiert insbesondere die Fairnessbedingung an die Aktion $CLoad(adr) \wedge op = \text{“read”}$, dass $cc(adr) \neq \bot$ irgendwann wahr wird. Die Definition von $CFlush(a)$ garantiert, dass dies bis zur Ausführung von $CDoRd$ erhalten bleibt. ◄

Unvollständigkeit von refinement mappings

In manchen Situationen existiert keine geeignete Verfeinerungsabbildung, weil die Implementierung nicht genügend Details enthält.

Beispiele:

- Die Formel $Clock \Rightarrow \exists sec : Clock2$ ist gültig, aber nicht mit ($\exists$-I) beweisbar: der Wert der Sekundenanzeige kann nicht aus der Stunden- und Minutenanzeige berechnet werden.
- Ähnlich gilt $MISpec \Rightarrow \exists mm, cc : CISpec$ (d.h. jeder Speicher könnte mit einem Cache implementiert worden sein), obwohl dies mit ($\exists$-I) nicht bewiesen werden kann.

Für solche Fälle existieren zusätzliche Beweisregeln, z.B.

$$(\exists\text{-H}) \quad \frac{I \Rightarrow \exists h : HI \qquad N \Rightarrow \forall h \exists h' : HN}{I \wedge \Box[N]_v \Rightarrow \exists h : HI \wedge \Box[N \wedge HN]_{v,h}}$$

erlaubt die „induktive Definition“ einer Hilfsvariablen h.

Der folgende Artikel enthält weitere Regeln und gibt Bedingungen für deren Vollständigkeit an:

M. Abadi, L. Lamport: *The Existence of Refinement Mappings*.
Theoretical Computer Science 81(2), S. 253–284 (1991).

11.4 Zusammenfassung

- Verfeinerung und Strukturierung werden in TLA durch logische Operatoren ausgedrückt.
- Dabei entspricht die Verfeinerung der Implikation, die parallele Komposition der Konjunktion und die Kapselung der Existenzquantifizierung über flexible Variable. Umbenennung kann durch Ersetzung flexibler Variablen ausgedrückt werden.
- Eine wesentliche Voraussetzung dafür ist die Stotterinvarianz von TLA-Formeln. Diese wird syntaktisch dadurch garantiert, dass Aktionen nur in der Form $\Box[A]_v$ (bzw. $\Diamond\langle A\rangle_v$) in temporalen Formeln vorkommen dürfen.
- Verfeinerungsbeweise für Spezifikationen mit internen Variablen

  $$Impl \Rightarrow \exists x : Spec$$

  basieren auf Verfeinerungsabbildungen, d.h. man beweist

  $$Impl \Rightarrow Spec[t/x]$$

  für einen geeigneten Term t, der angibt, wie die gekapselten Variablen x aus den Variablen der Implementierung berechnet werden können.

12 Analyse von Transitionssystemen durch Modelchecking

Die bisher vorgestellten Regeln zur Verifikation reaktiver Systeme in TLA können durch Theorembeweiser maschinell unterstützt werden. Der hohe Aufwand deduktiver Verifikation und die benötigten Spezialkenntnisse rechtfertigen derzeit allerdings einen praktischen Einsatz nur in Ausnahmefällen. Breite Akzeptanz finden dagegen sogenannte Modelchecking-Verfahren zur automatischen Analyse zustandsendlicher Modelle, die als Debugging-Techniken verwendet werden können.

Ziele

- Algorithmen zur Überprüfung von Invarianten und temporaler Formeln verstehen.
- Grundbegriffe der Theorie endlicher Automaten auf unendlichen Wörtern kennenlernen.
- „On-the-fly“-Modelchecking
- Abstraktionsverfahren zur Analyse (evtl. unendlich) großer Zustandsräume.

12.1 Überprüfung von Invarianten

Ein Transitionssystem Γ erfüllt die Invariante $\Box P$, wenn für jeden Ablauf $\sigma = s_0 \xrightarrow{A_0} s_1 \xrightarrow{A_1} s_2 \ldots$ von Γ alle Zustände s_i die Zustandsformel P erfüllen. Anders ausgedrückt: P ist in jedem erreichbaren Zustand von Γ erfüllt. (Ein Zustand s heißt erreichbar, wenn es einen Ablauf $s_0 s_1 \ldots$ von Γ gibt mit $s = s_i$ für ein $i \in \mathbb{N}$.)

Semantisch können Invarianten also durch Mengen von Zuständen beschrieben werden. Das Transitionssystem $\Gamma = (Z, I, \mathcal{A}, \delta)$ erfüllt die Invariante $Inv \subseteq Z$, wenn jeder erreichbare Zustand von Γ in Inv liegt.

```
Set seen = new Set();  Stack trace = new Stack();

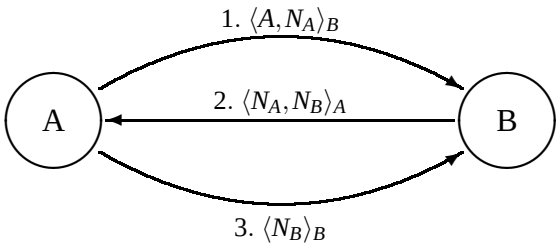
void dfs(State s) {
  if (! seen.contains(s)) {
     seen.insert(s);  trace.push(s);
     if (! s.satisfies(inv)) {
        throw new InvariantViolatedException();
     }
     Enumeration succs = getSuccessorStates(s);
     while (succs.hasMoreElements()) {
        dfs(succs.getNextElement());
     }
     trace.pop();
}  }

Enumeration inits = getInitialStates();
while (inits.hasMoreElements()) {
  try {  dfs(inits.getNextElement());  }
  catch(InvariantViolatedException e) {
     // Gegenbeispiel aus Stack-Inhalt ablesen
}  }
```

Abbildung 16: Basis-Algorithmus zur Überprüfung von Invarianten.

Ist die Zustandsmenge Z endlich (und δ effektiv gegeben), so können Invarianten durch Aufzählung der Menge der erreichbaren Zustände überprüft werden. Der in Abbildung 16 gezeigte Algorithmus erzeugt systematisch durch Tiefensuche alle erreichbaren Zustände und bricht mit einer Fehlermeldung ab, falls ein Zustand die Invariante verletzt. In diesem Fall enthält der Stack eine Folge der Zustände, die zur Verletzung der Invariante führt; diese kann dem Benutzer als Gegenbeispiel ausgegeben werden und damit Hinweise zum Debugging geben.

Beispiel 122: Needham-Schroeder-Protokoll



- einfaches Authentifizierungsprotokoll
- Ziel: Agenten A(lice) und B(ob) vereinbaren gemeinsames Geheimnis $\langle N_A, N_B \rangle$, bestehend aus einem Paar von Zufallszahlen („nonces“) N_A und N_B.
- Die Kommunikation erfolgt über ein unsicheres (d.h. öffentlich abhörbares) Netz mit asymmetrischer (public-key) Verschlüsselung.

Ablauf:

1. A generiert N_A und sendet $\langle A, N_A \rangle$ an B, verschlüsselt mit B's öffentlichem Schlüssel.

2. B generiert Zufallszahl N_B und antwortet mit $\langle N_A, N_B \rangle$.
3. A ist überzeugt, dass die Nachricht von B stammt („Authentizität"), weil nur B die Nonce N_A kennen konnte, und sendet N_B an B zurück.
4. B akzeptiert die Nachricht aus Schritt 3. Beide Agenten betrachten $\langle N_A, N_B \rangle$ als gemeinsames Geheimnis.

Annahmen:

- Der Verschlüsselungsalgorithmus sei nicht (bzw. nicht mit vertretbarem Aufwand) zu brechen, d.h. ein Angreifer kann eine Nachricht nur entschlüsseln, wenn er den zugehörigen privaten Schlüssel kennt.
- Nonces können nicht (mit vertretbarem Aufwand) erraten werden.
- Die Schlüssel von A und B seien nicht kompromittiert.

Frage: Ist das Protokoll sicher?

Darstellung als endliches Transitionssystem

Um das Protokoll durch ein endliches Transitionssystem darstellen und mit Hilfe von Modelchecking analysieren zu können, werden vereinfachende Annahmen getroffen:

- Beschränkung auf drei Agenten A(lice), B(ob) und I(ntruder)
- A und B nehmen an genau einem Ablauf des Protokolls teil (nicht unbedingt miteinander). Dabei beginnt A einen Protokollablauf, während B antwortet.
- Agent I kann maximal eine Nachricht lokal zwischenspeichern und maximal eine Zufallszahl generieren.

Aus diesen Annahmen folgt, dass maximal drei verschiedene Zufallszahlen N_A, N_B und N_I vorkommen. Diese können daher „vorberechnet" und durch symbolische Konstanten dargestellt werden.

Diese Vereinfachungen bedeuten, dass komplexere Fehlersituationen nicht erkannt werden, z.B. „Konfusion" bei mehreren verschränkten Protokollabläufen. Allgemein ist Modelchecking eher eine Technik zur Fehlersuche als zur Verifikation.

Modellierung in PROMELA (einer Modellierungssprache für Kommunikationsprotokolle)

PROMELA ist die Eingabesprache für den Modelchecker SPIN, der vor allem für die Analyse von Protokollen optimiert ist. Sie hat eine C-ähnliche Syntax.

Zunächst deklarieren wir Konstanten für die Typen der Nachrichten, die Namen der beteiligten Agenten, ihre Nonces und Schlüssel. (Diese werden ebenfalls durch symbolische Konstanten dargestellt, Ver- und Entschlüsselung von Nachrichten wird durch Record-Konstruktion und pattern matching modelliert.)

```
mtype = {msg1, msg2, msg3, alice, bob, intruder,
         nonceA, nonceB, nonceI, keyA, keyB, keyI, ok, fail};
```

Der „verschlüsselte Teil" einer Nachricht wird durch einen Record modelliert, der einen Schlüssel und zwei Datenfelder enthält.

```
typedef Crypt {
  mtype key, d1, d2;
}
```

Das Kommunikationsnetz wird durch einen globalen Datenkanal modelliert. Jede Nachricht auf diesem Kanal hat einen Typ, einen (intendierten) Empfänger und einen „verschlüsselten“ Inhalt. Um den Zustandsraum des Modells zu verkleinern, nehmen wir hier synchrone Kommunikation (Kanalkapazität 0) an; dies hat keinen Einfluss auf das Analyseergebnis.

```
chan network = [0] of {mtype, /* msg# */
                       mtype, /* receiver */
                       Crypt};
```

Die folgenden Variablen geben die (vermeintlichen) Kommunikationspartner von A und B sowie ihren Fortschritt im Protokoll an. Sie sind global deklariert, weil sie später in der Korrektheitsaussage des Protokolls benutzt werden.

```
mtype partnerA, partnerB;
mtype statusA, statusB = fail;
```

Ziel des Angreifers wird es sein, die Nonces von A und B zu entschlüsseln. Die folgenden Variablen geben an, inwiefern ihm dies gelungen ist.

```
bool knowNA, knowNB = false;
```

Das folgende Codefragment beschreibt das Verhalten von Agent A (die Modellierung von Agent B ist symmetrisch).[2]

```
active proctype Alice() {
  mtype  partner_key, partner_nonce;
  Crypt  data;

  if /* nichtdeterministische Auswahl des Partners */
  :: partnerA = bob; partner_key = keyB;
  :: partnerA = intruder; partner_key = keyI;
  fi;

  /* Versand der ersten Nachricht */
  data.key = partner_key;
  data.d1 = alice;
  data.d2 = nonceA;
  network ! msg1(partnerA, data);

  /* auf Antwort warten */
  network ? msg2(alice, data);
  /* sicherstellen, dass Schlüssel und Nonce korrekt sind */
  (data.key == keyA) && (data.d1 == nonceA);
  partner_nonce = data.d2;
```

[2] PROMELA kennt ein Konstrukt `if ... fi` zur nichtdeterministischen Auswahl zwischen mehreren Alternativen; diese werden jeweils durch `::` eingeleitet. Das Konstrukt `do ... od` beschreibt eine Endlosschleife, deren Rumpf ebenfalls mehrere Alternativen enthalten kann. Falls nötig, wird so lange gewartet, bis eine der Alternativen ausführbar ist.

```
  /* dritte Nachricht verschicken und Protokoll beenden */
  data.key = partner_key;
  data.d1 = partner_nonce;
  data.d2 = 0;
  network ! msg3(partnerA, data);
  statusA = ok;
}
```

Interessanter ist die Modellierung des Angreifers, da dieser kein festgelegtes Protokoll befolgt. Die Struktur des folgenden Codefragments ist eine Endlosschleife, deren Alternativen aus den möglichen Aktionen des Angreifers (Empfangen bzw. Abfangen einer Nachricht oder Versenden) bestehen.

```
active proctype Intruder() {
  mtype msg;
  Crypt data, intercepted;
  mtype icp_type;   /* Typ der abgefangenen Nachricht */

  do
  :: /* 1. Alternative: Empfangen oder Abfangen einer Nachricht */
     network ? msg (_, data) ->
     if /* evtl. zwischenspeichern, um sie später zu versenden */
     :: intercepted.key = data.key;
        intercepted.d1 = data.d1;
        intercepted.d2 = data.d2;
        icp_type = msg;
     :: skip;
     fi;
     /* Nachricht entschlüsseln, falls möglich und Nonces merken */
     if
     :: (data.key == keyI) ->
        if
        :: (data.d1 == nonceA || data.d2 == nonceA) -> knowNA = true;
        :: else -> skip;
        fi;
        if
        :: (data.d1 == nonceB || data.d2 == nonceB) -> knowNB = true;
        :: else -> skip;
        fi;
     :: else -> skip;
     fi;
  :: /* 2. Alternative: Versenden von Nachricht 1 an B */
     if /* zuvor abgefangene Nachricht versenden oder neu aufbauen */
     :: icp_type == msg1 -> network ! msg1(bob, intercepted);
     :: data.key = keyB;
        if /* eigene Identität benutzen oder als Alice ausgeben */
        :: data.d1 = alice;
        :: data.d1 = intruder;
        fi;
        if /* bekannte Nonces können verwendet werden */
        :: knowNA -> data.d2 = nonceA;
        :: knowNB -> data.d2 = nonceB;
        :: data.d2 = nonceI;
        fi;
        network ! msg1(bob, data);
     fi;
  :: /* übrige Alternativen zum Versenden von Nachrichten analog */
```

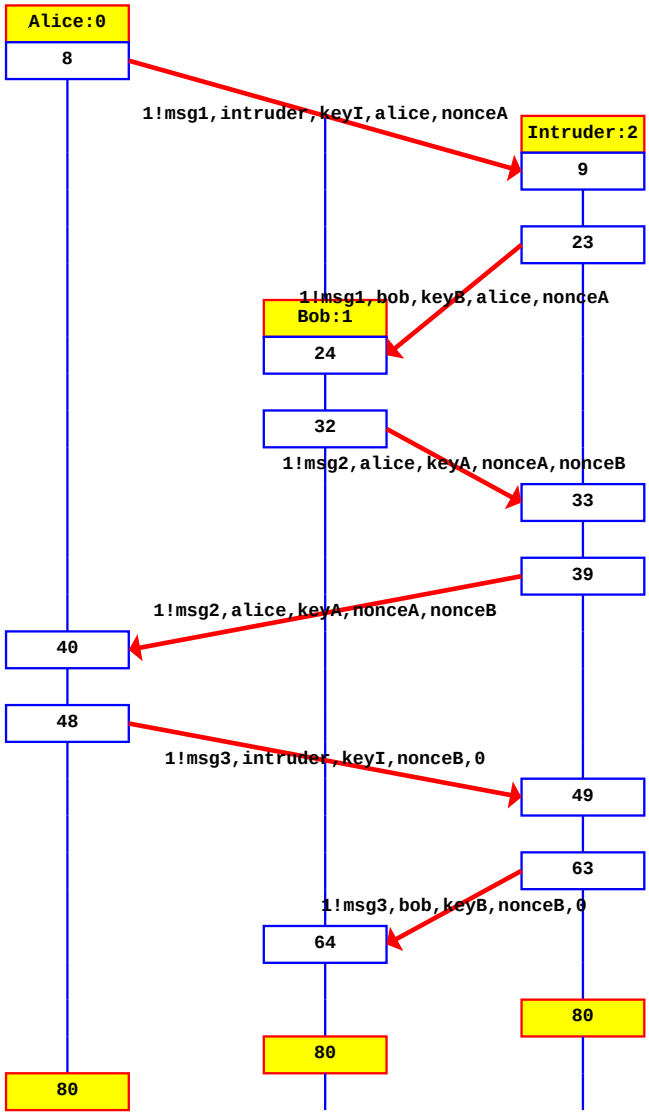


Abbildung 17: Gegenbeispiel zur Sicherheit des Needham-Schroeder-Protokolls.

```
      ...
  od;
}
```

Analyse des Modells. Die Sicherheit des Protokolls kann durch folgende Invarianten ausgedrückt werden:

$$(partnerA = \text{bob}) \Rightarrow \neg\, knowNA$$
$$(partnerB = \text{alice}) \Rightarrow \neg\, knowNB$$
$$(statusA = \text{ok} \wedge statusB = \text{ok}) \Rightarrow (partnerA = \text{bob} \Leftrightarrow partnerB = \text{alice})$$

Der Modelchecker SPIN findet in weniger als 1 Sekunde ein Gegenbeispiel und stellt es in dem in Abbildung 17 gezeigten Sequenzdiagramm dar. Dieser Ablauf kann wie folgt beschrieben werden:

1. A startet einen Lauf des Protokolls mit I. (Das ist durchaus sinnvoll: A wird i.a. nicht wissen, dass I ein Angreifer ist. Dieser kann sich z.B. als interessante Website tarnen und dadurch ehrliche Agenten veranlassen, mit ihm Kontakt aufzunehmen.)
2. Der Angreifer I benutzt die Nonce aus A's Nachricht, um seinerseits einen Protokollablauf mit B zu starten, wobei er sich als A ausgibt.
3. B antwortet an A. Diese Nachricht wird von I abgefangen und A wieder vorgespielt. (Das ist nicht unbedingt notwendig, solange keine vertrauenswürdigen Absenderangaben benutzt werden.) Beachten Sie, dass I die Nachricht nicht entschlüsseln kann und daher die Nonce von Agent B nicht erfährt.
4. A findet seine Nonce aus Nachricht 1 und antwortet, indem er die zweite Nonce seinem Partner, Agent I, schickt. Damit ist der Protokollablauf für A erfolgreich beendet.
5. Nun erhält I die Nonce aus B's Antwort in einer Form, die er entschlüsseln kann. Er kennt damit alle Nonces und sendet B's Nonce an B zurück.
6. Auch B kann damit seinen Protokollablauf erfolgreich abschließen.

Das Ergebnis dieses Ablaufs ist, dass A (korrekt) glaubt, seinen Partner I authentifiziert zu haben, während B fälschlicherweise davon überzeugt ist, ein gemeinsames Geheimnis mit A zu haben. Der Ablauf führt auch zur erfolgreichen Korrektur des Protokolls: die zweite Nachricht sollte durch $\langle B, N_A, N_B \rangle$ ersetzt werden. Denn dann bemerkt Agent A, dass die Nachricht von B statt seinem Partner I kommt und bricht das Protokoll ab.

Dieser Fehler war 17 Jahre lang unentdeckt geblieben, obwohl das Protokoll sehr einfach ist und in allen Kryptographie-Lehrbüchern beschrieben war. ◀

12.2 Büchi-Automaten

Zur Analyse komplexerer temporallogische Formeln sind Grundbegriffe der Theorie endlicher Automaten über unendlichen Wörtern nützlich.

Definition 61

1. *Ein* Büchi-Automat $\mathcal{B} = (Q, I, \delta, F)$ *über einem Alphabet* Σ *ist gegeben durch*
   - *eine endliche Menge* Q *von (Automaten-)Zuständen,*
   - *eine nichtleere Menge* $I \subseteq Q$ *von Anfangszuständen,*
   - *eine Übergangsrelation* $\delta \subseteq Q \times \Sigma \times Q$ *und*
   - *eine Menge* $F \subseteq Q$ *von akzeptierenden Zuständen.*
2. *Ein* Ablauf *von* $\mathcal{B}$ *über einem* ω*-Wort* $w = a_0 a_1 a_2 \ldots \in \Sigma^\omega$ *ist eine unendliche Folge* $\rho = q_0 q_1 q_2 \ldots$ *von Zuständen* $q_i \in Q$ *mit* $q_0 \in I$ *und* $(q_i, a_i, q_{i+1}) \in \delta$ *für alle* $i \in \mathbb{N}$*. Der Ablauf* ρ *heißt* akzeptierend*, wenn* $q_i \in F$ *gilt für unendlich viele* i*.*
3. *Die Sprache* $\mathcal{L}(\mathcal{B})$ *von* $\mathcal{B}$ *ist definiert durch*

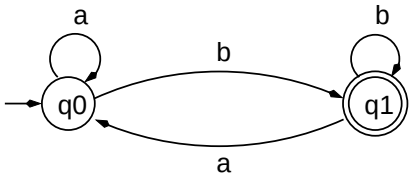
$$\mathcal{L}(\mathcal{B}) = \{ w \in \Sigma^\omega \mid \textit{es existiert ein akzeptierender Ablauf von } \mathcal{B} \textit{ über } w \}$$

Bemerkung:

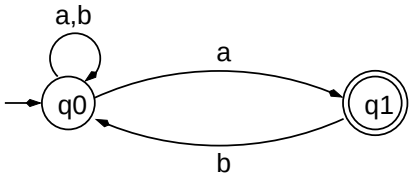
- Die Struktur eines Büchi-Automaten entspricht der eines gewöhnlichen NFA.
- Die Akzeptanz durch Endzustand wird ersetzt durch die Bedingung, unendlich oft einen akzeptierenden Zustand zu erreichen.

Beispiel 123: Büchi-Automaten

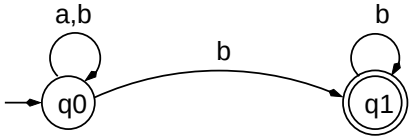
Der folgende deterministische Büchi-Automat über dem Alphabet $\{\text{'a'}, \text{'b'}\}$ akzeptiert genau die Wörter, die unendlich oft 'b' enthalten. Büchi-Automaten können graphisch genau wie NFAs dargestellt werden (akzeptierende Zustände sind doppelt umrandet).



Ähnlich definiert der folgende nichtdeterministische Büchi-Automat die Sprache der Wörter, die unendlich oft 'ab' enthalten.



Schließlich folgt hier ein Automat, der genau die Wörter akzeptiert, die nur endlich oft 'a' enthalten. Dazu „rät" der Automat, ab welchem Zeitpunkt kein 'a' mehr kommt, geht dann in den akzeptierenden Zustand über und „verifiziert", dass nur noch 'b' vorkommt. Man kann zeigen, dass es keinen deterministischen Büchi-Automaten gibt, der die selbe Sprache definiert.



◀

Eine nützliche Verallgemeinerung von Büchi-Automaten erlaubt mehrere Mengen von akzeptierenden Zuständen.

Definition 62 *Ein* verallgemeinerter Büchi-Automat $\mathcal{B} = (Q, I, \delta, \{F_1, \ldots, F_m\})$ *ist gegeben wie ein Büchi-Automat, enthält aber mehrere Mengen von Akzeptanzzuständen.*

Ein Ablauf $q_0 q_1 q_2 \ldots$ heißt akzeptierend, wenn jede Akzeptanzmenge F_j unendlich viele Zustände q_i des Ablaufs enthält.

Satz 63 *Für jeden verallgemeinerten Büchi-Automaten $\mathcal{B}$ gibt es einen (nicht verallgemeinerten) Büchi-Automaten $\mathcal{A}$, der dieselbe Sprache definiert.*

Beweis (Idee). Es sei $\mathcal{B} = (Q, I, \delta, \{F_1, \ldots, F_m\})$. Die Zustände von $\mathcal{A}$ sind Paare der Form (q,k) mit $q \in Q$ und $k \in \{1, \ldots, m\}$. Die Übergangsrelation $\delta_{\mathcal{A}}$ von $\mathcal{A}$ erlaubt Übergänge $((q,k), a, (q',k'))$ genau dann, wenn gilt:

- $(q,a,q') \in \delta$, d.h. Übergänge von $\mathcal{A}$ simulieren Übergänge von $\mathcal{B}$,
- $k' = \begin{cases} (k \bmod m) + 1 & \text{falls } q \in F_k \\ k & \text{sonst} \end{cases}$

  Das heißt, der Zähleranteil wird genau dann inkrementiert, wenn ein Zustand in der betreffenden Akzeptanzmenge besucht wurde.

Die akzeptierenden Zustände von $\mathcal{A}$ sind alle Zustände $(q,1)$ mit $q \in F_1$.

Die Idee dieser Konstruktion ist, dass $\mathcal{A}$ genau dann einen akzeptierenden Ablauf über einem Wort w hat, wenn dies auch für $\mathcal{B}$ gilt, da unendlich häufige Besuche in Zuständen $(q,1)$ mit $q \in F_1$ erzwingen, dass alle anderen Mengen F_k ebenfalls unendlich oft besucht werden. Q.E.D.

Die Theorie von Büchi-Automaten ist sehr ähnlich zu der von nichtdeterministischen endlichen Automaten—mit der Ausnahme, dass deterministische Büchi-Automaten echt schwächer sind als nichtdeterministische (vgl. Beispiel 123). Für das Modelchecking ist besonders die Entscheidbarkeit der Sprach-Leerheit wichtig. Gemäß Satz 63 gilt dies auch für verallgemeinerte Büchi-Automaten.

Satz 64 *Sei $\mathcal{B}$ ein Büchi-Automat. Es ist entscheidbar, ob $\mathcal{L}(\mathcal{B}) = \emptyset$ gilt.*

Beweis. Da Q endlich ist, ist $\mathcal{L}(\mathcal{B}) \neq \emptyset$ genau dann, wenn Zustände $q_0 \in I$ und $q_f \in F$ sowie endliche Wörter $x \in \Sigma^*$ und $y \in \Sigma^+$ existieren mit

$$q_0 \stackrel{x}{\Longrightarrow} q_f \qquad \text{und} \qquad q_f \stackrel{y}{\Longrightarrow} q_f$$

Dies ist genau dann der Fall, wenn der Graph von $\mathcal{B}$ eine starke Zusammenhangskomponente enthält, die einen akzeptierenden Zustand enthält und von einem Anfangszustand aus erreichbar ist. Letzteres ist (z.B. mithilfe des Tarjan-Algorithmus) mit Zeitkomplexität $O(|\,Q\,|)$ entscheidbar. Q.E.D.

Ist $\mathcal{B}$ verallgemeinerter Büchi-Automat, muss ein Zyklus $q_f \stackrel{y}{\Longrightarrow} q_f$ gesucht werden, der mindestens einen Zustand aus jeder Akzeptanzmenge enthält.

Büchi-Automaten und temporale Logik

Sei F eine quantorenfreie temporale Formel (ohne rigide Variablen). F enthält eine endliche Menge $\mathcal{V}$ von atomaren (Zustands-)Formeln; diese definiert ein Alphabet $\Sigma = \mathcal{P}(\mathcal{V})$. Jede Zustandsfolge kann als unendliches Wort $\sigma \in \Sigma^\omega$ aufgefasst werden, und F charakterisiert die Sprache

$$\mathcal{L}_F \;=_{\text{def}}\; \{\sigma \in \Sigma^\omega \mid [\![F]\!]_\sigma = \mathrm{T}\}$$

Unter diesem Blickwinkel sind temporale Formeln und Büchi-Automaten vergleichbar, und es ist interessant, ihre Ausdrucksstärke zu vergleichen.

Während temporallogische Formeln Eigenschaften von reaktiven Systemen „deklarativ“ beschreiben lassen, eignen sich Automaten besser zur algorithmischen Überprüfung, da sie endliche Objekte und damit besser handhabbar sind. Die Automaten aus Beispiel 123 legen nahe, dass sich typische temporale Eigenschaften durch Büchi-Automaten ausdrücken lassen.

Idee der Automaten-Konstruktion: $F \rightsquigarrow \mathcal{B}_F$

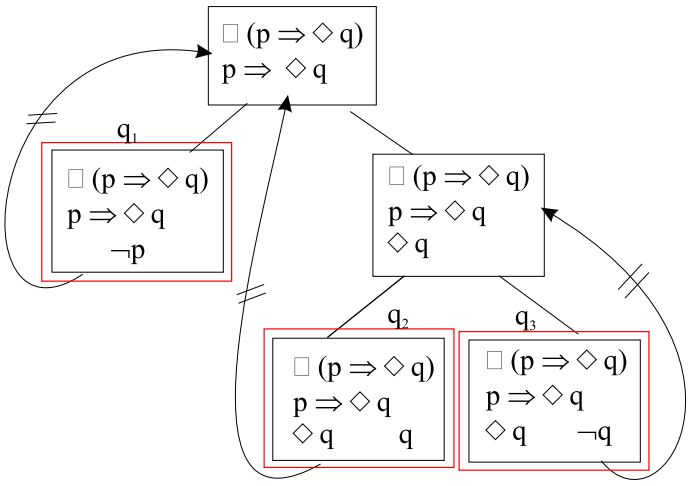
- Automaten-Zustände von $\mathcal{B}_F$ entsprechen maximal konsistenten Mengen von Teilformeln von F. Ein Zustand q „verspricht“, dass ein akzeptierender Ablauf ausgehend von q alle Formeln in q erfüllt.
- Anfangszustände sind genau die Zustände die F enthalten.
- Die Übergangsrelation stellt sicher, dass ein „aktueller Anteil“ sofort erfüllt wird, und eine „Restverpflichtung“ in den Folgezustand übertragen wird. Zum Beispiel muss für das Zutreffen einer Formel $\Box p$ sowohl p im aktuellen Zustand gelten als auch $\Box p$ ab dem folgenden Zustand erfüllt sein.
- Akzeptierende Zustände stellen sicher, dass Formeln $\Diamond p$ irgendwann erfüllt werden.

Komplexität dieser Konstruktion: Die Zustände entsprechen Mengen von Teilformeln der Ausgangsformel. Daher ist die Anzahl der Zustände i.a. exponentiell in der Länge der Formel.

Beispiel 124: Konstruktion eines Büchi-Automaten zur Formel $\Box(p \Rightarrow \Diamond q)$

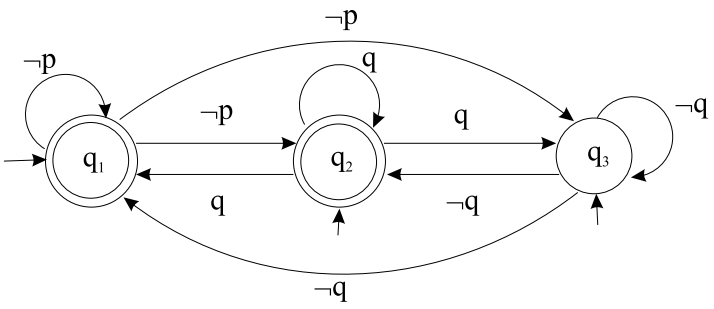
Im ersten Schritt konstruieren wir ein sogenanntes „Tableau“ zur gegebenen Formel. Dabei werden komplexe Formeln zerlegt und ggf. mehrere Nachfolgeknoten eingeführt: z.B. kann eine Formel $F \Rightarrow G$ dadurch erfüllt werden, dass entweder $\neg F$ oder G erfüllt wird. Knoten, die nur noch Formeln enthalten, die bereits zerlegt wurden oder nicht weiter zerlegt werden können, sind die sogenannten „states“ des Tableaus (die Knoten q_1, q_2 und q_3 der Abbildung). Ihre Nachfolger ergeben sich durch Bestimmung der „Restverpflichtung“ für den weiteren Ablauf:

- eine Formel $\Box F$ erzeugt $\Box F$ als Restverpflichtung,
- eine Formel $\Diamond F$ erzeugt $\Diamond F$ als Restverpflichtung, falls der Knoten auch $\neg F$ enthält.



Aus dem Tableau kann ein verallgemeinerter Büchi-Automat auf folgende Art abgelesen werden: Jeder „state“ des Tableaus entspricht einem Knoten des Büchi-Automaten. Anfangszustände sind die Automatenzustände, die die Ausgangsformel enthalten (im Beispiel sind das alle Zustände). Die Übergänge ergeben sich durch Verfolgung der Pfade zwischen den states im Tableau; sie werden mit den atomaren Formeln p (bzw. $\neg p$) beschriftet, die im Ausgangs-state enthalten sind (bzw. deren Negation enthalten ist). Für jede Formel der Form $\Diamond F$, die im Tableau vorkommt, wird eine Menge von akzeptierenden

Zuständen eingeführt, und zwar ist ein Zustand akzeptierend bzgl. $\Diamond F$, wenn der entsprechende state im Tableau entweder $\Diamond F$ nicht enthält oder F enthält. Im Beispiel gibt es eine Akzeptanzmenge, die der Formel $\Diamond q$ entspricht. Sie enthält die Zustände q_1 und q_2, aber nicht q_3.



Der entstehende Büchi-Automat akzeptiert genau die Abläufe, die die Ausgangsformel erfüllen. ◄

12.3 Automatenbasiertes Modelchecking

Die Theorie der (verallgemeinerten) Büchi-Automaten liefert ein Modelchecking-Verfahren, um zu entscheiden, ob alle Abläufe eines gegebenen Transitionssystems eine temporallogische Formel erfüllen.

Gegeben. Transitionssystem Γ, temporale Formel F.

Aufgabe. Entscheide, ob F für alle Abläufe von Γ erfüllt ist.

Lösung. Betrachte Γ als Büchi-Automaten mit trivialer Akzeptanzmenge $F_\Gamma = Z$.

$$
\begin{array}{c}
F \text{ gilt für jeden Ablauf von } \Gamma \\
\text{gdw.} \\
\text{kein Ablauf von } \Gamma \text{ erfüllt } \neg F \\
\text{gdw.} \\
\mathcal{L}(\Gamma) \cap \mathcal{L}(\mathcal{B}_{\neg F}) = \mathcal{L}(\Gamma \times \mathcal{B}_{\neg F}) = \emptyset
\end{array}
$$

Formal: Sei $\Gamma = (Z_\Gamma, I_\Gamma, \mathcal{A}, \delta_\Gamma)$ ein Transitionssystem und $\mathcal{B} = (Z_\mathcal{B}, I_\mathcal{B}, \delta_\mathcal{B}, F_\mathcal{B})$ ein Büchi-Automat, der die Negation der zu überprüfenden Formel charakterisiert.

Der *Produktautomat* $\Gamma \times \mathcal{B} = (Z, I, \delta, F)$ ist folgendermaßen definiert:

- $Z = Z_\Gamma \times Z_\mathcal{B}$ ist die Menge von Paaren aus Zuständen von Γ und $\mathcal{B}$.
- $I = I_\Gamma \times I_\mathcal{B}$, d.h. ein Paarzustand ist genau dann Anfangszustand, wenn seine Komponenten jeweils Anfangszustände des Transitionssystems bzw. von $\mathcal{B}$ sind.
- $(z,q) \longrightarrow (z',q')$ gdw.
  - $(z,A,z') \in \delta_\Gamma$ für ein $A \in \mathcal{A}$ und
  - $(q, z(\mathcal{V}), q') \in \delta_\mathcal{B}$

  Übergänge des Produktautomaten synchronisieren also die Übergänge von Γ und $\mathcal{B}$, wobei als Eingabe für $\mathcal{B}$ gerade die Belegung der atomaren Formeln durch den aktuellen Zustand des Transitionssystems genommen werden.

```
void dfs(StatePair s, boolean search_cycle) {
   Pair current = new Pair(s, search_cycle);
   if (! seen.contains(current)) {
      seen.insert(current);  trace.push(s);
      if (search_cycle && (s == seed)) {
         throw new CycleFoundException();
      }
      Enumeration succs = getSuccessorStatePairs(s);
      while (succs.hasMoreElements()) {
         dfs(succs.getNextElement(), search_cycle);
      }
      if (! search_cycle && s.isAccepting()) {
         seed = s;  dfs(s, true);
      }
      trace.pop();
} }
```

Abbildung 18: „On-the-fly“-Algorithmus zum Modelchecking.

- $F = Z_\Gamma \times F_\mathcal{B}$, d.h. ein Paar (z,q) ist akzeptierend, wenn q in $\mathcal{B}$ akzeptierender Zustand ist. (Ist Γ ein Transitionssystem mit Fairness, so muss diese Definition entsprechend angepasst werden.)

Dann gilt: $\Gamma \times \mathcal{B}$ akzeptiert genau die Abläufe von Γ, welche die zu überprüfende Formel verletzen. Daher ist $\mathcal{L}(\Gamma \times \mathcal{B}) = \emptyset$ genau dann, wenn es keine solchen Abläufe gibt, also Γ die untersuchte Eigenschaft besitzt.

Der in Abbildung 18 gezeigt Algorithmus ist eine Variante der Tiefensuchprozedur aus Abbildung 16 (die Abweichungen sind kursiv gedruckt). Die Prozedur arbeitet in zwei „Modi“, je nach Wert des Booleschen Parameters `search_cycle`. Im normalen Suchmodus (`search_cycle == false`) wird ein akzeptierendes Paar gesucht. Daraufhin wechselt die Suche in den Schleifensuch-Modus und untersucht, ob es einen nichttrivialen Zyklus zurück zu diesem Paar gibt. Falls ja, existiert ein akzeptierender Ablauf des Produktautomaten, und es kann ein Ablauf von Γ rekonstruiert werden, der die Eigenschaft verletzt.

Der Algorithmus vermeidet die vollständige Berechnung des Produktautomaten, indem das Erzeugen der Zustandspaare mit der Suche nach einem Akzeptanzzyklus kombiniert wird („on-the-fly“-Modelchecking). Er findet mindestens einen akzeptierenden Zyklus, falls ein solcher existiert. (Dies ist nicht ganz offensichtlich, da die Menge `seen` der bereits besuchten Paare auch nach einer erfolglosen Schleifensuche nicht gelöscht wird.) Seine Komplexität ist im schlechtesten Fall linear in der Größe von Γ und exponentiell in der Länge von F.

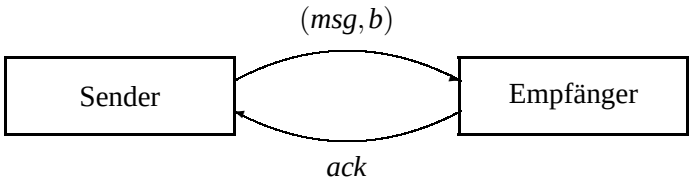
C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis: *Memory-efficient algorithms for the verification of temporal properties.* Formal Methods in System Design **1**, 275–288 (1992)

Beispiel 125: Analyse des Alternating-Bit-Protokolls mit SPIN

Die Aufgabe beim Alternating-Bit-Protokoll besteht darin, einen Strom von Daten von einem Sender zu einem Empfänger über ein unsicheres Medium zu übermitteln. Bei der Übertragung können Nachrichten verloren gehen, aber nicht verfälscht oder umgeordnet werden. Die bereits 1969 vorgeschlagene Lösung

K.A. Bartlett, R.A. Scantlebury, P.T. Wilkinson: *A note on reliable full-duplex transmission over half-duplex lines.* Communications of the ACM, 1969, vol. 12(5), S. 260-265.

führt ein Sequenzbit ein, anhand dessen der Empfänger feststellen kann, ob die aktuell übertragene Nachricht frisch ist. Dieses Sequenzbit wird als Acknowledgement an den Sender zurückgeschickt, der dadurch über den erfolgreichen Empfang informiert wird. Erhält der Sender kein Acknowledgement, versendet er die zuletzt gesandte Nachricht nochmals. Dabei wird angenommen, dass der Rückkanal ebenso unzuverlässig ist.



Um das Protokoll mithilfe von Modelchecking analysieren zu können, muss es als zustandsendliches Transitionssystem modelliert werden. Dazu nehmen wir eine beschränkte Kapazität `CH_LEN` der Übertragungskanäle und eine endliche (durch `MAXMSG` gegebene) Menge potenziell zu übertragender Nachrichten an.

```
#define MAXMSG  3      /* Daten 0, 1, ..., MAXMSG-1 */
#define CH_LEN  2      /* Kapazität der Übertragungskanäle */
```

Die folgenden Variablendeklarationen modellieren die Übertragungskanäle und die jeweils zuletzt gesendete bzw. empfangene Nachricht.

```
chan snd = [ CH_LEN ] of { bit, byte };
chan ack = [ CH_LEN ] of { bit };
byte lastSent = 0;
byte lastRcvd = 0;
byte currMsg;
```

Der folgende Prozess modelliert den Sender. Hier nehmen wir zur Vereinfachung an, dass zyklisch die Nachrichten `0,...,MAXMSG-1` versandt werden, sobald das aktuell verwendete Sequenzbit vom Empfänger als Acknowledgement erhalten wurde. Die Marken `SENDMSG` und `RCVACK` werden für die Formulierung der Korrektheitseigenschaften verwendet.

```
active proctype Sender() {
   bit sbit = 0;             /* Sequenzbit für aktuelle Nachricht */
   bit lack = sbit;          /* zuletzt vom Empfänger erhaltenes Sequenzbit */
   do
   :: if                     /* Nachricht senden */
      :: lack == sbit ->     /* neue Nachricht erzeugen */
         lastSent = (lastSent + 1) % MAXMSG;
         sbit = 1-sbit;
      :: else -> skip
      fi;
      SENDMSG: snd ! (sbit, lastSent)
   :: ack ? lack;  RCVACK: skip   /* Acknowledgement empfangen */
   od
}
```

Das Modell des Empfängers ist ähnlich. Doppelt empfangene Nachrichten werden ignoriert. Als Acknowledgement wird jeweils das zuletzt erhaltene Sequenzbit an den Sender zurückgeschickt.

```
active proctype Receiver() {
    bit rbit = 0;
```

```
    bit old_rbit = 0;
    do
    :: snd ? (rbit, currMsg); RCVMSG:       /* Nachricht empfangen */
       if
       :: rbit != old_rbit ->
          RCVNEW: old_rbit = rbit; lastRcvd = currMsg
       :: else -> skip
       fi
    :: ack ! rbit;  SNDACK: skip          /* Acknowledgement verschicken */
    od
}
```

Zur Modellierung von Nachrichtenverlusten führen wir einen separaten Prozess ein, der die aktuellen Nachrichten von den Kanälen entfernen kann.

```
active proctype LoseMsg() {
    do
    :: snd ? (_, _)        /* Nachrichtenverlust auf Datenkanal */
    :: ack ? _             /* Nachrichtenverlust auf Rückkanal */
    od
}
```

Frage 1: Kommen die Nachrichten in der richtigen Reihenfolge an?

Dies kann durch die Formel

$$\Box(\texttt{Receiver@RCVNEW} \Rightarrow \texttt{currMsg = (lastRcvd+1) mod MAXMSG})$$

ausgedrückt werden: das Prädikat `Receiver@RCVNEW` ist erfüllt, wenn der Empfängerprozess an der Marke `RCVNEW` steht, also gerade eine neue Nachricht empfangen hat.

Spin bestätigt, dass die Eigenschaft erfüllt ist. Die Laufzeit beträgt ca. 0.5 Sekunden.

Frage 2: Kommen alle Nachrichten an?

Wir formulieren diese Eigenschaft zunächst als

$$\Box(\texttt{Sender@SNDMSG} \Rightarrow \Diamond\texttt{Receiver@RCVMSG})$$

SPIN liefert nach kurzer Laufzeit (weniger als 0.1 Sekunden) das in Abbildung 19 dargestellte Gegenbeispiel, bei dem alle Nachrichten verlorengehen (alle gesandten Nachrichten werden vom Prozess `LoseMsg` empfangen).

Um solche Trivialfälle auszuschließen, formulieren wir die Eigenschaft um: wir nehmen an, dass immer wieder Nachrichten empfangen werden und fragen (zum Beispiel), ob jedes Versenden der Nachricht `2` von einem Empfang von `2` gefolgt wird.

$$\Box\Diamond\texttt{Receiver@RCVMSG} \wedge \Box\Diamond\texttt{Sender@RCVACK} \Rightarrow \Box(\texttt{Sender@SNDMSG} \wedge \texttt{lastSent=2} \Rightarrow \Diamond(\texttt{Receiver@RCVMSG} \wedge \texttt{lastRcvd=2}))$$

Diesmal bestätigt SPIN, dass die Eigenschaft zutrifft. Dazu sind ca. 170000 Zustandspaare zu überprüfen, die Laufzeit beträgt ca. 2.5 Sekunden (plus ca. 1 Sekunde für die Berechnung des Büchi-Automaten).

Frage 3: Werden immer wieder neue Nachrichten verschickt?

Wieder nehmen wir eine entsprechende Fairnessbedingung wie im vorigen Beispiel an und formulieren

$$\Box\Diamond\texttt{Receiver@RCVMSG} \wedge \Box\Diamond\texttt{Sender@RCVACK} \Rightarrow \Box\Diamond\texttt{lastSent=1} \wedge \Box\Diamond\texttt{lastSent=2}$$

Die Eigenschaft trifft zu; SPIN überprüft ca. 1.8 Millionen Zustandspaare in ca. 33 Sekunden.

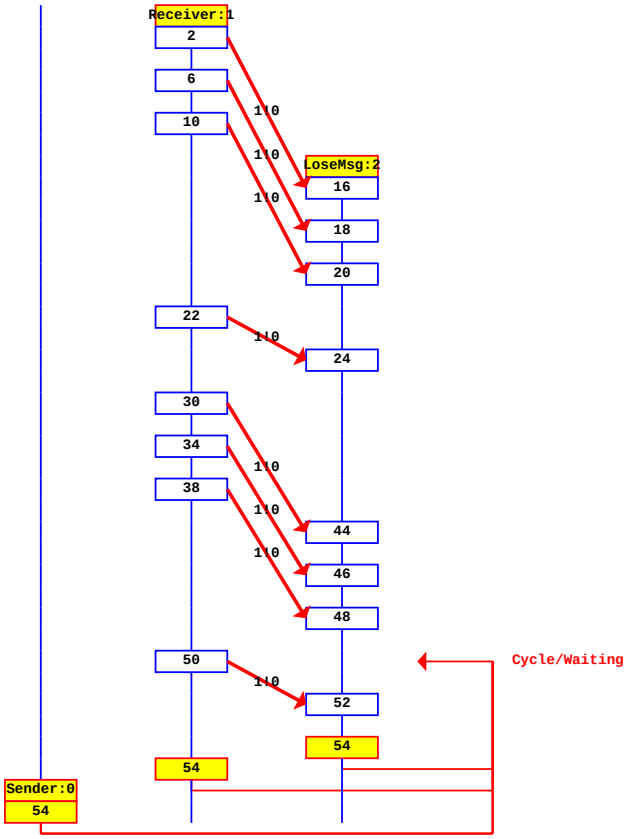


Abbildung 19: Gegenbeispiel zu Frage 2 in Beispiel 125.

Fazit: Die Eigenschaften sind für das beschriebene Modell erfüllt. Die Korrektheit des Protokolls ist also plausibel. Allerdings sollte das Modell durch Simulationsläufe validiert werden, um diese Aussage zu untermauern. Für eine tatsächliche Verifikation des Alternating-Bit-Protokolls wäre ferner zu zeigen, dass das vereinfachte Modell tatächlich (für die betrachteten Eigenschaften) „repräsentativ“ ist. Techniken, die solche Aussagen erlauben, werden im folgenden Abschnitt kurz angedeutet. ◀

12.4 Abstraktionstechniken

Das Hauptproblem beim Modelchecking stellt die sogenannte „Zustandsexplosion“ dar: die Anzahl der zu überprüfenden Zustände wächst i.a. exponentiell mit der Größe des Systems (z.B. Anzahl der Prozesse, betrachtete Datenbereiche, Kapazität der Kanäle, . . .).

Modelchecking-Verfahren mit expliziter Berechnung der Zustände wie in Abschnitt 12.3 sind bis zu ca. 10^7–10^8 zu überprüfenden Zustandspaaren verwendbar. Bei größeren Zustandsräumen kann die Menge `seen` der bereits überprüften Zustandspaare nicht mehr im Hauptspeicher gehalten werden; virtueller Speicher ist hierfür aber praktisch wertlos. Um den Speicherbedarf zu reduzieren, kann statt der Information über die tatsächlich besuchten Zustände eine Tabelle von Hashcodes gehalten werden. Kommt es zu einer Kollision, bedeutet dies, dass mögliche Fehler nicht entdeckt werden; trotzdem erreicht man mit dieser Technik eine relativ gute Abdeckung des Zustandsraums, solange Kollisionen nicht zu häufig werden.

Sogenannte *symbolische* Modelchecking-Algorithmen basieren auf kompakten Datenstrukturen zur Darstellung von Zustandsmengen und der Übergangsrelation. Sie berechnen in einem Schritt die Menge

alle Nachfolger- bzw. Vorgängerzustände einer Menge von Zuständen. Falls diese Techniken anwendbar sind (besonders im Bereich der Hardware-Verifikation), können Systeme im Bereich von 10^{100} und mehr potenziellen Zuständen überprüft werden.

Für noch größere Systeme bzw. für Systeme mit unendlichem Zustandsraum sind Abstraktionstechniken erfolgreich. Dabei wird die Korrektheit eines großen Systems (bzgl. einer bestimmten Eigenschaft) auf die Korrektheit einer kleinen Abstraktion zurückgeführt, die durch Modelchecking verifiziert werden kann. Um die Korrektheit der Abstraktion zu gewährleisten, genügt es in der Regel, einzelne Schritte des Systems zu betrachten, es ist also kein temporallogisches Schließen notwendig. Daher können normale prädikatenlogische Beweiser oder sogar automatische Techniken des Compilerbaus („abstract interpretation“) zum Einsatz kommen.

Wir deuten im folgenden die Technik sogenannter Boolescher Abstraktionen an. Dazu sei $\mathcal{P}$ eine endliche Menge von Zustandsformeln und O eine Menge von fundierten Ordnungen. Für $\prec \in O$ bezeichne $\preceq$ den reflexiven Abschluss von $\prec$; die Menge $O^{=}$ erweitere O um die Ordnungen $\preceq$.

Definition 65 *Ein* Boolesches Transitionssystem *(BTS) ist gegeben als* $\Gamma = (Z, I, \mathcal{A}, \delta, W, S, o)$ *mit:*

- $(Z, I, \mathcal{A}, \delta, W, S)$ *ist ein markiertes FTS mit endlicher Menge von Zuständen; jeder Zustand* $n \in Z$ *ist eine Menge von Formeln in* $\mathcal{P}$ *oder ihrer Negationen,*
- *es gelte* $\tau \notin \mathcal{A}$*; die Relation* δ_τ *erweitere* δ *um Übergänge* (n, τ, n) *für alle* $n \in Z$*,*
- *o ist eine Abbildung, die jeder Kante* $(n, A, m) \in \delta$ *eine endliche Menge* $\{(t_1, \prec_1), \ldots, (t_n, \prec_n)\}$ *von Paaren von Termen und Ordnungen* $\prec_i \in O^{=}$ *zuordnet,*

Boolesche Transitionssysteme sind endliche Transitionssysteme mit Fairness, deren Knoten mit Zustandsformeln beschriftet sind. Die Kanten können Annotationen tragen, um Informationen über fundierte Ordnungen darzustellen.

Im folgenden bezeichnen wir mit *n* sowohl die Knoten eines BTS als auch die Konjunktion der Formeln in *n*.

Definition 66 *Sei* $\Gamma = (Z, I, \mathcal{A}, \delta, W, S, o)$ *ein Boolesches Transitionssystem.*

Ein Ablauf *von* Γ *über einer Zustandsfolge* $\sigma = s_0 s_1 s_2 \ldots$ *hat die Form* $n_0 \xrightarrow{A_0} n_1 \xrightarrow{A_1} n_2 \ldots$*, so dass gilt:*

- $n_0 \xrightarrow{A_0} n_1 \xrightarrow{A_1} n_2 \ldots$ *ist ein Ablauf des FTS* $(Z, I, \mathcal{A} \cup \{\tau\}, \delta_\tau, W, S)$*,*
- $[\![n_i]\!]_{s_i} = \mathrm{T}$ *für alle* $i \in \mathbb{N}$*,*
- *für alle* $i \in \mathbb{N}$ *mit* $A_i \in \mathcal{A}$ *und alle* $(t, \prec) \in o(n_i, A_i, n_{i+1})$ *gilt* $[\![t' \prec t]\!]_{s_i, s_{i+1}} = \mathrm{T}$*,*
- *für alle* $i \in \mathbb{N}$ *mit* $A_i = \tau$ *und alle* $(t, \prec) \in o(n_i, A_i, m)$ *gilt* $[\![t' \preceq t]\!]_{s_i, s_{i+1}} = \mathrm{T}$*.*

Γ akzeptiert σ*, falls ein Ablauf von* Γ *über* σ *existiert.*

Abläufe eines BTS sind faire Abläufe, die alle Knoten- und Kantenmarkierungen erfüllen. Dabei sind auch (mit τ markierte) „Stotterübergänge“ zugelassen.

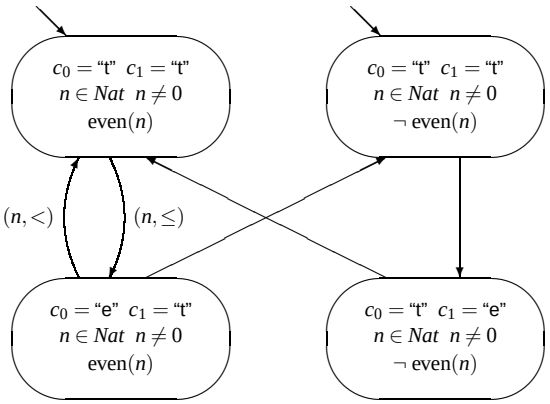
Der Beweis einer Implikation *Spec* $\Rightarrow$ *Prop* (für TLA-Formeln *Spec* und *Prop*) mit Hilfe eines BTS Γ erfolgt in zwei Schritten:

1. Jede Zustandsfolge, die *Spec* erfüllt, wird von Γ akzeptiert (Korrektheit der Abstraktion).
2. Jede von Γ akzeptierte Zustandsfolge erfüllt *Prop*.

Beispiel 126: Boolesches Transitionssystem für „dining mathematicians“

Idee: Zwei Prozesse (mit Zuständen „thinking“ und „eating“) sichern den gegenseitigen Ausschluss des Zustands „eating“ über eine gemeinsame Integer-Variable n. Je nachdem, ob n gerade oder ungerade ist, darf einer der Prozesse „essen“.

$$
\begin{array}{rcl}
\mathit{Init} & \equiv & \wedge\, n \in \mathbb{N} \wedge n \neq 0 \\
 & & \wedge\, c_0 = \text{“t”} \wedge c_1 = \text{“t”} \\
\mathit{Eat}_0 & \equiv & \wedge\, c_0 = \text{“t”} \wedge \mathrm{even}(n) \\
 & & \wedge\, c_0' = \text{“e”} \wedge c_1' = c_1 \wedge n' = n \\
\mathit{Thk}_0 & \equiv & \wedge\, c_0 = \text{“e”} \wedge c_0' = \text{“t”} \\
 & & \wedge\, n' = n \,\mathrm{div}\, 2 \wedge c_1' = c_1 \\
\mathit{Eat}_1 & \equiv & \wedge\, c_1 = \text{“t”} \wedge \neg\, \mathrm{even}(n) \\
 & & \wedge\, c_1' = \text{“e”} \wedge c_0' = c_0 \wedge n' = n \\
\mathit{Thk}_1 & \equiv & \wedge\, c_1 = \text{“e”} \wedge c_1' = \text{“t”} \\
 & & \wedge\, n' = 3 * n + 1 \wedge c_0' = c_0 \\
\mathit{Next} & \equiv & \mathit{Eat}_0 \vee \mathit{Thk}_0 \vee \mathit{Eat}_1 \vee \mathit{Thk}_1 \\
v & \equiv & (c_0, c_1, n) \\
\mathit{DM} & \equiv & \mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge \mathrm{WF}_v(\mathit{Next})
\end{array}
$$



Alle Kanten seien mit „*Next*“ markiert, und es sei schwache Fairness für *Next* vorausgesetzt.

Intuitiv repräsentiert das rechts abgebildete BTS die Abläufe der Spezifikation *DM*. Zum Beispiel kann im linken unteren Zustand nur die Aktion Thk_0 ausgeführt werden. Nach der Division durch 2 kann n gerade oder ungerade sein, und daher sind die beiden oberen Zustände gerade die möglichen Nachfolgezustände. ◀

Um die Korrektheit der Abstraktion gegenüber einer Spezifikation *Spec* zu beweisen, gibt der folgende Satz hinreichende Beweisverpflichtungen an, die kein temporallogisches Schließen benötigen.

Satz 67 *Sei $\Gamma = (Z, I, \mathcal{A}, \delta, W, S, o)$ ein Boolesches Transitionssystem und $\mathit{Spec} \equiv \mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L$ eine TLA-Spezifikation. Falls die folgenden Bedingungen zutreffen, akzeptiert Γ jeden Ablauf, der Spec erfüllt.*

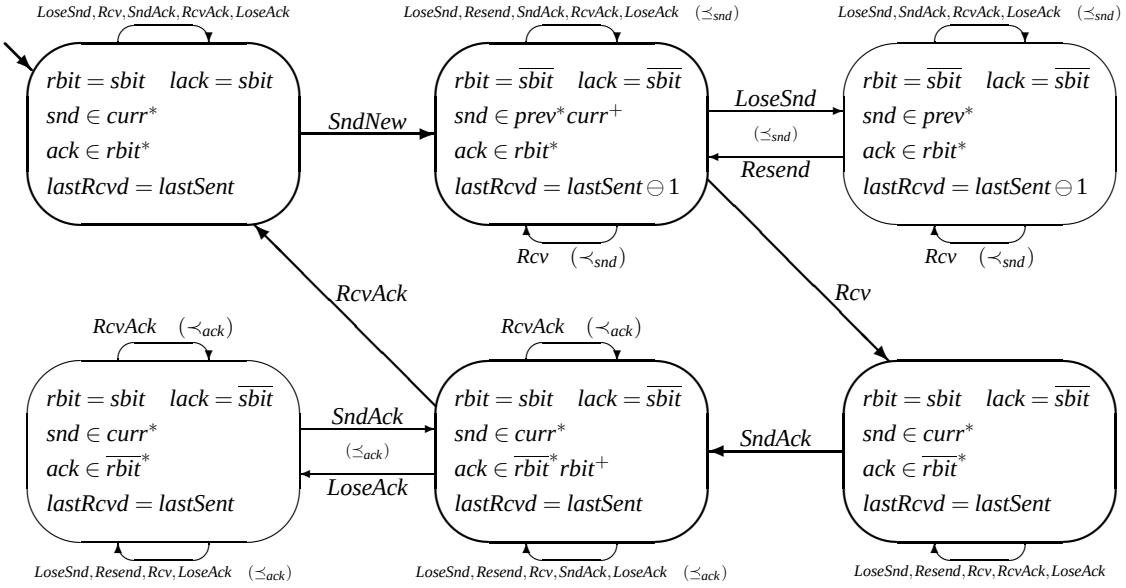
1. $\models \mathit{Init} \Rightarrow \bigvee_{n \in I} n.$
2. *Für alle $n \in Z$ gilt* $\models n \wedge [\mathit{Next}]_v \Rightarrow n' \vee \bigvee_{\{(A,m):(n,A,m)\in\delta\}} \langle A \rangle_v \wedge m'.$
3. *Für alle $n, m \in Z$, $A \in \mathcal{A}$ und alle $(t, \prec) \in o(n, A, m)$ gelten*
   (a) $\models n \wedge \langle A \rangle_v \wedge m' \Rightarrow t' \prec t$ *und*
   (b) $\models n \wedge [\mathit{Next}]_v \wedge n' \Rightarrow t' \preceq t.$
4. *Für jede Aktion $A \in W \cup S$:*
   (a) *Ist $A \in W$, so gilt* $\models \mathit{Spec} \Rightarrow \mathrm{WF}_v(A)$.
   (b) *Ist $A \in S$, so gilt* $\models \mathit{Spec} \Rightarrow \mathrm{SF}_v(A)$.
   (c) *Für alle $n \in Z$, in denen A ausführbar ist, gilt* $\models n \Rightarrow \mathrm{ENABLED}\langle A \rangle_v$.
   (d) *Für alle $n, m \in Z$ mit $(n, A, m) \notin \delta$ gilt* $\models n \wedge \langle A \rangle_v \Rightarrow \neg\, m'$.

Dagegen kann der Nachweis temporallogischer Eigenschaften *Prop* (mit atomaren Formeln in $\mathcal{P}$) für alle Abläufe von Γ durch Modelchecking erfolgen. Die Ordnungsannotationen eines BTS werden dabei in Annahmen übersetzt, die besagen, dass ein Ablauf, der unendlich viele mit $(t, \prec)$ markierte Transitionen enthält, auch unendlich viele Transitionen enthalten muss, die weder mit $(t, \prec)$ noch mit $(t, \preceq)$ markiert sind. Dadurch wird beispielsweise der Zyklus zwischen den beiden linken Zuständen des BTS aus Beispiel 126 ausgeschlossen, und die folgenden Eigenschaften können durch Modelchecking verifiziert werden:

$$\Box(n \in Nat \wedge n \neq 0) \qquad \Box\neg\,(c_0 = \text{“e”} \wedge c_1 = \text{“e”})$$
$$\Box\Diamond(c_0 = \text{“e”}) \qquad \Box\Diamond(c_1 = \text{“e”})$$

Beispiel 127: BTS für das Alternating-Bit-Protokoll (vgl. Beispiel 125)

Mit Hilfe Boolescher Abstraktionen kann die Korrektheit des Alternating-Bit-Protokolls ohne Einschränkungen bzgl. der Kanalkapazität bzw. der zu Grunde liegenden Nachrichtenmenge bewiesen werden. Grundidee ist die Beobachtung, dass auf jedem Kanal eine Folge von höchstens zwei verschiedenen Nachrichten gleichzeitig enthalten sein kann: zunächst endlich oft die „alte“ Nachricht, die bereits empfangen wurde und ignoriert werden kann, und danach endlich oft die „neue“ Nachricht. Die genaue Betrachtung des Protokolls führt zu folgendem BTS, dessen Korrektheit mithilfe von Satz 67 nachgewiesen werden kann.



Dabei wurden folgende Abkürzungen verwendet:

$$\begin{array}{lll} curr & \equiv & \langle sbit, lastSent\rangle \\ prev & \equiv & \langle \overline{sbit}, lastSent \ominus 1\rangle \\ \prec_{snd} & \text{steht für} & (\#(snd \rhd \{lastSent\}), <) \\ \prec_{ack} & \text{steht für} & (\#(ack \rhd \{rbit\}), <) \end{array}$$

Die Korrektheit des Protokolls kann durch Modelchecking dieses BTS bewiesen werden. ◄

12.5 Zusammenfassung

- Für endliche Transitionssysteme existieren automatische Analyseverfahren.

- Einfache Invariantenprüfer erlauben bereits den Nachweis interessanter Eigenschaften wie der partiellen Korrektheit oder der Deadlock-Freiheit.
- Die Überprüfung allgemeiner temporallogischer Eigenschaften basiert auf automatentheoretischen Zusammenhängen und Algorithmen.
- Ist eine Eigenschaft nicht erfüllt, berechnen Modelchecker ein Gegenbeispiel. Daher sind sie wertvolle Debugging-Werkzeuge, die industriell mittlerweile weit verbreitet sind.
- Das Problem der Zustandsexplosion begrenzt die Anwendbarkeit des Modelchecking.
- Abstraktionstechniken versprechen auch größere Systeme analysieren zu können. Sie dokumentieren außerdem anschaulich die Funktionsweise reaktiver Systeme.