

Logik und formale Methoden

Vorlesungsskript

von
Burkhardt Renz

Version 1.0.1

Burkhardt Renz
Technische Hochschule Mittelhessen
Rev 1.0 – 7. April 2021

© 2021 by Burkhardt Renz



Dieses Dokument ist lizenziert unter einer Creative Commons Namensnennung - Weitergabe unter gleichen Bedingungen 4.0 International Lizenz (siehe <http://creativecommons.org/licenses/by-sa/4.0/>).

Inhaltsverzeichnis

Inhaltsverzeichnis	i
1 Einleitung	1
1.1 Klassiker der Logik	1
1.2 Mathematische Logik	6
1.3 Logik und Informatik	11
1.4 Programm der Veranstaltung	13
I Aussagenlogik	15
2 Aussagen und Formeln	16
3 Die formale Sprache der Aussagenlogik	19
4 Die Semantik der Aussagenlogik	25
4.1 Modell, Belegung	28
4.2 Wahrheitstafel	30
4.3 Semantische Äquivalenz und Substitution	30
4.4 Boolesche Operatoren und funktionale Vollständigkeit . .	32
5 Das Beweissystem des natürlichen Schließens	35
5.1 Schlussregeln	37
5.1.1 Konjunktion	37
5.1.2 Disjunktion	38
5.1.3 Implikation	38
5.1.4 Negation	39
5.1.5 Widerspruchsbeweis, EFQ	39
5.2 Beispiele für das natürliche Schließen	39
5.2.1 Gentzens Beispiel	39
5.2.2 Abgeleitete Regeln der Aussagenlogik	40
5.3 Beweisstrategien	42
5.4 Eigenschaften der Herleitbarkeit \vdash	44
5.5 Vollständigkeit des natürlichen Schließens	47

5.5.1	Korrektheit des natürlichen Schließens für die Aussagenlogik	49
5.5.2	Vollständigkeit des natürlichen Schließens für die Aussagenlogik	50
5.5.3	Ein konstruktiver Beweis für den Vollständigkeitsatz in der Aussagenlogik	53
5.5.4	Kompaktheitssatz	59
6	Normalformen	60
6.1	Negationsnormalform NNF	60
6.2	Konjunktive Normalform CNF	62
6.3	Disjunktive Normalform DNF	64
6.4	Normalformen und Entscheidungsprobleme	65
6.4.1	CNF und Gültigkeit	65
6.4.2	DNF und Erfüllbarkeit	65
6.4.3	CNF und die Vollständigkeit des natürlichen Schließens	65
7	Die Komplexität des Erfüllbarkeitsproblems	67
7.1	Das Erfüllbarkeitsproblem	67
7.1.1	Entscheidungsfragen der Aussagenlogik	67
7.1.2	Das Erfüllbarkeitsproblem	68
7.2	Komplexität von Algorithmen	68
7.2.1	Arten von Algorithmen	69
7.2.2	Laufzeit von Algorithmen	69
7.2.3	Klassen von Problemen und \mathcal{NP} -Vollständigkeit	70
7.3	Die Komplexität des Erfüllbarkeitsproblems	71
8	Hornlogik	73
9	Erfüllbarkeit und SAT-Solver	76
9.1	DIMACS-Format	76
9.2	Tseitin-Transformation	78
9.2.1	Beispiel	78
9.3	DPLL und CDCL	79
9.3.1	Idee des Algorithmus	81
9.3.2	Beispiele	81
9.3.3	Von DPLL zu CDCL	85
10	Anwendungen der Aussagenlogik in der Softwaretechnik	86
10.1	Anwendungen von SAT-Techniken	86
10.2	Statische Codeanalyse	87
10.3	Feature-Modelle für (Software-)Produktlinien	88

II Prädikatenlogik	92
11 Objekte und Prädikate	93
11.1 Elemente der Sprache der Prädikatenlogik	94
11.2 Prädikate und Relationen	95
11.3 Beispiele von Aussagen in der Prädikatenlogik	96
12 Die formale Sprache der Prädikatenlogik	97
12.1 Signatur, Terme, Formeln	97
12.2 Freie und gebundene Variablen	100
12.3 Substitution	101
13 Semantik der Prädikatenlogik	104
13.1 Modell/Struktur	104
13.2 Semantische Folgerung und Äquivalenz	108
13.3 Fundamentale Äquivalenzen der Prädikatenlogik	109
14 Natürliches Schließen in der Prädikatenlogik	110
14.1 Schlussregeln	111
14.1.1 Allquantor	111
14.1.2 Existenzquantor	112
14.1.3 Gleichheit	112
14.1.4 „Konstantenquantifizierung“	112
14.2 Beispiele	113
14.3 Vollständigkeit des natürlichen Schließens	120
14.3.1 Die Korrektheit des natürlichen Schließens in der Prädikatenlogik	120
14.3.2 Die Vollständigkeit des natürlichen Schließens in der Prädikatenlogik	120
14.3.3 Der Modellexistenzsatz	122
15 Unentscheidbarkeit der Prädikatenlogik	127
15.1 Das Postsche Korrespondenzproblem	128
15.2 Die Unentscheidbarkeit der Prädikatenlogik	128
15.3 Der Sonderfall endlicher Universen	131
16 Anwendungen der Prädikatenlogik in der Softwaretechnik	134
16.1 Spezifikation und Analyse von Eigenschaften von Programmen	134
16.2 Analyse von Software mit Alloy	136
16.2.1 Die Idee hinter Alloy	136
16.2.2 Beispiele für Analysen mit Alloy	137
16.2.3 Die Sprache Alloy und das Werkzeug Alloy Analyzer	138
16.2.4 Die Sprache Alloy	139
16.2.5 Der Alloy Analyzer — unter der Haube	140
16.2.6 Ein (kleines) Beispiel	141

III Lineare Temporale Logik	145
17 Dynamische Modelle	146
17.1 Konzept der Transitionssysteme	146
17.2 Beispiel eines Programms mit zwei Threads	147
17.3 Temporale Logik	148
18 Die formale Sprache der linearen temporalen Logik (LTL)	150
19 Die Semantik der linearen temporalen Logik (LTL)	152
19.1 Kripke-Struktur	152
19.2 Äquivalenzen von Formeln der LTL (in der Pfad-Semantik)	158
19.3 Typische Aussagen in der LTL	158
19.3.1 Sicherheitseigenschaft	159
19.3.2 Lebendigkeitseigenschaft	159
19.3.3 Fairness	159
19.4 Büchi-Automaten	161
19.4.1 Automaten und unendliche Wörter	161
19.4.2 Eigenschaften von Büchi-Automaten	163
19.4.3 Das Leerheitsproblem für Büchi-Automaten	165
19.4.4 Verallgemeinerte Büchi-Automaten	165
19.5 Erfüllbarkeit in der LTL	166
19.5.1 Negationsnormalform (NNF) in der LTL	166
19.5.2 Von LTL zum Büchi-Automat	167
19.6 Auswertung von Formeln in Kripke-Strukturen	175
20 Natürliches Schließen in der LTL	180
20.1 Markierte Formeln und relationale Aussagen	180
20.2 Regeln für das natürliche Schließen in der LTL	181
20.2.1 Regeln für Boolesche Junktoren	182
20.2.2 Regeln für relationale Aussagen	183
20.2.3 Regeln für temporale Junktoren	185
20.3 Beispiele für das natürliche Schließen in der LTL	187
20.4 Vollständigkeit des natürlichen Schließens in der LTL	191
20.4.1 Das Beweissystem \mathcal{L}	191
20.4.2 Vollständigkeit des natürlichen Schließens in der LTL	192
21 Anwendungen der LTL in der Softwaretechnik	198
21.1 Model Checking	198
21.1.1 Konzept des Model Checkings mit SPIN	198
21.1.2 Eine Überraschung für Mordechai Ben-Ari	200
21.1.3 Überprüfung kryptographischer Protokolle mit SPIN	207
21.1.4 Anwendungen in der Industrie	214
21.2 Zielenmodell in der Anforderungsanalyse	220

21.2.1 Das Systemmodell in KAOS	221
21.2.2 Das Zielemodell	221

Literaturverzeichnis**224**

Kapitel 1

Einleitung

Die Wissenschaft der Logik befasst sich mit den *Formen* des Denkens unter Absehen vom jeweiligen Inhalt. Es geht um das Argumentieren, das Überzeugen, um *zwingende* Schlussfolgerungen. Es ist die Wissenschaft, bei der sich das Denken gewissermaßen mit sich selbst beschäftigt. Dies kann man auf sehr unterschiedliche, auch fragwürdige, Weise tun. In der *formalen Logik* geht es um ein *Kalkül*, bei dem aus gegebenen Prämissen Schlussfolgerungen gezogen werden durch die Manipulation von *Symbolen* — etwas was Computer gut können, ohne auch nur den geringsten Schimmer von der Bedeutung dieser Symbolen zu haben.

1.1 Klassiker der Logik

Ein Beispiel für eine zwingende Schlussfolgerung ist etwa folgende Argumentation über natürliche Zahlen:

Wenn $p > 2$ und Primzahl ist, dann ist $p + 1$ keine Primzahl
7 > 2 und 7 ist prim

Also: 7 + 1 = 8 ist keine Primzahl

Kenntnisse über natürliche Zahlen helfen, diese Argumentation zu überprüfen: Wenn die natürliche Zahl p größer als 2 und prim ist, dann muss p ungerade sein, denn sonst wäre p durch 2 teilbar, also keine Primzahl. Ist p ungerade, dann ist $p + 1$ gerade, also da größer als 2 garantiert keine Primzahl. Die erste Prämisse der Argumentation ist also zutreffend. Die zweite auch — man kann ja einfach nach den echten Teilern von 7 suchen und findet nur die 1. Beide Prämissen treffen zu, die Schlussfolgerung über die Zahl 8 folglich auch, das *Also* ist gerechtfertigt.

Wenn man die folgende Schlussfolgerung betrachtet, dann hat sie strukturell denselben Aufbau:

Wenn es regnet, ist die Straße nass.
Es regnet.

Also: Die Straße ist nass.

Diese Struktur, nämlich

$$\frac{P \rightarrow Q \\ P}{\text{Also: } Q}$$

nennt man *Modus Ponens*, aus dem Lateinischen *ponere* = stellen, setzen, also der setzende Modus, die Schlussfigur, die die Aussage Q „setzt“.

Wir finden sie auch bei folgender Argumentation —

Wenn die Erde eine Kugel ist, dann ist 7 eine Primzahl.
Die Erde ist eine Kugel.

Also: 7 ist eine Primzahl.

— und rein formal betrachtet sind auch hier beide Prämissen zutreffend, also die Schlussfolgerung zwingend. Aber während bei dem Beispiel mit der nassen Straße ein inhaltlicher Zusammenhang der Aussagen besteht, ist dies in diesem Beispiel offensichtlich nicht der Fall. Die Aussage über die Erde und die Aussage über die Zahl 7 schließen sozusagen *windschief* aneinander vorbei. Anders gesagt: Die rein formale Betrachtung hat auch ihren Preis, reichlich *unsinnige* Aussagen werden als zutreffend erachtet. In der Symbiose von formaler Logik und Informatik spielt dies allerdings keine Rolle, denn die Systeme, die die Informatik baut (und sie selbst) sind *selbst* formale Systeme.

Die Beobachtung, dass man die *formale Struktur* von Argumentationen ohne Kenntnisse der Inhalte betrachten und sie als zwingende *Schlussregeln* sehen kann, hat vielleicht als erster Aristoteles¹ gemacht. Er hat *Syllogismen* (aus dem Altgriechischen für „Zusammenrechnen“, „logischer Schluss“) betrachtet:

Eine Deduktion (*syllogismos*) ist also ein Argument, in welchem sich, wenn etwas gesetzt wurde, etwas anderes als das Gesetzte mit Notwendigkeit durch das Gesetzte ergibt.

– Aristoteles: Topik I 1, 100a25–27

¹ Aristoteles (384–322 v. Chr.), griechischer Philosoph

Die Syllogismen haben immer zwei Prämissen und eine Konklusion. Ein oft zitiertes Beispiel ist:

Alle Menschen sind sterblich.
Alle Griechen sind Menschen.
—————
Also: Alle Griechen sind sterblich.

Dieser Syllogismus wird *Modus Barbara* genannt, weil er von zwei All-Aussagen zu einer Schlussfolgerung führt, die auch eine All-Aussage ist.

Aus der Perspektive von Aristoteles ergeben sich Fragen wie:

- Was sind *gültige* Schlussregeln?
- Was ist ein *Beweis*?
- Wann ist eine Theorie *widerspruchsfrei*?
- ...

Man kann die Syllogismen als eine frühe Variante der Prädikatenlogik mit unären Prädikaten sehen. Erst 1879 hat Gottlob Frege² in seiner „Begriffsschrift“ die Prädikatenlogik formalisiert und damit die Grundlage gelegt für die heutige formale Logik.

Aristoteles betont im obigen Zitat, dass in einem Syllogismus die Schlussfolgerung mit *Notwendigkeit* aus den Prämissen folgt. Dies zeichnet ja gerade die Logik aus, wie in folgendem Zitat auch Goethe sie charakterisiert:

Mephistopheles:

Erklärt Euch, eh Ihr weiter geht,
Was wählt Ihr für eine Fakultät?

Schüler:

Ich wünschte recht gelehrt zu werden,
Und möchte gern, was auf der Erden
Und in dem Himmel ist, erfassen,
Die Wissenschaft und die Natur.

Mephistopheles:

Da seid Ihr auf der rechten Spur;
Doch müßt Ihr Euch nicht zerstreuen lassen.

Schüler:

Ich bin dabei mit Seel und Leib;

² Gottlob Frege (1848–1925), deutscher Logiker, Mathematiker und Philosoph.

Doch freilich würde mir behagen
Ein wenig Freiheit und Zeitvertreib
An schönen Sommerfeiertagen.

Mephistopheles:

Gebraucht der Zeit, sie geht so schnell von hinten,

Doch Ordnung lehrt Euch Zeit gewinnen.

Mein teurer Freund, ich rat Euch drum

Zuerst Collegium Logicum.

Da wird der Geist Euch wohl dressiert,

In spanische Stiefeln eingeschnürt,

Daß er bedächtiger so fortan

Hinschleiche die Gedankenbahn,

Und nicht etwa, die Kreuz und Quer,

Irrlichteliere hin und her.

Dann lehret man Euch manchen Tag,

Daß, was Ihr sonst auf einen Schlag

Getrieben, wie Essen und Trinken frei,

Eins! Zwei! Drei! dazu nötig sei.

Zwar ist's mit der Gedankenfabrik

Wie mit einem Weber-Meisterstück,

Wo ein Tritt tausend Fäden regt,

Die Schifflein herüber hinüber schießen,

Die Fäden ungesehen fließen,

Ein Schlag tausend Verbindungen schlägt.

Der Philosoph, der tritt herein

Und beweist Euch, es müßt so sein:

Das Erst wär so, das Zweite so,

Und drum das Dritt und Vierte so;

Und wenn das Erst und Zweit nicht wär,

Das Dritt und Viert wär nimmermehr.

Das preisen die Schüler allerorten,

Sind aber keine Weber geworden.

Wer will was Lebendigs erkennen und beschreiben,

Sucht erst den Geist heraus zu treiben,

Dann hat er die Teile in seiner Hand,

Fehlt, leider! nur das geistige Band.

– Goethe: Faust - Der Tragödie erster Teil, Vers 1896 ff.

Goethe betont die Strenge, das Normative, das *Zwingende* der Logik — und betrauert das ebenso: ihm fehlt dabei das Lebendige und das „geistige Band“!

Auch Hegel hat einen Einwand gegen die Sichtweise des formalen Schließens:

Es ist überhaupt eine bloß subjektive Reflexion, welche die Beziehung der Terminorum in abgesonderte Prämisse und einen davon verschiedenen Schlußsatz trennt:

Alle Menschen sind sterblich,
Cajus ist ein Mensch,
Also ist er sterblich.

Man wird sogleich von Langeweile befallen, wenn man einen solchen Schluß heranziehen hört; – dies röhrt von jener unnützen Form her, die einen Schein von Verschiedenheit durch die abgesonderten Sätze gibt, der sich in der Sache selbst sogleich auflöst. Das Schließen erscheint vornehmlich durch diese subjektive Gestaltung als ein subjektiver *Notbehelf*, zu dem die Vernunft oder der Verstand da ihre Zuflucht nehme, wo sie nicht *unmittelbar* erkennen könne.

– Hegel: Logik II, Werke Bd. 6, S.358

Formales Schließen in der Form eines Aristotelischen Syllogismus erscheint Hegel als „langweilig“, weil man gewissermaßen als Schlussfolgerung nur herausbekommt, was man bereits sowieso in die Prämisse hineingesteckt hat. Für ihn ist es nur der *Schein* der Verschiedenheit der Aussagen, der die Trivialität des Schlusses ausmacht.

Hegel selbst interessiert in seiner Logik etwas anderes: Er analysiert die Leistungen des Denken im „Allgemeinen“: Was sagt jemand, wenn er sagt, dass *A* die *Bedingung* für *B* ist? Was bedeutet es, *A* als *Grund* für *B* zu bezeichnen? Was macht das *Wesen* eines untersuchten Gegenstandes aus? Allgemein: wie geht das Denken im Begreifen einer Sache vor?

Zwei Perspektiven auf Logik

Ein erstes Fazit dieses schnellen (und etwas eklektizistischen) Blicks auf Klassiker der Logik ist, dass man Logik durchaus unterschiedlich sehen kann:

- **Erste Perspektive** Logik als *Gesetze des Denkens* im Sinne von (begründeten) *Normen* für korrekte Schlussfolgerungen und Argumentationen
- **Zweite Perspektive** Logik als *Gesetze des Denkens* im Sinne von „wie geht Denken?“

Wir werden bei der Betrachtung der Logik in der mathematischen Argumentation noch ein weitere Sichtweise kennenlernen.

1.2 Mathematische Logik



Abbildung 1.1: Papyrusfragment der „Elemente“ des Euklid

Abbildung 1.1 zeigt ein Papyrusfragment der „Elemente“ des Euklid³, ein Werk, das nicht nur die Geometrie als Wissenschaft begründet hat, sondern auch die Denk- und Argumentationsweise der Mathematik, die *axiomatische Methode*.

Euklid hat seine Untersuchung der Geometrie begründet auf fünf Postulate (heute würde man dazu Axiome sagen), aus denen er dann mit einigen wenigen Schlussregeln alle seine Sätze über die Geometrie herleitet. Euklid hat außerdem Definitionen der Begriffe Punkt, Gerade usw. seinen Postulaten vorangestellt. Heute werden in der Mathematik solche Grundbegriffe nicht definiert, sondern sie ergeben sich aus ihren Eigenschaften, die durch die Axiome festgelegt sind.

Die Axiome von Euklid sind:

1. Zu jedem Paar von Punkten gibt es genau eine Gerade, die durch diese Punkte geht.
2. Eine beliebige Strecke kann man zu einer Geraden verlängern.
3. Zu jedem Mittelpunkt und Radius kann man einen Kreis ziehen.
4. Alle rechten Winkel sind einander gleich.
5. Zu jeder Geraden und einem Punkt außerhalb dieser Geraden gibt es genau *eine* parallele Gerade durch diesen Punkt.

³ Euklid von Alexandria (3. Jahrhundert v. Chr.), griechischer Mathematiker.

Was wird man von den Axiomen erwarten?

- *Widerspruchsfreiheit*: Aus den Axiomen soll sich der Widerspruch nicht herleiten lassen, denn sonst würde ja *alles* aus den Axiomen folgen!
- *Unabhängigkeit*: Es soll nicht möglich sein, ein Axiom aus den anderen zu deduzieren. Diese Frage hat sich insbesondere bezüglich des fünften Axioms von Euklid gestellt, dem Parallelenaxiom. Der Versuch, es aus den anderen Axiomen herzuleiten, hat zu interessanten Entdeckungen geführt, die wir gleich kennenlernen werden.
- *Vollständigkeit*: Ein Axiomensystem ist vollständig, wenn man alle Sätze, die man auf Basis der Terme des Systems formulieren kann, entweder beweisen oder widerlegen kann. Euklids Axiomensystem ist übrigens nicht vollständig. David Hilbert⁴ hat 1899 in seiner Schrift „Grundlagen der Geometrie“ ein vollständiges Axiomensystem für die euklidische Geometrie entworfen.

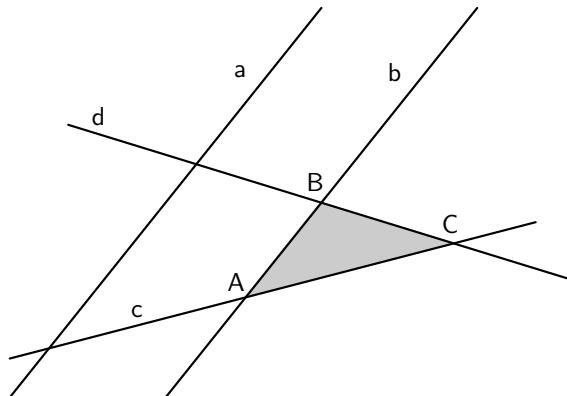


Abbildung 1.2: Euklidische Geometrie

Es ist nicht gelungen, das Parallelenaxiom aus den anderen Axiomen herzuleiten. Beim Versuch dies zu tun, wurde entdeckt, dass man durch Abwandlung des Parallelenaxioms neue interessante, sogenannte nicht-euklidische Geometrien definieren kann.

Euklidische Geometrie (der Ebene) Zu einer Geraden und einem Punkte außerhalb dieser Geraden, gibt es genau *eine* Parallelle. In dieser Geometrie ist die Summe der Winkel eines Dreiecks gerade 180° .

⁴ David Hilbert (1862–1943), deutscher Mathematiker.

Elliptische Geometrie (auf der Kugeloberfläche) In dieser Geometrie sind die Geraden gerade die Großkreise. Und es gilt: Zu einer Geraden und einem Punkt außerhalb derselben, gibt es *keine* Parallelen, d.h. verschiedene Geraden schneiden sich immer. In dieser Geometrie ist die Summe der Winkel eines Eulerschen Dreiecks immer größer als 180° . Abbildung 1.3 zeigt die Geraden a, b und c und das Eulersche Dreieck, das sie bilden.

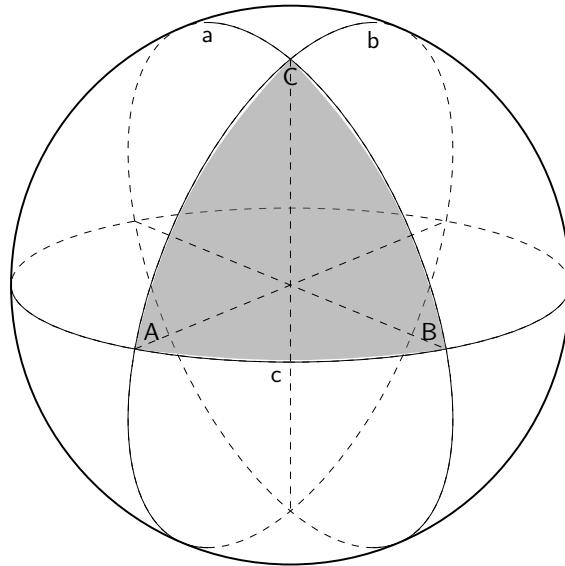


Abbildung 1.3: Elliptische Geometrie

Hyperbolische Geometrie In dieser Geometrie gibt es zu einer Geraden und einem Punkt außerhalb dieser Geraden *mindestens zwei* parallele Geraden. Es sind dann tatsächlich unendlich viele. Die Summe der Winkel im Dreieck ist immer kleiner als 180° .

Ein Modell der hyperbolischen Geometrie ist die Geometrie auf der offenen oberen Halbebene der Ebene, das Poincaré'sche Halbebene-Modell. Die Geraden sind Halbkreise oder senkrechte Halbgeraden in der offenen oberen Halbebene. In Abbildung 1.4 sind die Geraden a und b asymptotisch parallel, d.h. sie treffen sich auf der Grenze der Halbebene in einem sogenannten uneigentlichen Punkt, der nicht mehr zum Inneren der Halbebene gehört. a und c sind auch parallel, man sagt: ultraparallel, während b und c nicht parallel sind.

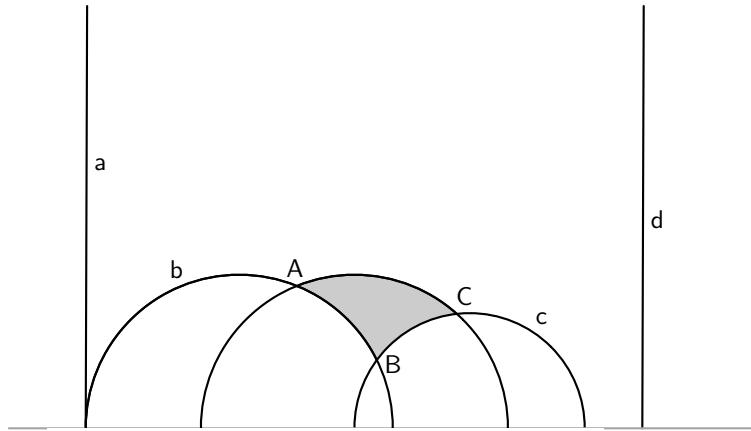


Abbildung 1.4: Hyperbolische Geometrie (Poincarés Halbebenen-Modell)

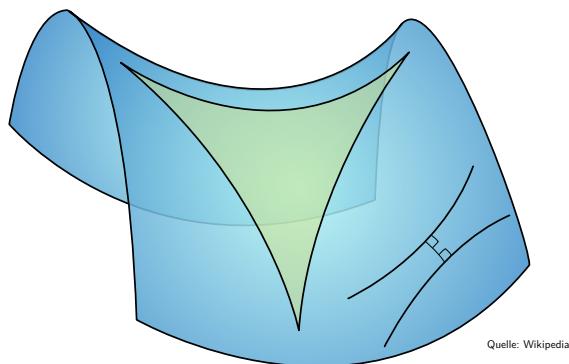


Abbildung 1.5: Hyperbolische Geometrie (Sattelfläche)

Theorie und Modell

In der mathematischen Logik hat sich (ausgehend von der Entdeckung nichteuklidischer Geometrien) eine Sprechweise von *Modell* durchgesetzt, die nicht verwechselt werden sollte mit dem Begriff des Modells, wie er zum Beispiel beim Softwaredesign mit der UML verwendet wird.

Im Beispiel der hyperbolischen Geometrie ist der Ausgangspunkt ein Axiomensystem und die Poincaré'sche Halbebene ist eine Struktur, in der sich die Terme des Axiomensystems interpretieren lassen und in der

die Axiome zutreffen. Es gibt für das Axiomensystem der hyperbolischen Geometrie auch andere Modelle, zum Beispiel die Sattelfläche. Abbildung 1.5 zeigt ein Dreieck auf der Sattelfläche.

Ein *Modell* einer *Theorie* ist in der mathematischen Logik also eine mit passenden Strukturen versehene Menge, auf die die Axiome und alle daraus ableitbaren Sätze der Theorie zutreffen. Es ist wichtig, diese Denk- und Sprechweise zu kennen, denn in Teil II der Veranstaltung wird ein *Model Finder* vorkommen und in Teil III geht es u.a. um *Model Checking* — beide Bezeichnungen kommen von der Sprechweise der mathematischen Logik.

Nebenbei bemerkt: es mag einem durch diese Sprechweise die Mathematik erscheinen wie ein Spiel mit willkürlich festgelegten Regeln, den Axiomen, für die sich, sofern widerspruchsfrei, schon eine passende Wirklichkeit, ein Modell wird finden lassen. Ganz so ist es nicht. Die Axiome zu finden ist die Aufgabe, einen Sachverhalt auf seinen ganz grundlegenden Kern zu reduzieren. Natürlich kann man auch mit Variationen der Axiome „spielen“, oft führt das zu gar nichts, manchmal eröffnet es neue Einsichten.

Seine Axiomatisierung der Geometrie schien für David Hilbert das Vorbild dafür zu sein, wie Mathematik zu denken habe. Die Mathematik sollte auf die Grundlage eines Axiomensystems gestellt werden, aus der die gesamte Mathematik mit definierten und als korrekt angesehenen Schlussregeln in Beweisen mit endlich vielen Schritten hergeleitet könnte.

Wir erörtern noch kurz, welche berechtigten allgemeinen Forderungen an die Lösung eines mathematischen Problems zu stellen sind: ich meine vor Allem, die, daß es gelingt, die Richtigkeit der Antwort durch eine endliche Anzahl von Schlüssen darzuthun und zwar auf Grund einer endlichen Anzahl von Voraussetzungen, welche in der Problemstellung liegen und die jedesmal genau zu formuliren sind. Diese Forderung der logischen Deduktion mittelst einer endlichen Anzahl von Schlüssen ist nichts anderes als die Forderung der Strenge in der Beweisführung.

– David Hilbert: Vortrag auf dem internationalen Mathematiker-Kongress Paris 1900

Dritte Perspektive auf die Logik

Man kann die bisherigen Erläuterungen so zusammenfassen: Wir betrachten Logik als eine formale Sprache, deren *Syntax* präzise und eindeutig definiert ist. Die *Semantik* der Symbole der Sprache ergibt sich durch Modelle der Sprache. Und ein *Beweissystem* mit Axiomen und Schlussregeln erlaubt es durch rein symbolische Manipulation von Ausdrücken

der Sprache neue Formeln herzuleiten.

- **Dritte Perspektive** Logik als *Kalkül* einer formalen Sprache — Syntax, Semantik und Beweissystem.

Und mit dieser dritten Perspektive sind wir nun schon gleich beim Thema Logik und Informatik. Denn mit formalen Sprachen und Umformungen von Ausdrücken kennt sich die Informatik ja bestens aus. Doch bevor wir auf falsche Ideen kommen, lassen wir noch Henri Poincaré⁵ zu Wort kommen:

Wer einer Schachpartie beiwohnt, dem wird es zum Verständnis der Partie nicht genügen, die Regeln über den Lauf der Figuren zu kennen. Das würde ihm nur erlauben zu erkennen, daß jeder Zug den Regeln entsprechend gespielt wurde, und dieser Vorzug hätte sehr wenig Wert. Es wäre jedoch das gleiche, wie es dem Leser eines mathematischen Buches ginge, wenn er nur Logiker wäre. Die Partie verstehen, das ist etwas ganz anderes, das heißt wissen, warum der Spieler mit dieser Figur zieht anstatt mit jener anderen, was er auch hätte tun können, ohne die Spielregeln zu übertreten; das heißt den inneren Grund zu erkennen, der aus dieser Reihe aufeinanderfolgender Züge ein organisches Ganzes macht. Mit viel mehr Grund ist diese Fähigkeit dem Spieler selbst nötig, das heißt dem Erfinder.

– Henri Poincaré: Der Wert der Wissenschaft, S.15

1.3 Logik und Informatik

[Symbolic] Logic and computer science share a symbiotic relationship. Computers provide a concrete setting for the implementation of logic. Logic provides language and methods for the study of theoretical computer science.

– [Hed04], Shawn Hedman: A First Course in Logic, S.xiv]

Diese enge Beziehung zwischen formaler Logik und Informatik zeigt sich in vielen Teilgebieten der Informatik, etwa:

Digitaltechnik

Kombinatorische Schaltungen implementieren logische Operatoren. In der Digitaltechnik als einem Teilgebiet der technischen Informatik spielt also die Aussagenlogik eine ganz grundlegende Rolle.

⁵ Henri Poincaré (1854–1912), französischer Mathematiker.

Komplexitätstheorie

In diesem Teilgebiet der Informatik geht es um die Komplexität algorithmisch behandelbarer Probleme, insbesondere was die Rechenzeit in Abhängigkeit von der Größe der Eingabe angeht. In der Komplexitätstheorie ist die Frage immer noch ungeklärt, ob effizient verifizierbare Probleme (der Klasse \mathcal{NP}) sich auch in polynomieller Zeit lösen lassen, also auch in der Klasse \mathcal{P} sind. Ein oder vielleicht *das* Musterbeispiel für ein Problem der Klasse \mathcal{NP} ist das Problem der Erfüllbarkeit einer Formel der Aussagenlogik.

Datenbanken

Relationale Datenbanken kann man sehen als endliche Modelle von Sprachen der Prädikatenlogik. Datenbankabfragen kann man deshalb auch in Formeln der Prädikatenlogik übersetzen. Man kann vereinfacht sagen, dass heutiges SQL als Sprache die Ausdrucksmächtigkeit der Prädikatenlogik mit transitivem Abschluss hat. Die Solidität relationaler Datenbanksysteme ist sicherlich auch darauf zurückzuführen, dass mit der relationalen Algebra, die auf der Prädikatenlogik basiert, eine mathematische Grundlage existiert.

Logik und Softwaretechnik

In dieser Veranstaltung interessieren wir uns aber insbesondere für den Zusammenhang zwischen Logik und Softwaretechnik.

A specification is a written description of what a system is supposed to do. Specifying a system helps us understand it. It's a good idea to understand a system before building it, so it's a good idea to write a specification of a system before implementing it.

...

Our basic tools for writing specifications is mathematics. Mathematics is nature's way of letting you know how sloppy your writing is. It's hard to be precise in an imprecise language like English or Chinese. In engineering, imprecision can lead to errors. To avoid errors, science and engineering have adopted mathematics as their language.

– [Lam02], Leslie Lamport: Specifying Systems, S.1f.]⁶

Da wir mit einem Computerprogramm letztlich eine formale Beschreibung des Verhaltens einer abstrakten Maschine formulieren, müssen

⁶ Leslie Lamport, amerikanischer Informatiker. Mehr zum Thema Spezifikation in Leslie Lamperts Vortrag *Thinking for Programmers*, URL: <http://channel9.msdn.com/Events/Build/2014/3-642>.

wir in der Softwaretechnik im Grunde den Beweis erbringen, dass diese Beschreibung die Anforderungen an das Programm tatsächlich erfüllt.

A solution of the problem [of providing a software based machine to fulfill a purpose in the real world, i.e. software development] must be based on at least the following descriptions:

- **requirement \mathcal{R}** : a statement of the customer's requirement;
- **domain properties \mathcal{W}** : a description of the given properties of the problem world;
- **specification \mathcal{S}** : a specification of the machine's behaviour at its interface with the problem world; and
- **program \mathcal{P}** : a program describing the machine's internal and external behaviour in a language that the general-purpose computer can interpret.

To show that the problem is solved we must discharge a proof obligation whose form is, roughly:

$$(\mathcal{P} \Rightarrow \mathcal{S}) \wedge ((\mathcal{S} \wedge \mathcal{W}) \Rightarrow \mathcal{R})$$

- [Jac02, Michael Jackson: Where, Exactly, Is Software Development?]⁷

Man kann das auch etwas plakativ ausdrücken (im folgenden Zitat steht S für Spezifikation, E für Environment — bei Jackson die Domain Properties und R für Requirements):

... the more we realised the key role of the fundamental logic—that $S, E \vdash R$ is truly the $E = mc^2$ of requirements engineering.

- [Hal11, Anthony Hall: $E = mc^2$ Explained]⁸

1.4 Programm der Veranstaltung

Viele Fragestellungen in der Softwaretechnik und der Programmierung haben eine Darstellung in der *Aussagenlogik*, die wir im ersten Teil der Veranstaltung behandeln werden. Ein Beispiel ist die Beherrschung der Variabilität in Softwareproduktlinien, für die Featuremodelle eingesetzt werden, bei deren Analyse Techniken des *SAT-Solving*, der Lösung des Erfüllbarkeitsproblems der Aussagenlogik, zum Einsatz kommen.

⁷ Michael Jackson (not the singer), britischer Informatiker.

⁸ Anthony Hall, britischer Informatiker.

Man kann sich auch fragen, ob Spezifikationen widerspruchsfrei sind, ob sie gewünschte Eigenschaften tatsächlich erfüllen, oder ob sich Gegenbeispiele finden lassen. Wir werden im zweiten Teil der Veranstaltung nach der Diskussion der *Prädikatenlogik* als Werkzeug die Sprache *Alloy* und den *Alloy Analyzer* kennenlernen, mit dem man leichtgewichtig und interaktiv „Mikromodelle“ von Architekturen, Entwürfen, Software überprüfen kann.

Man kann auch so vorgehen, dass man gebaute oder konzipierte Softwaresysteme logisch exakt beschreibt und dann prüft, ob sie gewünschte Eigenschaften tatsächlich erfüllen. Die Modelle sind häufig dynamische Modelle und die gewünschten Eigenschaften werden als Formeln der *linearen temporalen Logik (LTL)* formuliert. Mit der Technik des *Model Checking* kann man insbesondere in verteilten Systemen beweisbar überprüfen, ob Eigenschaften etwa der Fairness, des wechselseitigen Ausschlusses und Ähnliches erfüllt sind.

Wir werden also drei Logiken behandeln und jeweils einen Blick auf Anwendungen in der Softwareentwicklung werfen:

1. Aussagenlogik
2. Prädikatenlogik
3. Lineare temporale Logik

Dabei wird bei der Diskussion der Grundlagen die oben skizzierte dritte Perspektive auf die Logik verwendet, d.h.

- Die formale Sprache der jeweiligen Logik, also die *Syntax*.
- Die *Semantik* der jeweiligen Logik, also Modelle in denen die Ausdrücke der formalen Sprache repräsentiert werden können.
- Ein *Beweissystem*, das es erlaubt aus gegebenen Formeln durch Umformungen nach gewissen Regeln andere Formeln herzuleiten, und damit Beweise zu führen. Es gibt verschiedene solche Beweissysteme. Wir werden das Beweissystem des *natürlichen Schließens* in allen drei Logiken verwenden. Das natürliche Schließen kann auch durch Software unterstützt werden, eine Software, die die jeweiligen Schritte bei der Anwendung der Regeln überprüft. Die *Logic Workbench lwb*⁹ hat eine Komponente, die das natürliche Schließen in allen drei Logiken unterstützt.

⁹ [Wiki zu Natural Deduction](#) mit lwb auf github.

Teil I

Aussagenlogik

Kapitel 2

Aussagen und Formeln

In der Aussagenlogik werden Aussagen betrachtet, die wahr oder falsch sein können. Solche Aussagen nennt man *wahrheitsdefinite* Aussagen. Es gibt viele andere Formen von Aussagen in natürlichen Sprachen, wie z.B. ironische Äußerungen, Fragen oder Aufforderungen. Die Sprache der Aussagenlogik ist eine *formale* Sprache, in der nur wahrheitsdefinite Aussagen verwendet werden.

Beispiele

P = „Göttingen ist nördlich von Frankfurt“

Q = „6 ist eine Primzahl“

R = „Jede gerade Zahl > 2 ist die Summe zweier Primzahlen“¹

aber nicht:

„Könnten Sie bitte die Türe schließen“, eine Aufforderung

„Wie spät ist es“, eine Frage

„Guten Tag“, ein Gruß

Der *Inhalt* der Aussagen ist für die Aussagenlogik nicht wirklich von Belang. Wir studieren *nicht* den Wahrheitsgehalt von Aussagen, sondern die *Beziehung* der Aussagen.

Wir können atomare Aussagen miteinander verbinden und daraus zusammengesetzte Aussagen, *Formeln* bilden. Die Verbindung wird durch *Junktoren* hergestellt - siehe Tabelle 2.1

¹ Bei dieser Aussage handelt es sich um die Goldbach-Vermutung, benannt nach dem Mathematiker [Christian Goldbach](#) (1690–1764).

Tabelle 2.1: Junktoren und weitere Symbole

\neg	„nicht“, not	Negation
\wedge	„und“, and	Konjunktion
\vee	„oder“, or	Disjunktion
\rightarrow	„impliziert“, implies	Implikation
\top	„wahr“, true , verum	Wahrheit
\perp	„falsch“, false , absurdum	Widerspruch

Beispiele

- $P \wedge Q$ „Göttingen ist nördlich von Frankfurt *und* 6 ist eine Primzahl“ ①
 $P \vee Q$ „Göttingen ist nördlich von Frankfurt *oder* 6 ist eine Primzahl“ ②
 $P \rightarrow Q$ „Göttingen ist nördlich von Frankfurt *impliziert* 6 ist eine Primzahl“ ③
 $Q \rightarrow P$ „6 ist eine Primzahl *impliziert* Göttingen ist nördlich von Frankfurt“ ④

Bemerkungen

- Die erste Aussage ist falsch. Allerdings muss man bei der Übertragung von Aussagen aus der natürlichen Sprache Vorsicht walten lassen:
 Die folgenden beiden Aussagen haben einen unterschiedlichen Sinn
 „Er ging zur Schule und ihm war langweilig.“
 „Ihm war langweilig und er ging zur Schule.“
 obwohl die Aussagen $P \wedge Q$ und $Q \wedge P$ in der formalen Sprache der Aussagenlogik äquivalent sind.
- Die zweite Aussage ist wahr.
- Die dritte Aussage ist falsch.
- Die vierte Aussage ist wahr – obwohl offensichtlicher Unsinn! Implikationen in der formalen Logik darf man nicht mit *Kausalität* verwechseln. Die Aussage ist wahr, weil in der formalen Logik das Prinzip *ex falso quodlibet* gilt: Aus einer falschen Voraussetzung darf man alles folgern. Warum diese Definition der Implikation sinnvoll ist, werden wir später sehen.
- Es gibt Aussagen, die wahr sind, egal welche Wahrheitswerte die beteiligten atomaren Aussagen haben, wie z.B.
 $((P \rightarrow Q) \rightarrow (\neg P \vee Q)) \wedge ((\neg P \vee Q) \rightarrow (P \rightarrow Q))$
 Solche Aussagen nennt man allgemeingültig oder *Tautologie*.

Wir haben in dieser einführenden Diskussion zwei Konzepte verwendet, ohne sie genau zu unterscheiden: Die *Syntax* der Aussagenlogik, die festlegt, welche Formeln wir aus atomaren Aussagen und Junktoren bilden können, sowie die *Semantik* der Aussagenlogik, bei der wir von Wahrheitswerten der atomaren Aussagen reden und aus diesen den Wahrheitswert von Formeln ermitteln können.

In den folgenden drei Kapiteln werden wir diese beiden Konzepte und ihren Zusammenhang auf systematische Weise untersuchen.

Kapitel 3

Die formale Sprache der Aussagenlogik

In der Sprache der Aussagenlogik kombiniert man atomare Aussagen mit Junktoren. Dabei gehen wir von einer gegebenen Menge \mathcal{P} von Aussagensymbolen sowie Junktoren aus. Genau genommen beziehen sich die folgenden Definitionen auf die Wahl dieser Menge und man sollte von *einer* Sprache der Aussagenlogik sprechen.

Definition 3.1 (Alphabet der Aussagenlogik). Das *Alphabet* der Sprache der Aussagenlogik besteht aus

- (i) einer Menge \mathcal{P} von Aussagensymbolen,
- (ii) den Junktoren: $\neg, \wedge, \vee, \rightarrow$
- (iii) der Konstanten: \perp
- (iv) den zusätzlichen Symbolen: $(,)$

Bemerkungen

- Für eine Sprache der Aussagenlogik nennt man die Wahl der Menge der Aussagensymbole sowie der Junktoren usw. auch die *logische Signatur*.
- Viele Autoren geben eine fixe (abzählbare) Menge von Aussagensymbolen vor, etwa $\mathcal{P} = \{P_0, P_1, P_2, \dots\}$ und sprechen dann von *der Sprache der Aussagenlogik*.¹

¹ Die Menge der Aussagensymbole muss übrigens nicht unbedingt abzählbar sein, sondern kann eine beliebige Menge sein, siehe [Rau08, S. 4 und Abschnitt 1.5]. Das ist interessant, wenn man den Kompaktheitssatz der Aussagenlogik verwendet, um Aussagen über unendliche Mengen zu beweisen. In dieser Vorlesung werden wir uns damit nicht befassen, weil wir uns auf Anwendungen der formalen Logik in Informatik und Softwaretechnik konzentrieren.

Für Anwendungen der Aussagenlogik in der Informatik und der Softwareentwicklung haben wir es jedoch mit endlichen Mengen zu tun und wir möchten den Aussagensymbole auch „sprechende“ Bezeichnungen geben können. Deshalb geht in unsere Definition des Alphabets die Wahl der Menge der Aussagensymbole ein.

- Die Bezeichnung von \neg, \wedge etc. als *Junktoren* soll unterstreichen, dass wir eine formale Sprache definieren. Natürlich aber darf man schon daran denken, dass mit \neg die *Negation* („nicht“), mit \wedge die *Konjunktion* („und“), mit \vee die *Disjunktion* („oder“) und mit \rightarrow die *Implikation* („impliziert“) gemeint sind.
- Die Konstante \perp steht für den *Widerspruch* („falsch“).
- Auch was die Junktoren angeht, treffen wir in der Definition eine Wahl. Wir könnten z.B. noch weitere Junktoren hinzunehmen, wie etwa \leftrightarrow oder auch Junktoren weglassen. Wir werden später sehen, dass sich aus der Semantik der Aussagenlogik ergibt, dass die Menge von Operatoren, die unsere gewählten Junktoren definieren *funktional vollständig* ist, d.h. jede beliebige Boolesche Funktion darstellen kann.

Definition 3.2 (Formeln der Aussagenlogik). Die *Formeln* der Aussagenlogik sind Zeichenketten, die nach folgenden Regeln gebildet werden:

- (i) Jedes Aussagensymbol ist eine Formel und auch \perp ist eine Formel.
- (ii) Ist φ eine Formel, dann auch $(\neg\varphi)$.
- (iii) Sind φ und ψ Formeln, dann auch $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ und $(\varphi \rightarrow \psi)$

Bemerkungen

- In der Definition der Formeln der Aussagenlogik verwenden wir auch die Implikation, wenn wir z.B. sagen: „Ist φ eine Formel, dann auch $(\neg\varphi)$ “. Man muss also unterscheiden, ob wir *über* die Logik sprechen — man sagt dann auch: wir verwenden die *Metasprache* — oder ob wir eine logische Formel angeben — dann verwenden wir die *Objektsprache*, die wir eben definiert haben.
- Die Symbole φ und ψ sind also *Variablen der Metasprache*, sie stehen als Platzhalter für *Formeln der Objektsprache*.
- Unsere Definition der Menge der Formeln der Aussagenlogik ist eine sogenannte *induktive* Definition.
- Eine Formel, die nur aus einem Aussagensymbol oder \perp besteht, nennt man auch *atomare* Formel oder *Primformel*.

Als *Grammatik* in Backus-Naur-Darstellung² können wir diese induktive Definition der Formeln der Aussagenlogik so ausdrücken:

$$\varphi ::= P \mid \perp \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi)$$

mit Aussagensymbolen $P \in \mathcal{P}$ und (bereits gebildeten) Formeln φ .

Zu einer Formel φ kann man ihren Syntaxbaum bilden:

Definition 3.3. Als *Syntaxbaum* bezeichnen wir einen endlichen Baum, dessen Knoten mit Formeln beschriftet sind und der folgende Eigenschaften hat:

- (i) Die Blätter sind mit Primformeln beschriftet.
- (ii) Ist ein Knoten mit einer Formel der Form $(\neg\varphi)$ beschriftet, dann hat er genau ein Kind, das mit φ beschriftet ist.
- (iii) Ist ein Knoten mit einer Formel der Form $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ oder $(\varphi \rightarrow \psi)$ beschriftet, dann hat er genau zwei Kinder und das linke ist mit φ , das rechte mit ψ beschriftet.

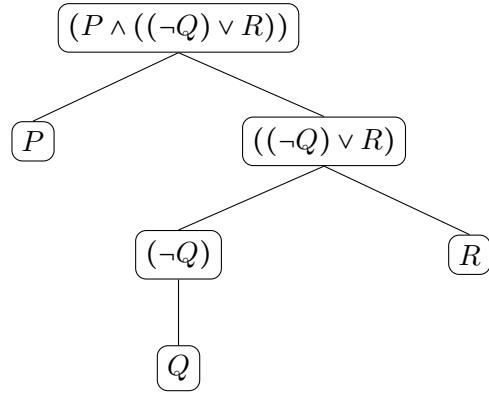
Ein Syntaxbaum repräsentiert die Formel, mit der die Wurzel des Baumes beschriftet ist.

Beispiel 3.1. Gegeben sei die Formel

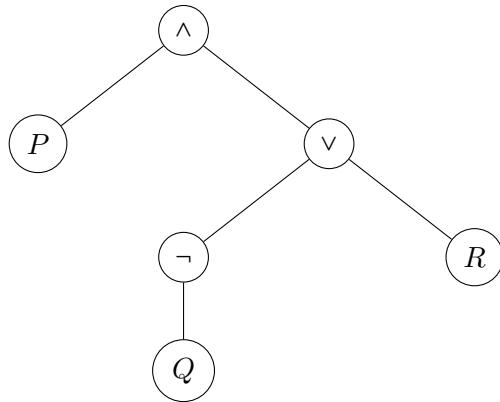
$$(P \wedge ((\neg Q) \vee R))$$

Der Syntaxbaum ist dann

² John W. Backus (1942–2007), amerikanischer Informatiker und Peter Naur (1928–2016), dänischer Informatiker (Tatsächlich war Naur nicht erfreut über diese Bezeichnung, er hätte „Backus-Normalform“ vorgezogen).



Wir verwenden oft die abgekürzte Darstellung des Syntaxbaums, in der die Knoten nicht mit den Subformeln bezeichnet werden, sondern mit dem Junktor. Nur an den Blättern des Baums kommen dann die Primformeln vor. In unserem Beispiel also



Bemerkung. Die Sprechweise „*der* Syntaxbaum einer Formel“ unterstellt, dass es nur *einen* solchen Baum zu einer gegebenen Formel gibt. Mit anderen Worten: die Grammatik aus der Definition der Formeln ist eindeutig.

Satz 3.1. *Jede Formel der Aussagenlogik hat genau einen Syntaxbaum.*

Zum Beweis von Aussagen über Formeln wendet man oft das Prinzip der *strukturellen Induktion* an.

Prinzip der strukturellen Induktion Sei \mathcal{E} eine Eigenschaft. Dann gilt $\mathcal{E}(\varphi)$ für alle Formeln φ , wenn gilt:

- (i) \mathcal{E} gilt für alle Primformeln und \perp .

- (ii) Gilt \mathcal{E} für φ , dann auch für $(\neg\varphi)$.
- (iii) Gilt \mathcal{E} für φ und ψ , dann auch für $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ und $(\varphi \rightarrow \psi)$.

Dieses Prinzip kann man anwenden, um obigen Satz zu beweisen.

Beweisidee für Satz 3.1. Man beweist den Satz in folgenden Schritten:
Zunächst zeigt man: Jede Formel hat eine gerade Anzahl von Klammern und zwar ebenso viele öffnende wie schließende Klammern.

Dann zeigt man: Jedes echte Anfangsstück einer Formel hat mehr öffnende als schließende Klammern.

Mit Hilfe dieser beiden Aussagen kann man dann den Satz beweisen. \square

Definition 3.4 (Subformel). Eine *Subformel* einer Formel φ ist ein Formel, die als Beschriftung an einem Knoten des Syntaxbaums von φ vorkommt.

Definition 3.5 (Rang einer Formel). Der *Rang* $rg(\varphi)$ einer Formel ist definiert durch

- (i) $rg(\varphi) = 0$, wenn φ eine Primformel.
- (ii) $rg((\varphi \square \psi)) = \max(rg(\varphi), rg(\psi)) + 1$, wobei \square für einen der Junktoren $\wedge, \vee, \rightarrow$ steht.
- (iii) $rg((\neg\varphi)) = rg(\varphi) + 1$.

Der Rang einer Formel ist gerade die Tiefe des Syntaxbaums, d.h. die maximale Pfadlänge von der Wurzel zu einem beliebigen Blatt.

Die Eindeutigkeit des Syntaxbaums erlaubt es uns, Konventionen zu vereinbaren, mit denen man Klammern „sparen“ kann. Man vereinbart, dass die Junktoren in folgender Reihenfolge binden, die stärkste Bindung zuerst: $\neg, \wedge, \vee, \rightarrow$. Außerdem sind \wedge und \vee linksassoziativ; \rightarrow ist rechtsassoziativ.

Präfix-Notation für Formeln der Aussagenlogik Für die Definition der Syntax der Aussagenlogik haben wir die *Infix-Notation* verwendet, wie das in den Lehrbüchern über formale Logik üblich ist.

Man kann stattdessen die *Präfix-Notation* verwenden und hat dann folgende Grammatik:

$$\varphi ::= P \mid \perp \mid (\neg \varphi) \mid (\wedge \varphi \dots) \mid (\vee \varphi \dots) \mid (\rightarrow \varphi \varphi)$$

In der Präfix-Notation kann man die Junktoren \wedge und \vee als *n-äre* Junktoren definieren.

In der **Logic Workbench (lwb)**, einer in Clojure geschriebenen Bibliothek von Funktionen für die Aussagen-, Prädikaten- und lineare temporale Logik wird Präfix-Notation mit folgenden Junktoren verwendet:

Tabelle 3.1: Junktoren für die Aussagenlogik in lwb

Junktor	Beschreibung	Arität
not	Negation	unär
and	Konjunktion	n-är
or	Disjunktion	n-är
impl	Implikation	binär
equiv	Biimplikation	binär
xor	Exklusives Oder	binär
ite	If-then-else	ternär

Beispiel 3.2. Die Formel

$$(P \wedge ((\neg Q) \vee R))$$

in Infix-Notation wird in der Präfix-Notation der Logic Workbench so geschrieben:

$$(\text{and } P \text{ (or (not } Q) \text{ R}))$$

Diese Schreibweise entspricht genau dem Syntaxbaum der Formel, dargestellt als verschachtelte Liste.

Kapitel 4

Die Semantik der Aussagenlogik

In der klassischen Aussagenlogik wird jedem Aussagensymbol ein Wahrheitswert aus der Menge $\{T, F\}$ zugeordnet. (T steht für `true` und F für `false`, viele Autoren verwenden auch die Menge $\{1, 0\}$).

Die Bedeutung einer Formel der Aussagenlogik ergibt sich dann daraus, dass man diese Zuordnung von den Aussagensymbolen auf die Formel erweitert, in dem man definiert, welcher Wahrheitswert sich beim Zusammensetzen von Formeln durch die Junktoren ergibt. Auf diese Weise definiert man den Booleschen *Operator* zum jeweiligen Junktor.

Bemerkung. Die Festlegung auf genau zwei Wahrheitswerte definiert eine *zweiwertige* Logik. Man kann auch andere Festlegungen treffen wie:

- Łukasiewicz¹-Logik mit den Wahrheitswerten $\{1, 1/2, 0\}$ oder $\{T, U, F\}$. Łukasiewicz versteht den Wert $1/2$ als „nicht bewiesen, aber auch nicht widerlegt“, manchmal wird der dritte Wert U auch als „unbekannt“ interpretiert, wie zum Beispiel in SQL.
- Zadeh²-Logik mit Wahrheitswerten im Intervall $[0, 1]$; eine Logik mit dieser Semantik wird auch Fuzzy-Logik genannt.

Sei $\mathbb{B} = \{T, F\}$. Die Junktoren $\neg, \wedge, \vee, \rightarrow$ können als *Operatoren* auf der Menge \mathbb{B} , also als Boolesche Operatoren aufgefasst werden, in dem man die Verknüpfungstafeln wie folgt definiert:

¹ Jan Łukasiewicz (1878–1956), polnischer Philosoph, Mathematiker und Logiker.

² Lotfi A. Zadeh (1921–2017), amerikanischer Informatiker.

\neg	φ	$\neg\varphi$	
	T	F	
	F	T	

$\neg\varphi$ genau dann true, wenn φ false.

\wedge	φ	ψ	$\varphi \wedge \psi$	
	T	T	T	
	T	F	F	$\varphi \wedge \psi$ ist genau dann true, wenn sowohl
	F	T	F	φ als auch ψ true sind.
	F	F	F	

\vee	φ	ψ	$\varphi \vee \psi$	
	T	T	T	
	T	F	T	$\varphi \vee \psi$ ist genau dann true, wenn einer
	F	T	T	der Operanden true ist. \vee ist also ein
	F	F	F	einschließendes oder.

\rightarrow	φ	ψ	$\varphi \rightarrow \psi$	
	T	T	T	
	T	F	F	$\varphi \rightarrow \psi$ ist genau dann true, wenn φ false
	F	T	T	oder ψ true ist.
	F	F	T	

Bemerkung. Der Operator \rightarrow wird auch als *materiale Implikation*³ bezeichnet. Er behauptet *keinen* kausalen Zusammenhang zwischen der linken Seite, dem *Antezedens* und der rechten Seite, der *Konsequenz* oder dem *Sukzedens*.

Es seien die Aussagen

$$\begin{aligned} P &= \text{„Die Erde umkreist die Sonne“}, \\ P' &= \text{„Die Sonne umkreist die Erde“ und} \\ Q &= \text{„6 ist Primzahl“} \end{aligned}$$

gegeben.

$P \rightarrow Q$ hat den Wahrheitswert F – wie man vielleicht erwarten würde, auch wenn kein inhaltlicher Zusammenhang zwischen den beiden Aussagen besteht.

Hingegen hat $P' \rightarrow Q$ den Wahrheitswert T – denn das Antezedens ist F. Das kann man vielleicht so interpretieren: „Wenn die Sonne um die Erde kreisen würde, dann wäre 6 eine Primzahl“ – und da ja die Sonne nicht um die Erde kreist, können wir über die 6 sagen, was wir wollen.

³ nach *Principia Mathematica* (1910) von Bertrand Russell (1872–1970), britischer Mathematiker und Logiker und Alfred North Whitehead (1861–1947), britischer Mathematiker.

Dahinter steckt ein klassisches logisches Prinzip: *ex falso quodlibet*⁴ – „aus Falschem folgt alles, was beliebt“.

Der für uns wichtigere Grund für die Verwendung der materialen Implikation ist jedoch, dass man mit dieser Definition der Implikation eine *einfache* Logik bekommt – in der zum Beispiel folgender Sachverhalt einfach ausdrückbar ist:

Die Aussage $\forall x \in \mathbb{N} : (x > 3) \rightarrow (x > 1)$ ist T.

Setze ein:

$x = 4$	$(4 > 3) \rightarrow (4 > 1)$	Ergebnis
	T T T	
$x = 2$	$(2 > 3) \rightarrow (2 > 1)$	Ergebnis
	F T T	
$x = 0$	$(0 > 3) \rightarrow (0 > 1)$	Ergebnis
	F F T	

In allen drei Fällen ist die Aussage true.

Bemerkung. Wenn die Implikation $P \rightarrow Q$ zutrifft, dann sagt man auch

- P ist eine *hinreichende* Bedingung für Q , denn wenn immer P zutrifft, dann ist auch Q wahr und
- Q ist eine *notwendige* Bedingung für P , denn wenn Q falsch ist, dann kann P nicht zutreffen.

Die materiale Implikation führt zu sogenannten Paradoxien, z.B.

- (1) $P \rightarrow (Q \rightarrow P)$ ist allgemeingültig
„Wenn P gilt, folgt P aus allem“.
- (2) $\neg P \rightarrow (P \rightarrow Q)$ ist allgemeingültig
„Wenn P nicht gilt, dann folgt alles aus P “.

⁴ eigentlich „ex falso sequitur quodlibet“.

4.1 Modell, Belegung

Sei \mathcal{P} die Menge der Aussagensymbole und \mathbb{B} die Menge der Wahrheitswerte.

Definition 4.1. Eine Abbildung $v : \mathcal{P} \rightarrow \mathbb{B}$ nennt man ein *Modell* für die Sprache der Aussagenlogik. Man nennt Modelle der Aussagenlogik auch spezifischer *Belegung*, weil durch v jedem Aussagensymbol ein Wahrheitswert zugewiesen wird.

Definition 4.2. Zu einem Modell $v : \mathcal{P} \rightarrow \mathbb{B}$ und einer Formel φ definiert man den Wahrheitswert $\llbracket \varphi \rrbracket_v \in \mathbb{B}$ der Formel induktiv durch

$$(i) \quad \llbracket P \rrbracket_v := v(P) \text{ für alle } P \in \mathcal{P}$$

$$(ii) \quad \llbracket \perp \rrbracket_v := F$$

$$(iii) \quad \llbracket (\neg \varphi) \rrbracket_v := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_v = F \\ F & \text{sonst} \end{cases}$$

$$(iv) \quad \llbracket (\varphi \wedge \psi) \rrbracket_v := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_v = T \text{ und } \llbracket \psi \rrbracket_v = T \\ F & \text{sonst} \end{cases}$$

$$(v) \quad \llbracket (\varphi \vee \psi) \rrbracket_v := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_v = T \text{ oder } \llbracket \psi \rrbracket_v = T \\ F & \text{sonst} \end{cases}$$

$$(vi) \quad \llbracket (\varphi \rightarrow \psi) \rrbracket_v := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_v = F \text{ oder } \llbracket \psi \rrbracket_v = T \\ F & \text{sonst} \end{cases}$$

Bemerkung. Sei φ eine Formel und v_1, v_2 seien Modelle mit $v_1(P) = v_2(P)$ für alle Aussagensymbole P in φ . Dann gilt:

$$\llbracket \varphi \rrbracket_{v_1} = \llbracket \varphi \rrbracket_{v_2}.$$

Man kann sich zu einer gegebenen Belegung v fragen, welchen Wahrheitswert eine Formel φ hat. Diese Frage ist einfach zu beantworten, indem man die Formel auswertet. Spannender ist die „umgekehrte“ Fragestellung: Gegeben ein Formel φ , gibt es dann eine Belegung, in der die Formel T ist, und wenn ja, welche? Diese ist die Frage nach der *Erfüllbarkeit* der Formel.

Definition 4.3 (Erfüllbarkeit). Eine Formel φ heißt *erfüllbar*, wenn es ein Modell v gibt mit $\llbracket \varphi \rrbracket_v = T$.

Definition 4.4 (Falsifizierbarkeit). Eine Formel φ heißt *falsifizierbar*, wenn es ein Modell v gibt mit $\llbracket \varphi \rrbracket_v = F$.

Definition 4.5 (Allgemeingültigkeit). Eine Formel φ heißt *allgemeingültig*, wenn für alle Modelle v gilt: $\llbracket \varphi \rrbracket_v = \top$. Man schreibt dann $\models \varphi$ und nennt φ eine *Tautologie*.

Definition 4.6 (Unerfüllbarkeit). Eine Formel φ heißt *unerfüllbar*, wenn für alle Modelle v gilt: $\llbracket \varphi \rrbracket_v = \bot$. Man schreibt dann $\not\models \varphi$ und nennt φ eine *Kontradiktion*.

Etwas salopp kann man diese Definitionen so auffassen:

- Eine Formel ist erfüllbar, wenn es eine „Welt“ gibt, in der sie wahr ist.
- Eine Formel ist falsifizierbar, wenn es eine „Welt“ gibt, in der sie nicht wahr ist.
- Eine Formel ist allgemeingültig, wenn sie in jeder möglichen „Welt“ wahr ist.
- Eine Formel ist unerfüllbar, wenn sie in keiner aller möglichen „Welt“ wahr ist.

Satz 4.1 (Dualitätsprinzip). *Eine Formel φ ist genau dann allgemeingültig, wenn $\neg\varphi$ unerfüllbar ist.*

Beweis. Sei φ allgemeingültig, d.h. für alle Belegungen v gilt $\llbracket \varphi \rrbracket_v = \top$, d.h. aber auch dass für alle v gilt $\llbracket \neg\varphi \rrbracket_v = \bot$, d.h. $\neg\varphi$ ist unerfüllbar. Und da $\varphi \equiv \neg\neg\varphi$ gilt auch die Umkehrung. \square

				erfüllbar
φ	ψ	$\neg\psi$	$\neg\varphi$	
allgemeingültig	erfüllbar, nicht allgemeingültig		unerfüllbar	
				falsifizierbar

Abbildung 4.1: Dualitätsprinzip

4.2 Wahrheitstafel

Um zu prüfen, ob eine Formel allgemeingültig oder erfüllbar ist, kann man die Wahrheitstafel verwenden. Im Prinzip listet man in der Wahrheitstafel alle möglichen Belegungen der Aussagensymbole in der Formel auf und berechnet in jeder dieser „Welten“ den Wahrheitswert der Formel. Man geht dabei so vor:

1. Man ermittelt die Menge der Aussagensymbole der Formel. Für jede mögliche Belegung eines Symbols mit einem Wahrheitswert bildet man eine Zeile der Wahrheitstafel. Hat die Formel n verschiedene Aussagensymbole, hat die Wahrheitstafel also 2^n Zeilen. Jedes Symbol bekommt eine Spalte der Wahrheitstafel.
2. Man bildet eine weitere Spalte der Wahrheitstafel, die mit der Formel beschriftet ist. In dieser Spalte trägt man den Wert der Formel für die Belegung der jeweiligen Zeile ein. Dazu geht man entsprechend des zugehörigen Syntaxbaums der Formel von unten nach oben und ermittelt sukzessive die Wahrheitswerte der Subformeln, bis man beim Wahrheitswert der Formel angelangt ist.

Satz 4.2. *Die Methode des Aufstellens der Wahrheitstafel ist ein Entscheidungsverfahren für das Erfüllbarkeitsproblem und für das Gültigkeitsproblem der Aussagenlogik.*

Beweis. Eine Formel φ ist *erfüllbar*, wenn es in der Wahrheitstafel *mindestens eine* Zeile mit dem Ergebnis T gibt.

Eine Formel φ ist *allgemeingültig*, wenn in der Wahrheitstafel *alle* Zeilen das Ergebnis T haben. \square

4.3 Semantische Äquivalenz und Substitution

Definition 4.7 (Semantische Äquivalenz). Zwei Formeln φ und ψ sind *semantisch äquivalent*, geschrieben $\varphi \equiv \psi$, wenn für alle Belegungen v gilt: $\llbracket \varphi \rrbracket_v = \llbracket \psi \rrbracket_v$.

Definition 4.8 (Logische Konsequenz). Sei Γ eine Menge von Formeln. Eine Formel φ heißt *logische Konsequenz* von Γ , geschrieben $\Gamma \vDash \varphi$, wenn für alle Modelle v gilt:

Ist $\llbracket \gamma \rrbracket_v = T$ für alle $\gamma \in \Gamma$, dann ist auch $\llbracket \varphi \rrbracket_v = T$.

Definition 4.9 (Substitution). Sei φ eine Subformel von ψ und φ' eine beliebige Formel. Dann ist $\psi[\varphi'/\varphi]$ (lese: ψ mit φ' an Stelle von φ) die Formel, die man erhält, wenn man jedes Vorkommen von φ in ψ durch φ' ersetzt (substituiert).

Satz 4.3 (Substitutionssatz). Sei φ eine Subformel von ψ und φ' eine semantisch äquivalente Formel, d.h. $\varphi \equiv \varphi'$, dann gilt:

$$\psi \equiv \psi[\varphi'/\varphi].$$

□

Wichtige semantische Äquivalenzen

Assoziativität

$$\begin{aligned}(\varphi \vee \psi) \vee \chi &\equiv \varphi \vee (\psi \vee \chi) \\(\varphi \wedge \psi) \wedge \chi &\equiv \varphi \wedge (\psi \wedge \chi)\end{aligned}$$

Kommutativität

$$\begin{aligned}\varphi \vee \psi &\equiv \psi \vee \varphi \\\varphi \wedge \psi &\equiv \psi \wedge \varphi\end{aligned}$$

Distributivität

$$\begin{aligned}\varphi \vee (\psi \wedge \chi) &\equiv (\varphi \vee \psi) \wedge (\varphi \vee \chi) \\\varphi \wedge (\psi \vee \chi) &\equiv (\varphi \wedge \psi) \vee (\varphi \wedge \chi)\end{aligned}$$

De Morgans⁵ Gesetze

$$\begin{aligned}\neg(\varphi \vee \psi) &\equiv \neg\psi \wedge \neg\varphi \\\neg(\varphi \wedge \psi) &\equiv \neg\psi \vee \neg\varphi\end{aligned}$$

Idempotenz

$$\begin{aligned}\varphi \vee \varphi &\equiv \varphi \\\varphi \wedge \varphi &\equiv \varphi\end{aligned}$$

Doppelte Negation

$$\neg\neg\varphi \equiv \varphi$$

Komplement

$$\begin{aligned}\varphi \wedge \neg\varphi &\equiv \perp \\\varphi \vee \neg\varphi &\equiv \top\end{aligned}$$

Identitätsgesetze

$$\begin{aligned}\varphi \wedge \top &\equiv \varphi \\\varphi \vee \perp &\equiv \varphi\end{aligned}$$

Absorption

$$\begin{aligned}\varphi \wedge (\varphi \vee \psi) &\equiv \varphi \\\varphi \vee (\varphi \wedge \psi) &\equiv \varphi\end{aligned}$$

Bemerkung. Betrachtet man die Menge der Formeln der Aussagenlogik und bildet die Menge der Äquivalenzklassen bezüglich der semantischen Äquivalenz (\equiv), dann sagen obige Aussagen unter anderem, dass diese Menge eine *Boolesche Algebra* ist.

⁵ Augustus De Morgan (1806–1871), englischer Mathematiker.

4.4 Boolesche Operatoren und funktionale Vollständigkeit

Die Anzahl der unären Operatoren auf \mathbb{B} ist 4, die der binären Operatoren ist 16. Allgemein gilt:

Satz 4.4. Für $n \in \mathbb{N}$ gibt es 2^{2^n} n -äre Boolesche Operatoren.

Beweis. Hat der Operator n Argumente, dann gibt es 2^n n -Tupel von möglichen verschiedenen Werten für die Argumente. Jede dieser Kombinationen kann als Ergebnis des Operators einen der beiden Wahrheitswerte T oder F haben, also gibt es 2^{2^n} Möglichkeiten. \square

Folgende Tabelle gibt alle möglichen binären Operatoren an:

x_1	x_2	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8
T	T	T	T	T	T	T	T	T	T
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

x_1	x_2	o_9	o_{10}	o_{11}	o_{12}	o_{13}	o_{14}	o_{15}	o_{16}
T	T	F	F	F	F	F	F	F	F
T	F	T	T	T	T	F	F	F	F
F	T	T	T	F	F	T	T	F	F
F	F	T	F	T	F	T	F	T	F

Bezeichnungen der Operatoren:

- o_1 Tautologie
- o_2 Disjunktion
- o_3 Replikation (umgekehrte Implikation, notwendige Bedingung)
- o_4 Präpendenz (Projektion auf den ersten Operanden)
- o_5 (Materiale) Implikation
- o_6 Postpendenz (Projektion auf den zweiten Operanden)
- o_7 Äquivalenz
- o_8 Konjunktion
- o_9 Exklusion (nicht beides, `nand`, Sheffer Stroke)
- o_{10} Kontravalenz (exklusives Oder, `xor`)

- o_{11} Postnonpedenz (Negation des zweiten Operanden)
- o_{12} Postsektion ($x_1 \wedge \neg x_2$)
- o_{13} Pränonpedenz (Negation des ersten Operanden)
- o_{14} Präsektion ($\neg x_1 \wedge x_2$)
- o_{15} Rejektion (weder noch, **nor**)
- o_{16} Antilogie (immer F)

Offenbar kann man Operatoren durch andere Operatoren ausdrücken, z.B.

$$\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$$

oder

$$\varphi \rightarrow \psi \equiv \neg \varphi \vee \psi$$

oder

$$\varphi \wedge \psi \equiv \neg(\neg \varphi \vee \neg \psi)$$

Definition 4.10. Eine Menge Boolescher Operatoren heißt *funktional vollständig*, wenn man jeden beliebigen Operator durch diese Operatoren logisch äquivalent ausdrücken kann.

Satz 4.5. Für jeden n -ären Booleschen Operator gibt es eine äquivalente Formel, die nur die Operatoren \neg und \vee hat. D.h. $\{\neg, \vee\}$ ist funktional vollständig.

Beweis. Jede Boolesche Funktion kann man durch eine Wahrheitstafel darstellen. Für eine Zeile der Wahrheitstafel, deren Belegung die Formel wahr macht, bildet man die Konjunktion der Atome, die in der Belegung wahr sind, bzw. der Negation der Atome, die in der Belegung der Zeile F sind. Für eine Zeile der Wahrheitstafel, deren Belegung für die Formel zu F auswertet, bildet man entsprechend die Negation der gebildeten Konjunktion. Die Disjunktion dieser Formeln ist äquivalent zur gegebenen Formel und hat nur die Operatoren \neg, \vee, \wedge . Also kann jede Boolesche Funktion durch \neg, \vee, \wedge dargestellt werden.

Nun brauchen wir nur noch \wedge durch \neg und \vee darstellen:

$$\varphi_1 \wedge \varphi_2 \equiv \neg(\neg \varphi_1 \vee \neg \varphi_2)$$

□

Bemerkung. Um die Darstellung der Formel als Disjunktion von Konjunktionen aus der Wahrheitstafel zu bilden, genügt es, die Zeilen anzusehen, in der die Formel T wird und daraus die Konjunktionen zu bilden.

Ebenso kann man die Formel als Konjunktion von Disjunktionen darstellen, indem man in der Wahrheitstafel die Zeilen verwendet, die für die Formel F ergeben und das Ergebnis mit der Regel von De Morgan zu einer Disjunktion von Konjunktionen umformt.

Auf diese Weise kann man aus der Wahrheitstafel einer Formel die disjunktive Normalform sowie die konjunktive Normalform bilden, siehe Kapitel 6 über Normalformen in der Aussagenlogik.

Kapitel 5

Das Beweissystem des natürlichen Schließens

Mit der Wahrheitstafel gibt es eine einfache Möglichkeit, festzustellen, ob eine Formel der Aussagenlogik erfüllbar ist. Man stellt die Wahrheitstafel auf und prüft, ob es eine Zeile mit dem Ergebnis T gibt.

Es gibt jedoch Beispiele, an denen man leicht sieht, dass diese Methode ihre Probleme hat. Nehmen wir etwa als Beispiel die folgende Formel:

$$(P_1 \wedge (P_1 \rightarrow P_2) \wedge \cdots \wedge (P_{n-1} \rightarrow P_n)) \rightarrow P_n$$

Diese Formel hat n Atome, die Wahrheitstafel also 2^n Zeilen.

Wir können jedoch auf eine andere Weise zu einer Lösung kommen. Wenn wir akzeptieren, dass die Schlussregel (man nennt sie *Modus ponens*)

$$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi}$$

erlaubt ist, dann kann man so argumentieren:

P_1 und $P_1 \rightarrow P_2$ sind gegeben, also gilt P_2 .

P_2 und $P_2 \rightarrow P_3$ sind gegeben, also gilt P_3 .

...

P_{n-1} und $P_{n-1} \rightarrow P_n$ sind gegeben, also gilt P_n .

Und aus dieser Argumentation folgt, dass es sich bei der gegebenen Formel um eine Tautologie handelt.

In diesem Beispiel haben wir eine Schlussregel angewandt. Wichtig ist dabei zu sehen, dass sich damit die Perspektive gewandelt hat. In der Ermittlung der Wahrheitstafel wird die *Semantik* der Aussagenlogik verwendet: für jedes mögliche Modell, jede mögliche Belegung wird der Wahrheitswert der Formel ermittelt. In der Argumentation mit der obigen Schlussregel wird kein Bezug auf die Semantik genommen, sondern die Schlussregel wird verwendet, um Transformationen an den Symbolen vorzunehmen, die die Formel ausmachen.

Sei $\Gamma = \{\gamma_1, \gamma_2, \dots, \gamma_n\}$ eine Menge von Formeln und φ eine Formel, dann gibt es zwei Sichten auf die Frage, ob φ aus Γ folgt:

- *Semantische Sicht*

$$\Gamma \vDash \varphi$$

In allen Modellen (allen „möglichen Welten“), in denen $\gamma_1, \gamma_2, \dots, \gamma_n$ gelten, ist auch φ wahr.

- *Syntaktische Sicht*

$$\Gamma \vdash \varphi$$

Man kann aus den gegebenen Formeln $\gamma_1, \gamma_2, \dots, \gamma_n$ die Formel φ herleiten, indem man ausschließlich die Regeln des Beweissystems verwendet.

Es gibt für die Aussagenlogik (und für die Prädikatenlogik) verschiedene solcher Beweissysteme. Wir werden uns in der Veranstaltung mit dem *natürlichen Schließen* (auch *natürliche Deduktion*) befassen.

Das Kalkül des natürlichen Schließen wurde 1934 von Gerhard Gentzen¹ und unabhängig von ihm von Stanisław Jaśkowski² entwickelt.

Die Bezeichnung „Natürliches Schließen“ röhrt daher, dass die Regeln des Kalküls das „natürliche“ Argumentieren von Mathematikern formalisieren.

¹ Gerhard Gentzen (1909–1945), deutscher Mathematiker und Logiker [Gen35].

² Stanisław Jaśkowski (1906–1966), polnischer Logiker.

„Mein erster Gesichtspunkt war folgender: Die Formalisierung des logischen Schließens, wie sie insbesondere durch Frege, Russell und Hilbert entwickelt worden ist, entfernt sich ziemlich weit von der Art des Schließens, wie sie in Wirklichkeit bei mathematischen Beweisen geübt wird. [...] Ich wollte nun zunächst einmal einen Formalismus aufstellen, der dem wirklichen Schließen möglichst nahekommt. So ergab sich ein ‚Kalkül des natürlichen Schließens‘“ [Gen35, S. 176].

5.1 Schlussregeln

Für die Herleitung von Formeln gibt es im natürlichen Schließen pro logischem Symbol zwei Regeln:

- eine, die das Symbol einführt (*Introduction*, abgekürzt durch i) und
- eine zweite, die das Symbol entfernt (*Elimination* auch: Auflösung), abgekürzt durch e).

Jede Regel gibt an, was *gegeben* sein muss (oberhalb des Strichs), damit die Umformung gemacht werden darf, also was sich aus dem Gegebenen *ergibt* (unterhalb des Strichs).

5.1.1 Konjunktion

Die Regeln für die Konjunktion sind:

	<i>Einführung</i>	<i>Elimination</i>
\wedge	$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \text{ ai}$	$\frac{\varphi \wedge \psi}{\varphi} \text{ ae}_1 \quad \frac{\varphi \wedge \psi}{\psi} \text{ ae}_2$

Die Konjunktion kann man einführen, wenn man Herleitungen für die beiden Formeln der Konjunktion bereits hat.

Für die Elimination der Konjunktion gibt es zwei Subregeln: Eine Herleitung der Gesamtformel der Konjunktion kann man sowohl als Herleitung der linken Teilformel als auch der rechten Teilformel nehmen.

5.1.2 Disjunktion

Die Regeln für die Disjunktion sind:

	<i>Einführung</i>	<i>Elimination</i>
\vee	$\frac{\varphi}{\varphi \vee \psi} \quad \text{vi}_1$ $\frac{\psi}{\varphi \vee \psi} \quad \text{vi}_2$	$\frac{\begin{array}{ c c } \hline \varphi & \psi \\ \vdots & \vdots \\ \chi & \chi \\ \hline \end{array}}{\chi} \quad \text{ve}$

Wenn man eine Herleitung für φ hat, hat man auch eine Herleitung für $\varphi \vee \psi$, ebenso darf man die Herleitung von ψ als Beweis für $\varphi \vee \psi$ nehmen.

Will man die Disjunktion entfernen und dabei χ herleiten, muss man für jede Teilformel der Disjunktion eine Herleitung von χ finden. Diese Regel entspricht also der Beweistechnik der Fallunterscheidung.

5.1.3 Implikation

Die Regeln für die Implikation sind:

	<i>Einführung</i>	<i>Elimination</i>
\rightarrow	$\frac{\begin{array}{ c } \hline \varphi \\ \vdots \\ \psi \\ \hline \end{array}}{\varphi \rightarrow \psi} \quad \rightarrow i$	$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi} \quad \rightarrow e, MP$

Die Implikation leitet man her, indem man die Hypothese als gegeben annimmt und dann daraus die Folgerung herleitet. In der Regel wird in der Box oberhalb des Strichs angegeben, dass φ nur *innerhalb* der Box als gegeben angenommen werden darf. Die senkrechten Punkte : markieren die Beweisverpflichtung, nämlich dass sie durch einen Beweis ersetzt werden müssen, der ψ aus φ herleitet.

Die Implikation kann man entfernen, wenn man die Hypothese φ bewiesen hat und ebenso, dass $\varphi \rightarrow \psi$ gilt. Dann hat man ψ bewiesen. Diese Schlussfigur ist schon seit der Antike geläufig und wird als *Modus ponens* bezeichnet, deshalb auch die Abkürzung MP.

5.1.4 Negation

	<i>Einführung</i>	<i>Elimination</i>
\neg	$\begin{array}{c} \varphi \\ \vdots \\ \perp \end{array}$ $\neg\varphi$	$\frac{\varphi \quad \neg\varphi}{\perp}$ $\neg e$

Will man beweisen, dass $\neg\varphi$ gilt — also die Negation einführen —, nimmt man an, dass φ bewiesen ist und führt diesen Beweis dann fort, bis man den Widerspruch \perp hergeleitet hat. Daraus ergibt sich, dass $\neg\varphi$ bewiesen ist.

Die Negation kann man auflösen, wenn man zu $\neg\varphi$ auch φ gegeben hat, denn dann folgt daraus der Widerspruch.

5.1.5 Widerspruchsbeweis, EFQ

	<i>Einführung</i>	<i>Elimination</i>
RAA, \perp	$\begin{array}{c} \neg\varphi \\ \vdots \\ \perp \end{array}$ RAA	$\frac{\perp}{\varphi}$ $\perp e, EFQ$

Man kann eine Formel φ in der klassischen Logik auch durch einen Widerspruchsbeweis herleiten³. Man nimmt das Gegenteil von φ an und führt diese Annahme zum Widerspruch. Die Regel sagt dann, dass nach diesem Widerspruchsbeweis φ bewiesen ist.

Dass aus dem Widerspruch jede beliebige Aussage folgt, wird auch als *Ex falso quodlibet* oder genauer *Ex falso sequitur quodlibet* bezeichnet.

5.2 Beispiele für das natürliche Schließen

5.2.1 Gentzens Beispiel

Als erstes Beispiel nehmen wir ein Beispiel aus der Arbeit von Gerhard Gentzen, mit der er das natürliche Schließen motiviert [Gen35].

³ In der *intuitionistischen* Logik ist diese Schlussregel nicht erlaubt.

Bewiesen werden soll die Formel

$$(X \vee (Y \wedge Z)) \rightarrow ((X \vee Y) \wedge (X \vee Z))$$

Im folgenden Beweis geben die Angaben rechts die jeweils verwendete Regel an sowie die Zeile (oder Zeilen), auf die sie angewandt wurden.

1.	$(X \vee (Y \wedge Z))$	angenommen
2.	X	angenommen
3.	$(X \vee Y)$	$\vee i_1$ 2
4.	$(X \vee Z)$	$\vee i_1$ 2
5.	$((X \vee Y) \wedge (X \vee Z))$	$\wedge i$ 3, 4
6.	$(Y \wedge Z)$	angenommen
7.	Y	$\wedge e_1$ 6
8.	$(X \vee Y)$	$\vee i_2$ 7
9.	Z	$\wedge e_2$ 6
10.	$(X \vee Z)$	$\vee i_2$ 9
11.	$((X \vee Y) \wedge (X \vee Z))$	$\wedge i$ 8, 10
12.	$((X \vee Y) \wedge (X \vee Z))$	$\wedge e$ 1, 2-5, 6- 11
13.	$(X \vee (Y \wedge Z)) \rightarrow ((X \vee Y) \wedge (X \vee Z))$	$\rightarrow i$ 1-12

5.2.2 Abgeleitete Regeln der Aussagenlogik

Hat man eine Herleitung $\varphi \vdash \psi$, dann kann man sie in anderen Herleitungen wie eine Regel verwenden. Zwar führen wir den Beweis im natürlichen Schließen mit bestimmten Symbolen aus, da jedoch der Beweis selbst ganz unabhängig von der Wahl der speziellen Symbole ist, kann sein Ergebnis selbst wie eine Regel verwendet werden.

Auf dem Merkblatt für die Regeln des natürlichen Schließens <https://esb-dev.github.io/mat/rules.pdf> sind vier solcher abgeleiteter Regeln aufgeführt, die im Folgenden bewiesen werden.

$$\frac{\varphi}{\neg\neg\varphi} \quad \neg\neg i$$

$$\frac{\neg\neg\varphi}{\varphi} \quad \neg\neg e$$

$$\frac{\varphi \rightarrow \psi \quad \neg\psi}{\neg\varphi} \quad MT$$

$$\frac{}{\varphi \vee \neg\varphi} \quad TND$$

Beweis für die Einführung der doppelten Negation:

1. P gegeben
2. $\neg P$ angenommen
3. \perp $\neg e$ 2, 1
4. $\neg\neg P$ $\neg i$ 2-3

Beweis für die Elimination der doppelten Negation:

1. $\neg\neg P$ gegeben
2. $\neg P$ angenommen
3. \perp $\neg e$ 1, 2
4. P RAA 2-3

Beweis für Modus Tollens

1. $P \rightarrow Q$ gegeben
2. $\neg Q$ gegeben
3. P angenommen
4. Q $\rightarrow e$ 1, 3
5. \perp $\neg e$ 2, 4
6. $\neg P$ $\neg i$ 3-5

Beweis für *Tertium Non Datur*

1. $\neg(P \vee \neg P)$ angenommen
2. P angenommen
3. $P \vee \neg P$ $\vee i_1$ 2
4. \perp $\neg e$ 1, 3
5. $\neg P$ $\neg i$ 2-4
6. $P \vee \neg P$ $\vee i_2$ 5
7. \perp $\neg e$ 1, 6
8. $P \vee \neg P$ RAA 1-7

Die Beweise für diese abgeleiteten Regeln haben wir mit *konkreten* Formeln, in diesem Fall Primformeln wie P und Q , durchgeführt. Sie als Regel zu verwenden bedeutet jedoch, dass sie für *jede* beliebige Formel gelten, weshalb ja auch in der Formulierung der Regel die Symbole der Metasprache φ und ψ verwendet wurden. Da aber in den Beweisen selbst

keinerlei spezifische Eigenschaften von P oder Q verwendet wurden, können wir die Beweise für beliebige Formeln abstrahieren — und somit wie Regeln verwenden.

Dies ist in der Logic WorkBench generell der Fall: ein im natürlichen Schließen geführter Beweis kann gespeichert werden (mit der Funktion `export` und dann in anderen Beweisen verwendet werden (siehe [Wiki zum Natürlichen Schließen in lwb](#)).

5.3 Beweisstrategien

In diesem Abschnitt wird an einem Beispiel erläutert, welche Strategien man beim Entwickeln einer Herleitung im natürlichen Schließen verwenden kann.

Als Beispiel nehmen wir:

$$P \rightarrow (Q \vee R) \vdash Q \vee (\neg P \vee R)$$

Der erste Schritt besteht immer darin, dass man die Voraussetzungen als gegeben hinschreibt und dann eine Lücke lässt, der das zu zeigende Ziel folgt. Die Lücke stellt die Beweisverpflichtung dar.

In unserem Beispiel führt dieser erste Schritt zu:

1. $P \rightarrow (Q \vee R)$ gegeben
2. :
3. $Q \vee (\neg P \vee R)$

Im weiteren Vorgehen untersucht man, welche Regeln man anwenden kann. Dazu gibt es zwei Möglichkeiten:

- Bei den Zeilen oberhalb der Beweisverpflichtung gibt es eine Formel, für deren Hauptjunktor eine Regel zur *Elimination* anwendbar ist. Dann kann man diese Regel *vorwärts* verwenden und erhält eine neue Zeile oberhalb der Beweisverpflichtung — oder man schließt den Beweis ab.
- Bei den Zeilen unterhalb der Beweisverpflichtung gibt es eine Formel, für deren Hauptjunktor eine Regel zur *Einführung* angewandt werden kann. Dann kann man diese Regel *rückwärts* anwenden und erhält eine neue Zeile (oder einen Block).

In unserem Beispiel ist der Hauptjunktor der Voraussetzung die Implikation. Um sie auflösen zu können, haben wir als Regel den Modus Ponens, der allerdings nur angewandt werden kann, wenn nicht nur $P \rightarrow (Q \vee R)$ gegeben ist, sondern auch P .

Betrachten wir also das Ziel. Dort ist der Hauptjunktor die Disjunktion. Diese können wir einführen, wenn eine der Seiten gegeben ist. Das ist aber auch nicht der Fall.

In dieser Situation muss man zu anderen Waffen greifen. Wir können immer die Regel TND vorwärts und die Regel RAA rückwärts anwenden.

In unserem Beispiel scheint TND eine gute Wahl zu sein.

Dann haben wir folgende Situation:

1. $P \rightarrow (Q \vee R)$ gegeben
2. $P \vee \neg P$ TND
3. :
4. $Q \vee (\neg P \vee R)$

In den folgenden Schritten hilft das Anwenden der Regeln für die Elimination von Junktoren bzw. der Einführung von Junktoren.

Wir verwenden die Regel zur Elimination der Disjunktion:

1. $P \rightarrow (Q \vee R)$ gegeben
2. $P \vee \neg P$ TND
3.

P	angenommen
:	
$Q \vee (\neg P \vee R)$	
4.

$\neg P$	angenommen
:	
$Q \vee (\neg P \vee R)$	
5. $Q \vee (\neg P \vee R)$ $\vee e$

Es ist jetzt nicht mehr schwierig, die beiden Beweisverpflichtungen zu erfüllen:

1.	$P \rightarrow (Q \vee R)$	gegeben
2.	$P \vee \neg P$	TND
3.	P	angenommen
4.	$Q \vee R$	MP 1, 3
5.	Q	angenommen
6.	$Q \vee (\neg P \vee R)$	$\vee i_1 5$
7.	R	angenommen
8.	$\neg P \vee R$	$\vee i_2 7$
9.	$Q \vee (\neg P \vee R)$	$\vee i_2 8$
10.	$Q \vee (\neg P \vee R)$	$\vee e 4, 5-6, 7-9$
11.	$\neg P$	angenommen
12.	$\neg P \vee R$	$\vee i_1 11$
13.	$Q \vee (\neg P \vee R)$	$\vee i_2 12$
14.	$Q \vee (\neg P \vee R)$	$\vee e 2, 3-10, 11-13$

Man mag versucht sein, die genannten Strategie *schematisch* anzuwenden und einfach auszuprobieren, welche Regel anwendbar ist. Dies kann in einfachen Fällen zu einer Herleitung führen, trägt jedoch wenig zum *Verständnis* des bewiesenen Sachverhalts bei. Deshalb zum Schluß die

Goldene Regel

Man muss sich die Aussage klarmachen, die man beweisen möchte und einen Beweis formulieren, ohne direkt an die Regeln zu denken, sondern eher wie die *Idee* des Beweises aussehen kann. Erst dann setzt man diese Idee in die Anwendung der Regeln um.

Werkzeuge wie die Logic Workbench eignen sich dann dafür zu überprüfen, ob die Umsetzung der Idee in einzelne Schritte korrekt durchgeführt wurde. Man vermeidet damit Flüchtigkeitsfehler.

5.4 Eigenschaften der Herleitbarkeit \vdash

In diesem Abschnitt betrachten wir Eigenschaften der Herleitbarkeit \vdash mittels des natürlichen Schließens.

Im Folgenden seien stets Γ, Γ' , Δ Mengen von Formeln der Aussagenlogik sowie φ und ψ einzelne Formeln der Aussagenlogik.

Lemma 5.1 (Monotonie).

$$\Gamma \vdash \varphi \Rightarrow \Gamma \cup \Delta \vdash \varphi$$

Beweis. Monotonie ist eine offensichtliche Eigenschaft von Herleitungen: Zu einer gegebenen Herleitung von φ aus Γ kann man weitere Voraussetzungen nach Belieben hinzufügen, sie werden ja für die Herleitung gar nicht notwendigerweise benötigt. \square

Lemma 5.2.

- (a) $\varphi \in \Gamma \Rightarrow \Gamma \vdash \varphi$
- (b) $\Gamma \vdash \varphi$ und $\Gamma' \vdash \psi \Rightarrow \Gamma \cup \Gamma' \vdash \varphi \wedge \psi$
- (c) $\Gamma \vdash \varphi \wedge \psi \Rightarrow \Gamma \vdash \varphi$ und $\Gamma \vdash \psi$
- (d) $\Gamma \cup \{\varphi\} \vdash \psi \Rightarrow \Gamma \vdash \varphi \rightarrow \psi$
- (e) $\Gamma \vdash \varphi$ und $\Gamma' \vdash \varphi \rightarrow \psi \Rightarrow \Gamma \cup \Gamma' \vdash \psi$
- (f) $\Gamma \vdash \perp \Rightarrow \Gamma \vdash \varphi$
- (g) $\Gamma \cup \{\neg\varphi\} \vdash \perp \Rightarrow \Gamma \vdash \varphi$

Beweis.

- (a) Wenn $\varphi \in \Gamma$ gilt, besteht die Herleitung von φ trivialerweise aus der Wiederholung von φ .
- (b) Wegen Monotonie gilt $\Gamma \vdash \varphi \Rightarrow \Gamma \cup \Gamma' \vdash \varphi$ sowie $\Gamma' \vdash \psi \Rightarrow \Gamma \cup \Gamma' \vdash \psi$, also wegen der Regel $\wedge i$ auch $\Gamma \cup \Gamma' \vdash \varphi \wedge \psi$.
- (c) Wenn man aus der Herleitung von $\varphi \wedge \psi$ mit der Regel $\wedge e_1$ bzw. der Regel $\wedge e_2$ die Konjunktion eliminiert, erhält man $\Gamma \vdash \varphi$ bzw. $\Gamma \vdash \psi$.
- (d) Die Aussage gilt wegen der Regel $\rightarrow i$.
- (e) Wegen Monotonie gilt unter den gegebenen Voraussetzung $\Gamma \cup \Gamma' \vdash \varphi$ sowie $\Gamma \cup \Gamma' \vdash \varphi \rightarrow \psi$, also $\Gamma \cup \Gamma' \vdash \psi$ wegen der Regel $\rightarrow e$, dem Modus Ponens.
- (f) Die Aussage gilt wegen der Regel EFQ.
- (g) Eine Herleitung für $\Gamma \vdash \varphi$ ergibt sich durch die Regel RAA, nach deren Anwendung man die gegebene Herleitung einsetzt und den Widerspruch erhält.

\square

Satz 5.1 (Endlichkeitssatz). *Gilt $\Gamma \vdash \varphi$, dann gibt es eine endliche Teilmenge $\Gamma' \subseteq \Gamma$, für die $\Gamma' \vdash \varphi$ gilt.*

Beweis. Wenn $\varphi \in \Gamma$ ist, dann reicht $\Gamma' = \{\varphi\}$ für die Herleitung aus. Für die Induktion über die Länge von Herleitungen setzen wir voraus, dass die Aussage nun für alle Herleitungen der Länge $n - 1$ gelte.

Für eine Herleitung der Länge n , betrachten wir die letzte angewandte Regel. Sie braucht nur endlich viele vorausgesetzte Aussagen, also folgt die Behauptung durch natürliche Induktion. \square

Definition 5.1. (Konsistenz) Eine Menge Γ von Formeln der Aussagenlogik ist *konsistent*, wenn $\Gamma \not\vdash \perp$, *inkonsistent* andernfalls.

Lemma 5.3. Folgende Aussagen sind äquivalent:

- (i) Γ ist konsistent
- (ii) Es gibt kein φ mit $\Gamma \vdash \varphi$ und $\Gamma \vdash \neg\varphi$
- (iii) Es gibt mindestens eine Formel φ mit $\Gamma \vdash \neg\varphi$

Das Lemma lässt sich auch so formulieren:

Folgende Aussagen sind äquivalent:

- (i') Γ ist inkonsistent
- (ii') Es gibt ein φ mit $\Gamma \vdash \varphi$ und $\Gamma \vdash \neg\varphi$
- (iii') Für alle φ gilt $\Gamma \vdash \varphi$

Beweis.

- (i') \Rightarrow (iii') Sei φ eine beliebige Formel, da Γ inkonsistent ist, gilt $\Gamma \vdash \varphi$ nach Lemma 5.2 (f).
- (iii') \Rightarrow (ii') Da alle Formeln herleitbar sind, gibt es ein φ mit $\Gamma \vdash \varphi$ sowie $\Gamma \vdash \neg\varphi$.
- (ii') \Rightarrow (i') Es gibt nach Voraussetzung ein φ mit $\Gamma \vdash \varphi$ und $\Gamma \vdash \neg\varphi$, d.h. $\Gamma \vdash \varphi \wedge \neg\varphi$, also wegen der Regel $\neg e$ auch $\Gamma \vdash \perp$.

\square

Lemma 5.4.

- (i) $\Gamma \vdash \varphi \Leftrightarrow \Gamma \cup \{\neg\varphi\} \vdash \perp$
- (ii) $\Gamma \not\vdash \varphi \Leftrightarrow \Gamma \cup \{\neg\varphi\} \not\vdash \perp$, d.h. $\Gamma \cup \{\neg\varphi\}$ ist konsistent.
- (iii) $\Gamma \vdash \neg\varphi \Leftrightarrow \Gamma \cup \{\varphi\} \vdash \perp$

Beweis.

- (i) \Rightarrow) Wenn $\Gamma \vdash \varphi$ gilt und zusätzlich die Voraussetzung $\neg\varphi$, kann man mit der Regel $\neg e$ den Widerspruch herleiten.
 \Leftarrow) siehe Lemma 5.2 (g).

- (ii) ergibt sich unmittelbar aus (i).
- (iii) \Rightarrow) Wenn $\Gamma \vdash \neg\varphi$ gilt und zusätzlich die Voraussetzung φ , kann man mit der Regel $\neg e$ den Widerspruch herleiten.
- \Leftarrow) Eine Herleitung für $\Gamma \vdash \neg\varphi$ ergibt sich durch die Regel $\neg i$, nach deren Anwendung man die gegebene Herleitung einsetzt und den Widerspruch erhält.

□

Definition 5.2. (Maximal konsistente Formelmenge) Eine Formelmenge Γ ist *maximal konsistent*, wenn sie konsistent ist, aber jede echte Obermenge $\Gamma' \supset \Gamma$ inkonsistent ist.

Maximal konsistente Mengen sind im Prinzip folgende Mengen von Aussagen: Gegeben sei eine Belegung v und $\Gamma = \{\varphi \mid \llbracket \varphi \rrbracket_v = T\}$, dann ist diese Menge konsistent und sogar maximal konsistent.

Lemma 5.5. Eine maximal konsistente Menge Γ ist abgeschlossen bezüglich Herleitbarkeit, d.h.

$$\Gamma \vdash \varphi \Rightarrow \varphi \in \Gamma \text{ für alle } \varphi.$$

Beweis. Es gelte für eine maximal konsistente Menge Γ und eine Formel $\varphi: \Gamma \vdash \varphi$. Angenommen $\varphi \notin \Gamma$, dann ist $\Gamma \cup \{\varphi\}$ inkonsistent, weil Γ maximal konsistent ist. Also gilt nach Lemma 5.4 $\Gamma \vdash \neg\varphi$. D.h. es gibt ein φ mit $\Gamma \vdash \varphi$ und $\Gamma \vdash \neg\varphi$, d.h. Γ ist inkonsistent. □

Lemma 5.6. Eine maximal konsistente Menge Γ ist negationstreu, d.h. für eine beliebige Formel φ gilt

$$\text{entweder } \Gamma \vdash \varphi \text{ oder } \Gamma \vdash \neg\varphi.$$

Beweis. Gilt $\Gamma \vdash \varphi$, dann kann wegen der Konsistenz von Γ nicht $\Gamma \vdash \neg\varphi$ gelten.

Gilt $\Gamma \not\vdash \varphi$, dann ist $\Gamma \cup \{\neg\varphi\}$ konsistent (Lemma 5.2 (g)). Da Γ maximal konsistent ist, folgt dann $\neg\varphi \in \Gamma$, und somit $\Gamma \vdash \neg\varphi$. □

5.5 Vollständigkeit des natürlichen Schließens

Mit dem Beweissystem des natürlichen Schließens haben wir eine weiteren Zugang zur Aussagenlogik neben der semantischen Sicht, wie bereits oben erwähnt:

Sei $\Gamma = \{\gamma_1, \gamma_2, \dots, \gamma_n\}$ eine Menge von Formeln und φ eine Formel, dann gibt es zwei Sichten auf die Frage, ob φ aus Γ folgt:

- In der *semantische Sicht* betrachten wir die Frage der *logischen Konsequenz*: $\Gamma \vDash \varphi$.
- In der *syntaktischen Perspektive* stellt sich die Frage, ob man für eine Formel eine *Herleitung* findet: $\Gamma \vdash \varphi$.

Nun stellt sich natürlich die Frage, ob die beiden Zugänge gleichwertig sind. Im Grunde handelt es sich im zwei Fragen:

- Ist das Beweissystem *korrekt*? Anders gesagt: wir können aus gegebenen Voraussetzungen nur wahre Schlüsse ziehen.

$$\Gamma \vdash \varphi \Rightarrow \Gamma \vDash \varphi$$

- Ist das Beweissystem *vollständig*? Ist es möglich mit den Regeln auch jede in der Semantik zutreffende Schlussfolgerung herzuleiten, also die Umkehrung:

$$\Gamma \vDash \varphi \Rightarrow \Gamma \vdash \varphi$$

Die erste Frage, die nach der *Korrektheit* des natürlichen Schließens, ist die einfachere Angelegenheit, schließlich haben wir die Regeln ja so ausgewählt, dass sie Wahrheit erhalten. Die zweite Frage, die nach der *Vollständigkeit* des Beweissystems, ist schwieriger einzusehen, weil ja nicht auf Anhieb erkennbar ist, wie wir aus der semantischen Sicht auf die Existenz einer Herleitung schließen können, oder sie sogar konstruieren können.

Es sei noch bemerkt, dass es neben dem natürlichen Schließen noch andere Beweissysteme für die Aussagenlogik gibt, zum Beispiel das Hilbert-Kalkül, semantische Tableaus, Gentzens Sequenzenkalkül oder die Resolution. Für alle diese Beweissysteme kann man die Vollständigkeit zeigen. Vollständigkeit ist also eine Eigenschaft der Logik selbst: es gibt die Möglichkeit ein Kalkül zu finden, mit dem durch rein symbolische Manipulation von Formeln jede semantisch wahre Aussage gezeigt werden kann.⁴

⁴ “However, completeness should be understood not as a statement about these specific rules, but as a statement about the logic itself. Completeness asserts the existence of a list of rules that allows us to deduce every consequence from any set of formulas of the logic.” [Hed04, S.44]

Zum Begriff *Vollständigkeit* Wir hatten in der Einleitung 1.2 schon den Begriff der Vollständigkeit und auch in 4.4 war von „funktionaler Vollständigkeit“ die Rede. Man muss diese drei Bedeutungen auseinanderhalten⁵:

- Eine Menge von Booleschen Operatoren heißt *funktional vollständig*, wenn man jede Boolesche Funktion $f : \mathbb{B}^n \rightarrow \mathbb{B}, n \geq 1$ durch diese Operatoren darstellen kann, siehe 4.4.
- Eine durch ein Axiomensystem definierte Theorie heißt *vollständig*, wenn jede geschlossene Formel oder ihre Negation in der Theorie enthalten ist, siehe 1.2.
- Ein Beweissystem, ein Kalkül für eine Logik heißt *vollständig*, wenn jedes wahre Theorem auch mit den Regeln des Kalküls hergeleitet werden kann. Dieser Begriff von Vollständigkeit ist das Thema in diesem Abschnitt. Man sieht leicht, dass damit nicht Vollständigkeit im Sinne einer vollständigen Theorie gemeint ist, denn die Formel P kann in der Aussagenlogik offensichtlich weder als wahr noch als falsch bewiesen werden.

5.5.1 Korrektheit des natürlichen Schließens für die Aussagenlogik

Satz 5.2 (Korrektheit des natürlichen Schließens).

$$\Gamma \vdash \varphi \Rightarrow \Gamma \vDash \varphi$$

(Informeller) Beweis:

Zunächst muss man präzise sagen, was in Herleitungen erlaubt ist.

Dazu muss man unterscheiden zwischen Regeln, die man einfach dadurch anwenden kann, indem man syntaktisch die Ausdrücke über dem Strich der Regel durch den Ausdruck unter dem Strich ersetzt. Ein Beispiel ist die Einführung von \wedge aus zwei gegebenen Formeln.

Andere Regeln sind so aufgebaut, dass sie selbst wieder eine Beweisverpflichtung enthalten, wie etwa die Einführung von $\varphi \rightarrow \psi$. Für diesen Schritt wird angenommen, dass φ bereits hergeleitet wurde und man zeigt unter dieser Annahme, dass man dann ψ herleiten kann. In diesem Fall darf die Annahme φ nicht außerhalb des Bereichs der Beweisverpflichtung verwendet werden.

⁵ Der Wikipedia-Artikel über [Vollständigkeit in der Logik](#) beschreibt die drei Bedeutungen des Begriffs recht gut.

Um es mit Richard Bornat auszudrücken [Bor05, Definition 5.2, Seite 56] auszudrücken:

In a box-and-line proof

1. every line must be justified either as a premise or by use of a rule appealing to *previous* lines or boxes;
2. if an appealed-to line is inside a box, then that box must also enclose the justified line.

Wenn eine Herleitung diese Bedingungen erfüllt, dann wird sie wahre Aussagen in wahre Aussagen überführen, wenn jede der Regeln des natürlichen Schließens das tut. Man muss also für jede Regel überprüfen, ob sie Wahrheit erhält.

Zwei Beispiele für die Argumentation, die Korrektheit der restlichen Regeln kann man ähnlich überprüfen:

Die Regel $\frac{\varphi \quad \psi}{\varphi \wedge \psi} \wedge i$ erhält Wahrheit, denn sind die Voraussetzungen wahr, dann auch die Folgerung. Das ist gerade die semantische Definition von \wedge .

$$\boxed{\begin{array}{c} \varphi \\ \vdots \\ \psi \end{array}}$$

Auch die Regel $\frac{\varphi \rightarrow \psi}{\varphi \rightarrow \psi} \rightarrow i$ erhält Wahrheit: Wenn wir annehmen, dass $\llbracket \varphi \rrbracket = T$ und es gelingt unter Anwendung unserer wahrheitserhaltender Regeln zu zeigen, dass dann auch ψ wahr ist, dann bedeutet das, dass $\llbracket \varphi \rightarrow \psi \rrbracket = T$ ist, wie man an der Wahrheitstafel von \rightarrow sieht. \square

5.5.2 Vollständigkeit des natürlichen Schließens für die Aussagenlogik

Es geht darum, dass jede wahre Aussage auch hergeleitet werden kann. Wir haben also alle Regen, die dafür erforderlich sind. Ordentlicher ausgedrückt:

Satz 5.3 (Vollständigkeit des natürlichen Schließens).

$$\Gamma \vDash \varphi \Rightarrow \Gamma \vdash \varphi$$

Man kann diesen Satz durch einen Widerspruchsbeweis zeigen. In Büchern über mathematische Logik wird gerne diese Art des Beweises verwendet.

Eine andere Möglichkeit des Beweises besteht darin, dass man aus der gegebenen wahren Aussage eine Herleitung des natürlichen Schließens

konstruiert. Man zeigt also, dass man ein Programm schreiben kann, das die Herleitung ausgibt und wie es dies tun müsste. Dies ist eine Art des Beweises, die einem Informatiker naheliegt, so wird der Beweis etwa in [HR04] geführt.

Wir wollen beide Beweise kennenlernen, in diesem Abschnitt zunächst den indirekten Beweis.

Wir benötigen zwei wichtige Sätze, um den Widerspruchsbeweis führen zu können:

Satz 5.4 (Satz von Lindenbaum⁶). *Jede konsistente Menge Γ kann zu einer maximal konsistenten Menge $\Gamma^* \supseteq \Gamma$ erweitert werden.*

Beweis. Wir setzen für den Beweis voraus, dass unsere Sprache abzählbar viele Formeln enthält⁷: $\varphi_0, \varphi_1, \varphi_2, \dots$. Nun kann man eine aufsteigende Folge von Formelmengen definieren, deren Vereinigung maximal konsistent ist.

Die Konstruktion geht so:

$$\begin{aligned}\Gamma_0 &= \Gamma \\ \Gamma_{n+1} &= \begin{cases} \Gamma_n \cup \{\varphi_n\} & \text{falls } \Gamma_n \cup \{\varphi_n\} \text{ konsistent} \\ \Gamma_n \cup \{\neg\varphi_n\} & \text{andernfalls} \end{cases} \\ \Gamma^* &= \bigcup \{\Gamma_n \mid n \geq 0\}\end{aligned}$$

Es ist nun zu zeigen, dass Γ^* konsistent und maximal konsistent ist:

Jede der Formelmengen Γ_n ist konsistent, so wurden sie ja konstruiert.

Nehmen wir nun an, ihre Vereinigung Γ^* wäre nicht konsistent. Dann könnte man aus Γ^* den Widerspruch herleiten. Für diese Herleitung genügen endlich viele der Aussagen aus Γ^* , d.h. aber, dass es ein n gibt, so dass alle die für die Herleitung benötigten Aussagen in Γ_n sind, also $\Gamma_n \vdash \perp$, aber Γ_n ist konsistent. Die Annahme ist also widerlegt, Γ^* ist konsistent.

⁶ Adolf Lindenbaum (1904–1941), polnischer Logiker und Mathematiker.

⁷ Man kann den Beweis auch für überabzählbare Mengen führen, siehe [Rau08, Lemma 4.3]

Es bleibt zu zeigen, dass Γ^* maximal konsistent ist. Nehmen wir eine konsistente Menge von Aussagen $\Delta \supseteq \Gamma^*$. Ist $\varphi \in \Delta$, dann ist φ eine der Formeln φ_i , und demzufolge in Γ_{i+1} enthalten, also in Γ^* , d.h. $\Delta = \Gamma^*$. \square

Satz 5.5 (Modellexistenzsatz). *Eine konsistente Menge Γ ist erfüllbar.*

Beweis. Wir können wegen dem Satz von Lindenbaum annehmen, dass Γ eine maximale konsistente Menge ist. Wir definieren eine Belegung, indem wir auf den Primformeln festlegen

$$v(P_i) = \begin{cases} T & \text{für } P_i \in \Gamma \\ F & \text{sonst} \end{cases}$$

Nun muss man also zeigen, dass für eine beliebige Formel φ gilt:

$$v(\varphi) = T \Leftrightarrow \varphi \in \Gamma$$

Dazu kann man strukturelle Induktion über den Aufbau der Formeln machen. Da wir wissen, dass $\{\wedge, \neg\}$ funktional vollständig ist, genügt es die Induktion für diese beiden Junktoren zu machen.

1. Für Primformeln gilt die Behauptung per Definition der Belegung.
2. Für $\varphi = \psi \wedge \chi$:
 $v(\varphi) = T \Leftrightarrow v(\psi) = v(\chi) = T$ also durch die Induktionsvoraussetzung: $\psi, \chi \in \Gamma$, also auch $\varphi \in \Gamma$ wegen der Maximalität von Γ .
Umgekehrt: Maximale konsistente Mengen sind abgeschlossen bezüglich Herleitbarkeit (Lemma 5.5), d.h. $\psi \wedge \chi \in \Gamma \Leftrightarrow \psi, \chi \in \Gamma$, also $v(\varphi) = T$.
3. Für $\varphi = \neg\psi$:
 $v(\varphi) = T \Leftrightarrow v(\psi) = F$ also durch die Induktionsvoraussetzung: $\psi \notin \Gamma$ und $\varphi \in \Gamma$.
Umgekehrt: $\varphi \in \Gamma$ ergibt $\psi \notin \Gamma$, also $v(\psi) = F$, also $v(\varphi) = T$.

Aus der strukturellen Induktion folgt die Aussage, wir haben ein Modell konstruiert. \square

Nun können wir den Vollständigkeitssatz beweisen:

Beweis des Vollständigkeitssatzes. Zu zeigen ist, dass $\Gamma \vDash \varphi \Rightarrow \Gamma \vdash \varphi$ gilt. Wir nehmen das Gegenteil an, dass also zwar $\Gamma \vDash \varphi$, aber $\Gamma \not\vdash \varphi$ der Fall ist.

Wenn $\Gamma \not\vdash \varphi$, dann ist $\Gamma \cup \{\neg\varphi\}$ nach Lemma 5.4 (ii) konsistent. Nach dem Satz von Lindenbaum gibt es eine maximale konsistente Erweiterung

Γ^* für $\Gamma \cup \{\neg\varphi\}$. Nach dem Modellexistenzsatz gibt es ein Modell für Γ^* , das alle diese Formeln wahr macht. Also ist dann auch $\neg\varphi$ wahr, das bedeutet aber, dass $\Gamma \not\models \varphi$ gelten würde, was der Voraussetzung widerspricht. \square

5.5.3 Ein konstruktiver Beweis für den Vollständigkeitssatz in der Aussagenlogik

Für den konstruktiven Beweis setzen wir voraus, dass die Menge Γ endlich ist und es gilt $\Gamma \vDash \varphi$. Es ist dann zu zeigen, dass $\Gamma \vdash \varphi$ gilt, indem wir angeben, wie man eine Herleitung erstellen kann.

Zunächst reduzieren wir die Fragestellung auf den Fall einer allgemeingültigen Formel, d.h. auf

$$\vDash \varphi \Rightarrow \vdash \varphi$$

Mit Lemma 5.7 wird aus dem allgemeinen Fall $\Gamma \vDash \varphi$ eine allgemeingültige Formel. Wenn wir für diese zeigen, dass es eine Herleitung gibt, können wir mit Lemma 5.8 eine Herleitung $\Gamma \vdash \varphi$ finden.

Lemma 5.7. Für Aussagen $\gamma_1, \gamma_2, \dots, \gamma_n$ und φ gilt:

$$\gamma_1, \gamma_2, \dots, \gamma_n \vDash \varphi \Rightarrow \vDash (\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi)$$

Beweis. Wir betrachten den Syntaxbaum der Formel $\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi$ in Abb. 5.1.

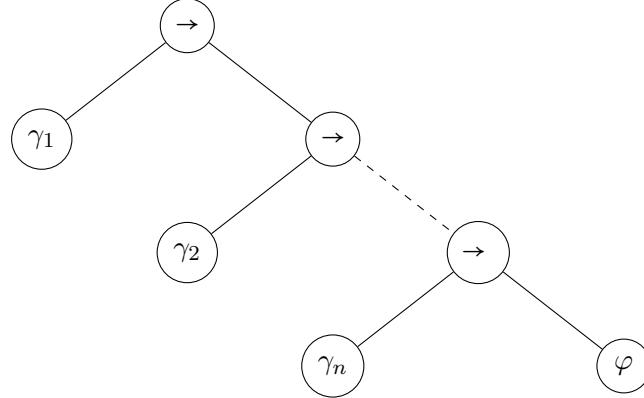


Abbildung 5.1: Syntaxbaum für Lemma 5.7

Angenommen $\#(\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi) \neq 0$, d.h. es gibt eine Belegung bei der die Aussage F ergibt. Dann muss die Wurzel des Baums F sein. Das ist nur möglich, wenn γ_1 T ist und die zweite Implikation F — und so weiter. Also ergibt sich, dass alle der γ s T sind, aber φ F. Das steht aber im Widerspruch zur Voraussetzung $\gamma_1, \gamma_2, \dots, \gamma_n \vDash \varphi$. \square

Lemma 5.8. Für Aussagen $\gamma_1, \gamma_2, \dots, \gamma_n$ und φ gilt:

$$\vdash (\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi) \Rightarrow \gamma_1, \gamma_2, \dots, \gamma_n \vdash \varphi$$

Beweis. Nach Voraussetzung gibt es eine Herleitung für $\vdash (\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi)$. Wir müssen einen Beweis konstruieren für $\gamma_1, \gamma_2, \dots, \gamma_n \vdash \varphi$.

Wir gehen so vor:

1.	γ_1	gegeben
2.	γ_2	gegeben
:	:	:
n.	γ_n	gegeben
:	$\boxed{\quad : \quad}$	Beweis aus der Voraussetzung
m.	$\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi$	
m+1.	$\gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi$	$\rightarrow e 1, m$
m+2.	$\gamma_3 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi$	$\rightarrow e 2, m+1$
:	:	:
m+n.	φ	$\rightarrow e n, m+n-1$

□

Mit diesen beiden Lemmas haben wir Aufgabe darauf reduziert, zu zeigen, dass

$$\vDash \varphi \Rightarrow \vdash \varphi$$

gilt. Die Idee des Beweise besteht darin, aus der Wahrheitstafel der Formel φ die Herleitung zu konstruieren.

Lemma 5.9. φ sei eine Formel und $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n$ seien die Literale einer Zeile der Wahrheitstafel für φ . Dann gilt:

1. Wenn der Wert der Zeile T ist, dann gibt es eine Herleitung $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n \vdash \varphi$
2. Wenn der Wert der Zeile F ist, dann gibt es eine Herleitung $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n \vdash \neg\varphi$

Beweis. Der Beweis geht durch strukturelle Induktion über den Formelaufbau:

Wenn die Formel atomar ist, etwa P , dann muss man zeigen, dass $P \vdash P$ und $\neg P \vdash \neg P$. Diese Beweise sind einfach die Übernahme der gegebenen Aussage in die Schlussfolgerung.

Wenn die Formel nicht atomar ist, dann hat sie eine (bei \neg als Hauptjunktor) oder zwei Subformeln. Nach Induktionsvoraussetzung können wir voraussetzen, dass das Lemma für diese Subformeln bereits gilt. Das bedeutet, dass wir für jeden Junktor einen Beweis für alle Fälle seiner zugeordneten Wahrheitstafel finden müssen.

Falls $\varphi = \neg\varphi_1$:

Wenn $\varphi \equiv T$ ist, brauchen wir einen Beweis $\neg\varphi_1 \vdash \neg\varphi_1$, was automatisch gilt.

Wenn $\varphi \equiv F$ ist, brauchen wir einen Beweis $\varphi_1 \vdash \neg\neg\varphi_1$:

1. φ_1 gegeben
2. $\neg\varphi_1$ angenommen
3. \perp $\neg e$ 2, 1
4. $\neg\neg\varphi_1$ $\neg i$ 2-3

Falls $\varphi = \varphi_1 \rightarrow \varphi_2$:

Wenn $\varphi \equiv T$ müssen wir drei Fälle betrachten, nämlich φ_1 ist F oder φ_2 ist T:

1. $\neg\varphi_1$ gegeben
2. φ_2 gegeben
3. φ_1 angenommen
4. φ_2 übernommen 2
5. $\varphi_1 \rightarrow \varphi_2 \rightarrow i$ 3-4

1. $\neg\varphi_1$ gegeben
2. $\neg\varphi_2$ gegeben
3. φ_1 angenommen
4. \perp $\neg e$ 1, 3
5. φ_2 EFQ 4
6. $\varphi_1 \rightarrow \varphi_2 \rightarrow i$ 3-5

1. φ_1 gegeben
2. φ_2 gegeben
3. φ_1 angenommen
4. φ_2 übernommen 2
5. $\varphi_1 \rightarrow \varphi_2 \rightarrow i$ 3-4

Bleibt für die Implikation noch der Fall, dass φ zu F auswertet:

1. φ_1 gegeben
2. $\neg\varphi_2$ gegeben
3. $\varphi_1 \rightarrow \varphi_2$ angenommen
4. φ_2 $\rightarrow e$ 3, 1
5. \perp $\neg e$ 2, 4
6. $\neg(\varphi_1 \rightarrow \varphi_2)$ $\neg i$ 3-5

Falls $\varphi = \varphi_1 \wedge \varphi_2$:

Wenn φ zu T auswertet, haben wir einen Fall zu betrachten:

1. φ_1 gegeben
2. φ_2 gegeben
3. $\varphi_1 \wedge \varphi_2$ $\wedge i$ 1, 2

Ist $\varphi \equiv F$ sind drei Fälle zu zeigen:

1. φ_1 gegeben
2. $\neg\varphi_2$ gegeben
3. $\varphi_1 \wedge \varphi_2$ angenommen
4. φ_2 $\wedge e$ 3
5. \perp $\neg e$ 2, 4
6. $\neg(\varphi_1 \wedge \varphi_2)$ $\neg i$ 3-5

Der Fall $\neg\varphi_1$ und φ_2 ist symmetrisch zum eben gezeigten Fall.

Für den Fall $\neg\varphi_1$ und $\neg\varphi_2$ kann man obigen Beweis verwenden, denn die Aussage in Zeile 1 haben wir für die Herleitung ja garnicht verwendet, sie funktioniert also auch, wenn statt φ_1 $\neg\varphi_1$ gegeben ist.

Falls $\varphi = \varphi_1 \vee \varphi_2$:

Ist $\varphi \equiv T$ haben wir drei Fälle zu betrachten:

1. φ_1 gegeben
2. φ_2 gegeben
3. $\varphi_1 \vee \varphi_2$ $\vee i_1$ 1

Diese Herleitung geht auch, wenn $\neg\varphi_2$ vorausgesetzt wird. Und nehmen wir den Fall, dass $\neg\varphi_1$ gilt, gibt es die symmetrische Herleitung mit der Regel $\vee i_2$.

Ist schließlich $\varphi \equiv F$ können wir folgende Herleitung nehmen:

1.	$\neg\varphi_1$	gegeben
2.	$\neg\varphi_2$	gegeben
3.	$\varphi_1 \vee \varphi_2$	angenommen
4.	φ_1	angenommen
5.	\perp	$\neg e 1, 4$
6.	φ_2	angenommen
7.	\perp	$\neg e 2, 6$
8.	\perp	$\vee e 3, 4-5, 6-7$
9.	$\neg(\varphi_1 \vee \varphi_2)$	$\neg i 3-8$

□

Nun können wir den konstruktiven Beweis des Vollständigkeitssatzes abschließen, in dem wir zeigen, dass

$$\vDash \varphi \Rightarrow \vdash \varphi$$

gilt.

Beweis. Da $\vDash \varphi$ gilt, hat die Formel φ eine Wahrheitstafel, in der alle Zeilen zu T auswerten. Seien P_1, P_2, \dots, P_n die Aussagensymbole in φ . Die Wahrheitstafel hat dann 2^n Zeilen.

Wir beginnen die Konstruktion der Herleitung, indem wir mit der Regel TND als erste Zeile der Herleitung $P_1 \vee \neg P_1$ einführen. Der nächste Schritt besteht dann darin, dass wir zwei Unterbeweise für die Auflösung des \vee der ersten Zeile aufmachen. In jeder dieser beiden Boxen wenden wir nun erneut die Regel TND an mit P_2 , gefolgt von $\vee e$.

Nach zwei Schritten sieht die Herleitung aus wie in Abbildung 5.2 dargestellt.

Wenn man das Verfahren abwechselnd die Regel TND und die Auflösung von \vee bis zu P_n fortsetzt, dann hat man 2^n Boxen mit Beweisverpflichtungen. Jede der Boxen entspricht genau einer Zeile der Wahrheitstafel und hat als Voraussetzungen die entsprechenden Literale \hat{P}_i .

In jeder Box können wir nach Lemma 5.9 eine Herleitung rekursiv konstruieren. Diese Herleitungen fügen wir an Stelle der Beweisverpflichtungen ein — und so erhalten wir eine Herleitung für φ . □

Bemerkung. In Lemma 5.9 und im Beweis eben wurden alle Regeln zur Einführung und Auflösung von $\wedge, \vee, \rightarrow, \neg$ verwendet, außerdem EFQ

1.	$P_1 \vee \neg P_1$	TND
2.	P_1	angenommen
3.	$P_2 \vee \neg P_2$	TND
4.	P_2	angenommen
5.	:	
6.	φ	
7.	$\neg P_2$	angenommen
8.	:	
9.	φ	
10.	φ	$\vee e$
11.	$\neg P_1$	angenommen
12.	$P_2 \vee \neg P_2$	TND
13.	P_2	angenommen
14.	:	
15.	φ	
16.	$\neg P_2$	angenommen
17.	:	
18.	φ	
19.	φ	$\vee e$
20.	φ	$\vee e$

Abbildung 5.2: Herleitung einer Tautologie

und TND. Dies zeigt, dass diese Regeln für das natürliche Schließen in der klassischen Aussagenlogik ausreichend sind.

5.5.4 Kompaktheitssatz

Der Vollständigkeitssatz hat eine wichtige Folgerung, den Endlichkeitssatz der Erfüllbarkeit oder den *Kompaktheitssatz*:

Satz 5.6 (Kompaktheitssatz). *Eine Menge Γ von Aussagen ist genau dann erfüllbar, wenn jede endliche Teilmenge von Γ erfüllbar ist.*

Beweis. Wenn Γ erfüllbar ist, dann gilt das selbstverständlich auch für jede endliche Teilmenge von Γ .

Für die umgekehrte Richtung nehmen wir an, dass Γ nicht erfüllbar ist, um einen Widerspruchsbeweis zu führen. Nach dem Vollständigkeitssatz gilt dann $\Gamma \vdash \perp$. Ein solcher Beweis kommt aber mit einer endlichen Menge von Voraussetzungen Γ' aus. Das bedeutet, dass es eine endliche Teilmenge von Γ gibt, eben Γ' , die nicht erfüllbar ist, was der Voraussetzung widerspricht. Also muss Γ erfüllbar gewesen sein. \square

Kapitel 6

Normalformen

Formeln der Aussagenlogik können äquivalent in vielen Formen ausgedrückt werden. Oft möchte man jedoch der Einfachheit halber eine bestimmte syntaktische Gestalt der Formeln voraussetzen können. Dazu betrachten wir in diesem Kapitel die konjunktive und die disjunktive Normalform für aussagenlogische Formeln.

6.1 Negationsnormalform NNF

Wir beobachten zunächst, dass man jede Formel äquivalent so umformen kann, dass sie keine Implikationen mehr enthält. Dazu verwendet man die Äquivalenz $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$.

Definition 6.1 (Literal). Ein *Literal* ist eine Primaussage P oder ihre Negation $\neg P$.

Wir formulieren einen Algorithmus für die Elimination der Implikation aus aussagenlogischen Formeln:

```
function IMPL_FREE( $\varphi$ ) {
    // pre: beliebige Formel  $\varphi$ 
    // post: äquivalente Umformung von  $\varphi$ , die kein  $\rightarrow$  mehr enthält
    case {
         $\varphi$  ist Literal:
            return  $\varphi$ ;
         $\varphi$  hat die Form  $\neg\varphi_1$ :
            return  $\neg\text{IMPL\_FREE}(\varphi_1)$ ;
         $\varphi$  hat die Form  $\varphi_1 \wedge \varphi_2$ :
            return  $\text{IMPL\_FREE}(\varphi_1) \wedge \text{IMPL\_FREE}(\varphi_2)$ ;
         $\varphi$  hat die Form  $\varphi_1 \vee \varphi_2$ :
            return  $\text{IMPL\_FREE}(\varphi_1) \vee \text{IMPL\_FREE}(\varphi_2)$ ;
         $\varphi$  hat die Form  $\varphi_1 \rightarrow \varphi_2$ :
            return  $\neg\varphi_1 \vee \text{IMPL\_FREE}(\varphi_2)$ ;
    }
}
```

```

        return  $\neg \text{IMPL\_FREE}(\varphi_1) \vee \text{IMPL\_FREE}(\varphi_2);$ 
    }
}

```

Definition 6.2 (Negationsnormalform NNF). Eine Formel φ ohne Implikation ist in der *Negationsnormalform NNF*, wenn jede Negation direkt vor einer Primaussage steht.

Beispiele

$\neg\neg\neg P$	nicht NNF	$\neg P$	NNF
$\neg(P \wedge Q)$	nicht NNF	$\neg P \vee \neg Q$	NNF
$\neg(P \vee Q)$	nicht NNF	$\neg P \wedge \neg Q$	NNF

Folgender Algorithmus bringt eine Formel in die Negationsnormalform:

```

function NNF( $\varphi$ ) {
// pre:  $\varphi$  hat keine Implikationen
// post: äquivalente Umformung von  $\varphi$  in NNF
    case {
         $\varphi$  ist Literal:
            return  $\varphi$ ;
         $\varphi$  hat die Form  $\neg\neg\varphi_1$ :
            return NNF( $\varphi_1$ );
         $\varphi$  hat die Form  $\varphi_1 \wedge \varphi_2$ :
            return NNF( $\varphi_1$ )  $\wedge$  NNF( $\varphi_2$ );
         $\varphi$  hat die Form  $\varphi_1 \vee \varphi_2$ :
            return NNF( $\varphi_1$ )  $\vee$  NNF( $\varphi_2$ );
         $\varphi$  hat die Form  $\neg(\varphi_1 \wedge \varphi_2)$ :
            return NNF( $\neg\varphi_1$ )  $\vee$  NNF( $\neg\varphi_2$ );
         $\varphi$  hat die Form  $\neg(\varphi_1 \vee \varphi_2)$ :
            return NNF( $\neg\varphi_1$ )  $\wedge$  NNF( $\neg\varphi_2$ );
    }
}

```

In der Logic Workbench lwb transformiert die Funktion `nnf` eine Formel in die Negationsnormalform:

```

(nnf '(not (not P)))
; => P

(nnf '(not (not (not P))))
; => (not P)

(nnf '(not (and P Q)))
; => (or (not P) (not Q))

```

```
(nmf '(not (or P q)))
; => (and (not P) (not q))
```

6.2 Konjunktive Normalform CNF

Definition 6.3 (Klausel). Eine Formel der Form $\varphi = \hat{P}_1 \vee \hat{P}_2 \vee \dots \vee \hat{P}_n$ mit Literalen \hat{P}_i nennt man eine *Klausel*.

Beispiele:

$P \vee \neg Q \vee \neg R$ ist eine Klausel
 $\neg P_1 \vee \neg P_2 \vee \dots \vee \neg P_n \vee Q$ ist eine Klausel,
sie ist äquivalent zu
 $P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow Q$.

Bemerkung. Oft stellt man Klauseln als Mengen von Literalen dar, wobei mit \bar{P} die Negation $\neg P$ einer Primaussage bezeichnet wird. Der Grund dafür besteht darin, dass logischen Operationen mit Formeln in konjunktiver Normalform und Klauseln mengentheoretischen Operationen auf Klauselmengen und Klauseln entsprechen.

Beispiele korrespondierend zu obigen Beispielen:

$\{P, \bar{Q}, \bar{R}\}$
 $\{\bar{P}_1, \bar{P}_2, \dots, \bar{P}_n, Q\}$

Definition 6.4 (Konjunktive Normalform CNF). Eine Formel φ ist in der *konjunktiven Normalform CNF*, wenn sie die Form $\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$ hat mit lauter Klauseln φ_i .

Wir benötigen eine Hilfsfunktion:

```
function DISTR( $\varphi_1, \varphi_2$ ) {
// pre:  $\varphi_1$  und  $\varphi_2$  sind in CNF
// post: berechnet CNF für  $\varphi_1 \vee \varphi_2$ 
    case {
         $\varphi_1$  hat die Form  $\varphi_{11} \wedge \varphi_{12}$ :
            return DISTR( $\varphi_{11}, \varphi_2$ )  $\wedge$  DISTR( $\varphi_{12}, \varphi_2$ );
         $\varphi_2$  hat die Form  $\varphi_{21} \wedge \varphi_{22}$ :
            return DISTR( $\varphi_1, \varphi_{21}$ )  $\wedge$  DISTR( $\varphi_1, \varphi_{22}$ );
        default:
            return  $\varphi_1 \vee \varphi_2$ ;
    }
}
```

```

    }
}
```

Mit Hilfe von DISTR ist es nun einfach, einen Algorithmus für das Erzeugen der CNF zu formulieren:

```

function CNF( $\varphi$ ) {
// pre:  $\varphi$  ist in NNF
// post: eine zu  $\varphi$  äquivalente Formel in CNF
  case {
     $\varphi$  ist Literal:
      return  $\varphi$ ;
     $\varphi$  hat die Form  $\varphi_1 \wedge \varphi_2$ :
      return CNF( $\varphi_1$ )  $\wedge$  CNF( $\varphi_2$ );
     $\varphi$  hat die Form  $\varphi_1 \vee \varphi_2$ :
      return DISTR(CNF( $\varphi_1$ ), CNF( $\varphi_2$ ));
  }
}
```

In der Logic Workbench lwb transformiert die Funktion `cnf` eine beliebige Formel in die konjunktive Normalform:

```

(cnfs '(impl P Q))
; => (and (or Q (not P)))

(cnfs '(and (impl P Q) R (or Q (not P))))
; => (and (or Q (not P)) (or R))

(cnfs '(and (impl P Q) (and R S) (or P (not Q))))
; => (and (or Q (not P)) (or R) (or S) (or P (not Q)))
```

Die Funktion `cnf?` prüft, ob eine Formel in der konjunktiven Normalform ist:

```

(cnfs? '(or (and P Q)))
; => false

(cnfs? (cnfs '(or (and P Q))))
; => true

(cnfs? '(and (or P Q)))
; => true

(cnfs? '(or P Q))
; => false

(cnfs? '(and (or P Q) R))
; => false
```

```
(cnf? '(and (or P Q) (or R)))
; => true
```

Die letzten vier Beispiele zeigen, dass die Logic Workbench eine spezielle Form der konjunktiven Normalform erwartet: der Hauptjunktor muss ein `and` sein und die Klauseln müssen den Junktor `or` haben, d.h. ein Atom allein genügt der Prüfung nicht. Die erwartete Syntax ist exakt jene, die die Funktion `cnf` generiert.

6.3 Disjunktive Normalform DNF

Definition 6.5 (Monom). Eine Formel der Form $\psi = \hat{P}_1 \wedge \hat{P}_2 \wedge \dots \wedge \hat{P}_n$ mit Literalen \hat{P}_i bezeichnet man als *Monom*.

Definition 6.6 (Disjunktive Normalform DNF). Eine Formel ψ ist in der *disjunktiven Normalform DNF*, wenn sie die Form $\psi_1 \vee \psi_2 \vee \dots \vee \psi_n$ hat mit lauter Monomen ψ_i .

Beobachtung: („Dualität“ von CNF und DNF)

Ist die Formel φ in CNF, dann ist $\neg\varphi$ modulo simpler Transformationen in DNF.

Beweis. Ist φ in CNF, dann hat φ die Form $\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$ mit Klauseln φ_i . Betrachte nun $\neg\varphi$, also $\neg(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n)$. Nach De Morgan ist dies äquivalent zu $\neg\varphi_1 \vee \neg\varphi_2 \vee \dots \vee \neg\varphi_n$. Wendet man nun De Morgan auch auf die Klauseln φ_i an und eliminiert doppelte Negation, dann erhält man Monome und die Formel $\neg\varphi$ ist in DNF. \square

In der Logic Workbench lwb transformiert die Funktion `dnf` eine beliebige Formel in die disjunktive Normalform:

```
(dnf '(impl P Q))
; => (or (and (not P)) (and Q))

(dnf '(and (impl P Q) R (or Q (not P))))
; => (or (and (not P) R Q) (and (not P) R) (and R Q))

(dnf '(and (impl P Q) (and R S) (or P (not Q))))
; => (or (and (not Q) (not P) S R) (and S R Q P))

(dnf? '(or (and (not P)) (and Q)))
; => true
```

Bemerkung. Die konjunktive und die disjunktive Normalform einer Formel kann man auch an der Wahrheitstafel ablesen, wie wir es im Beweis von Satz 4.5 getan haben.

6.4 Normalformen und Entscheidungsprobleme

Ein *komplementäres* Paar von Literalen ist eine Primaussage samt seiner Negation, also z.B. P und $\neg P$.

6.4.1 CNF und Gültigkeit

Satz 6.1 (CNF und Gültigkeit). *Sei φ in CNF. Dann gilt:*

φ ist allgemeingültig \Leftrightarrow Jede Klausel enthält ein komplementäres Paar von Literalen

Beweis. Wenn φ allgemeingültig ist, dann könnte man sich als kürzeste Darstellung in CNF \top vorstellen, \top gehört aber nicht zum Alphabet unsere Sprache. Eine Darstellung der Formel in CNF enthält also Klauseln mit Aussagensymbolen aus φ . Diese Klauseln sind mit \wedge verbunden, müssen also alle zu \top auswerten, damit die Formel wahr wird. Eine Klausel kann aber nur zu \top auswerten, wenn sie ein komplementäres Paar von Literalen enthält. \square

6.4.2 DNF und Erfüllbarkeit

Satz 6.2 (DNF und Erfüllbarkeit). *Sei ψ in DNF. Dann gilt:*

ψ ist unerfüllbar \Leftrightarrow Jedes Monom enthält ein komplementäres Paar von Literalen

Beweis. Die kürzeste Darstellung von ψ ist \perp . Angenommen die Kontradiktion ψ ist als Disjunktion von Monomen dargestellt. Dann müssen diese alle ebenfalls Kontradiktionen sein. Ein Monom ist genau dann unerfüllbar, wenn sie ein komplementäres Paar von Literalen enthält. \square

6.4.3 CNF und die Vollständigkeit des natürlichen Schließens

Wenn φ eine Tautologie in konjunktiver Normalform ist, dann kann man sehr einfach eine Herleitung für φ im Beweissystem des natürlichen Schließens konstruieren:

Da φ in CNF ist, brauchen wir für jede Klausel eine Herleitung, diese können wir dann durch die Regel $\wedge i$ zu einem Gesamtbeweis zusammensetzen.

Eine Herleitung für eine Klausel einer Tautologie erhält man so: Jede Klausel hat nach obigem Satz ein Paar komplementärer Literale. Wir haben also eine Klausel der Form $(\dots \vee P \vee \neg P \vee \dots)$. Dies ist sehr einfach herzuleiten: Wir verwenden TND und starten mit $P \vee \neg P$. Danach kommt die Auflösung des \vee durch Fallunterscheidung: einmal ist P gegeben und wir können die restlichen Literale der Klausel dazu „erfinden“. Und dann ist $\neg P$ gegeben und in diesem Fall können wir dasselbe tun.

Eine weitere Überlegung: Jeden Schritt des Algorithmus für die Umformung einer beliebigen Formel in die konjunktive Normalform kann man durch die Regeln des natürlichen Schließens nachvollziehen. Also können wir aus einer beliebigen Tautologie auch die konjunktive Normalform dieser Formel herleiten.

Diese beiden Überlegungen skizzieren einen weiteren konstruktiven Beweis für die Vollständigkeit des natürlichen Schließens als Beweissystem für die Aussagenlogik.

Kapitel 7

Die Komplexität des Erfüllbarkeitsproblems

7.1 Das Erfüllbarkeitsproblem

7.1.1 Entscheidungsfragen der Aussagenlogik

In der Aussagenlogik kann man sich zu einer gegebenen Formel φ folgende Fragen stellen:

- Ist φ *allgemeingültig*?

Diese Frage nennt man auch das Gültigkeitsproblem. Die Formel φ ist allgemeingültig, wenn sie für jede beliebige Belegung der Aussagensymbole wahr ist. Ist eine Formel φ allgemeingültig, nennt man φ auch eine *Tautologie*.

- Ist φ *unerfüllbar*?

Diese Frage nennt man auch das Unerfüllbarkeitsproblem. Die Formel φ ist unerfüllbar, wenn es keine Belegung der Aussagensymbole gibt, die sie wahr macht. Ist eine Formel φ unerfüllbar, sagt man auch φ ist der *Widerspruch* oder eine *Kontradiktion*.

- Ist φ *erfüllbar*?

Diese Frage nennt man auch das Erfüllbarkeitsproblem. Die Formel φ ist erfüllbar, wenn es eine Belegung gibt, unter der die Formel wahr ist. In der Regel ist man dann natürlich auch interessiert daran, ein solches Modell zu finden.

- Ist φ *falsifizierbar*?

Diese Frage nennt man auch das Widerlegungsproblem. Die Formel φ ist falsifizierbar oder widerlegbar, wenn es eine Belegung gibt, unter der die Formel falsch ist. Auch dann möchte man üblicherweise wissen, für welches Modell die Formel falsch ist.

Diese Fragen hängen miteinander zusammen. Denn aus den Definitionen ergibt sich unmittelbar:

- Eine Formel φ ist genau dann allgemeingültig, wenn $\neg\varphi$ unerfüllbar ist.
- Eine Formel φ ist genau dann falsifizierbar, wenn $\neg\varphi$ erfüllbar ist.

Also genügt es, die Frage zu betrachten, ob eine Formel *erfüllbar* ist. Wenn ja, unter welcher Belegung?

7.1.2 Das Erfüllbarkeitsproblem

Die Frage, ob eine Formel erfüllbar ist, wird als das *Erfüllbarkeitsproblem*, auch *SAT-Problem* (*SAT* = *satisfiability*) oder ganz kurz *SAT*, bezeichnet.

Es ist nicht schwierig, das Erfüllbarkeitsproblem zu lösen. Hat man eine Formel φ , dann stellt man die Wahrheitstafel für die Formel auf und sieht nach, ob es eine Zeile der Wahrheitstafel gibt, in der die Formel wahr ist.

Dieses Vorgehen ist jedoch nur für Formeln mit wenig Atomen geeignet. Hat φ n Atome, dann hat die Wahrheitstafel 2^n Zeilen. Hat also zum Beispiel eine Formel 100 Atome und die Untersuchung einer Zeile der Wahrheitstafel dauert 10^{-10} Sekunden, dann dauert die Untersuchung der gesamten Wahrheitstafel $2^{100} \times 10^{-10}$ Sekunden, also etwa 4×10^{12} Jahre. „Das ist länger als die Zeit, die seit der Entstehung des Universums vergangen ist.“¹

7.2 Komplexität von Algorithmen

Ein *Algorithmus* ist ein Verfahren, das in endlich vielen Schritten zur Lösung eines Problems führt. Mit dieser informellen Definition eines Algorithmus wollen wir im Folgenden arbeiten.²

Die *Komplexität* eines Algorithmus wird gemessen durch die Menge an Ressourcen (Zeit oder Speicher), die für die Durchführung des Algorithmus im Prinzip benötigt wird. Wir werden die Laufzeit eines Algorithmus im Folgenden als seine Komplexität betrachten.

¹ Die beispielhafte Berechnung der Ineffizienz der Entscheidung des Erfüllbarkeitsproblems durch die Wahrheitstafel ist aus: Uwe Schöning: *Das SAT-Problem* in: Informatik Spektrum Band 33 Heft 5 Oktober 2010

² Präziseres findet man in: [Sip13, Chap. 3.3]

7.2.1 Arten von Algorithmen

Definition 7.1. Ein Algorithmus ist *deterministisch*, wenn die Berechnung und somit das Ergebnis vollständig durch die Eingabe bestimmt wird. Ein deterministischer Algorithmus ist genau dann *korrekt*, wenn das Ergebnis zur gegebenen Eingabe korrekt ist.

Beispiel 7.1. Der Algorithmus mittels der Wahrheitstafel die Erfüllbarkeit einer Formel zu berechnen ist ein deterministischer korrekter Algorithmus für das Erfüllbarkeitsproblem.

Definition 7.2. Ein Algorithmus ist *nichtdeterministisch*, wenn seine Schritte mehrere mögliche Folgeschritte haben, deren Wahl nicht durch die Eingabe und bisherige Berechnung bestimmt wird. Zu einer Eingabe sind also mehrere verschiedene Berechnungsergebnisse möglich. Ein nichtdeterministischer Algorithmus heißt *korrekt*, wenn unter den möglichen Ergebnissen wenigstens eines korrekt ist.

Beispiel 7.2. Ein nichtdeterministischer Algorithmus für das Erfüllbarkeitsproblem ist leicht zu definieren. Der erste Schritt besteht darin, eine Belegung der Aussagensymbole zu raten. Im zweiten Schritt wird die Formel in dieser Belegung evaluiert.

Dieser Algorithmus ist sogar ein korrekter nichtdeterministischer Algorithmus. Wenn die Formel erfüllbar ist, dann gibt es eine erfüllende Belegung. Rät der Algorithmus mal richtig, dann entscheidet er das Erfüllbarkeitsproblem korrekt.

7.2.2 Laufzeit von Algorithmen

Nun definieren wir Maße für die Komplexität von Algorithmen:

Definition 7.3. Ein Algorithmus heißt *polynomiell*, wenn seine Laufzeit nach oben durch ein Polynom in n (n ist die Größe der Eingabe) beschränkt ist. Algorithmen, die in polynomieller Zeit ein korrektes Ergebnis liefern, werden oft auch als *effiziente* Algorithmen bezeichnet.

Beispiel 7.3. Unser nichtdeterministischer Algorithmus durch Erraten das Erfüllbarkeitsproblem zu lösen ist polynomiell: Das Erraten einer Belegung ist linear bezüglich der Größe der Eingabe, d.h. der Anzahl n der Atome und die Überprüfung, ob das Erratene zutrifft ist offensichtlich effizient durchführbar.

Definition 7.4. Ein Algorithmus läuft in *exponentieller* Zeit, wenn die Laufzeit nach unten durch eine Funktion 2^{cn} für die Größe der Eingabe n und ein $c > 0$ beschränkt ist.

Beispiel 7.4. Die Methode mit der Wahrheitstafel die Erfüllbarkeit einer Formel zu entscheiden ist exponentiell: Die Größe der Eingabe ist die Zahl n der Atome der Formel. Im schlechtesten Fall muss man zur Entscheidung alle 2^n Zeilen der Wahrheitstafel konstruieren.

7.2.3 Klassen von Problemen und \mathcal{NP} -Vollständigkeit

Definition 7.5. Die Klasse \mathcal{P} bezeichnet die Probleme, die in polynomieller Zeit durch einen deterministischen Algorithmus gelöst werden können.

Beispiel 7.5. Es ist nicht bekannt, ob das Erfüllbarkeitsproblem zur Klasse \mathcal{P} gehört. Es wird vermutet, dass dies nicht der Fall ist, wie wir gleich genauer diskutieren werden.

Definition 7.6. Die Klasse \mathcal{NP} bezeichnet die Probleme, die ein nicht-deterministischer Algorithmus in polynomieller Zeit lösen kann.

Beispiel 7.6. Das Erfüllbarkeitsproblem gehört zur Klasse \mathcal{NP} . Man sagt auch: SAT ist \mathcal{NP} .

Man kann auch so ausdrücken, dass ein Problem in \mathcal{NP} ist: eine *Lösung* des Problems ist in polynomieller Zeit *überprüfbar*, auch wenn sie möglicherweise nicht in polynomieller Zeit *gefunden* werden kann.

Vermutung: Ist $\mathcal{P} = \mathcal{NP}$? Dies ist eine grundlegende Frage der Informatik, die ungelöst ist.³ Allgemein wird vermutet, dass gilt:

$$\mathcal{P} \neq \mathcal{NP}.$$
⁴

Zur genaueren Bestimmung der Komplexität des Erfüllbarkeitsproblems benötigen wir noch weitere Definitionen:

Definition 7.7. Ein Problem P ist \mathcal{NP} -schwierig, wenn sich jedes Problem Q in \mathcal{NP} deterministisch in polynomieller Zeit auf P reduzieren lässt.

³ Das Clay Mathematics Institute <http://www.claymath.org> hat dieses Problem als eines der 7 Millenniums-Probleme ausgewählt – neben u.a. der Riemann-Vermutung oder der (mittlerweile von Grigori Perelman gelösten) Poincaré-Vermutung.

⁴ Donald E. Knuth sieht das anders: “...almost everybody who has studied the subject thinks that satisfiability cannot be decided in polynomial time. The author of this book, however, suspects that $N^{O(1)}$ -step algorithms do exist, yet that they’re unknowable. Almost all polynomial time algorithms are so complicated that they lie beyond human comprehension, and could never be programmed for an actual computer in the real world. Existence is different from embodiment.” [Knu15, S. 1]

Definition 7.8. Ein Problem P ist \mathcal{NP} -vollständig, wenn es in \mathcal{NP} und \mathcal{NP} -schwierig ist.

7.3 Die Komplexität des Erfüllbarkeitsproblems

Nun stehen alle Definitionen zur Verfügung, um den Satz zu formulieren, der die Komplexität des Erfüllbarkeitsproblems bestimmt:

Satz 7.1 (Cook 1971, Levin 1973). *Das Erfüllbarkeitsproblem ist \mathcal{NP} -vollständig.⁵*

Wie sieht es mit dem komplementären Problem der Nichterfüllbarkeit bzw. der Allgemeingültigkeit aus? Dieses Problem ist insofern schwieriger, als man ja zeigen muss, dass es kein Modell gibt, bzw. dass die Aussage in allen Belegungen wahr ist. Hier hilft „Raten“ nicht mehr. Genauer sagt man:

Definition 7.9. Ein Problem ist in der Klasse $Co\text{-}\mathcal{NP}$, wenn das komplementäre Problem in \mathcal{NP} ist.

Beispiel: Das Nichterfüllbarkeitsproblem ist in $Co\text{-}\mathcal{NP}$.

Satz 7.2. $Co\text{-}\mathcal{NP} = \mathcal{NP}$ genau dann, wenn Nichterfüllbarkeit in \mathcal{NP} ist.

Vermutung: Es ist nicht bekannt, ob es einen nichtdeterministischen polynomiellen Algorithmus für das Nichterfüllbarkeitsproblem gibt. Man nimmt vielmehr an, dass gilt:

$$Co\text{-}\mathcal{NP} \neq \mathcal{NP}$$

Die Ergebnisse über die Komplexität des Erfüllbarkeitsproblems können zu der Annahme verleiten, dass es keinen „effizienten“ Algorithmus für SAT geben kann und deshalb Probleme, die man als aussagenlogische Formeln formulieren kann, nur gelöst werden können, wenn man wenige aussagenlogischen Atome in der Formel hat.

⁵ Stephen A. Cook, amerikanischer Informatiker, heute Professor für Informatik in Toronto. Er erhielt 1982 für diesen Satz den Turing-Award. Leonid Levin, ukrainischer Informatiker, hat die Theorie der \mathcal{NP} -Vollständigkeit und den Satz über das Erfüllbarkeitsproblem 1973 entwickelt. Seine Ergebnisse waren im Westen zunächst nicht bekannt. 1978 emigrierte er in die USA.

Dies ist jedoch keineswegs der Fall: „Zum Glück gibt es und gab es Algorithmenentwickler, Theoretiker und Praktiker, die sich von diesem Negativergebnis [Satz von Cook und Levin] nicht abschrecken ließen. Dies hat in den vergangenen Jahren dazu geführt, dass die ‚SAT-Solver‘-Technologie immer weiter vorangeschritten ist [und] dass diese das SAT-Problem lösenden Verfahren heutzutage mit Formeln, die Tausende von Variablen enthalten, in Sekundenbruchteilen fertig werden.“⁶

⁶ So schreibt Uwe Schöning im bereits zitierten Artikel. Ein solcher SAT-Solver ist **Sat4j** – siehe <http://www.sat4j.org/>.

Kapitel 8

Hornlogik

Betrachtet man spezielle Klassen von Formeln, dann findet man oft effiziente Algorithmen für die Entscheidung des Erfüllbarkeitsproblems. Eine wichtige solche Klasse sind die Hornformeln¹.

Definition 8.1 (Hornklausel). Eine Klausel $\hat{P}_1 \vee \hat{P}_2 \vee \dots \vee \hat{P}_n$ heißt *Hornklausel*, wenn höchstens eines der Literale \hat{P}_i positiv ist.

Definition 8.2 (Hornformel). Eine Formel φ in CNF heißt *Hornformel*, wenn jede Klausel eine Hornklausel ist.

Definition 8.3 (Arten von Hornklauseln).

- Besteht eine Hornklausel nur aus einem positiven Literal, dann heißt sie *Tatsachenklausel*, kurz *Tatsache*.
- Hat eine Hornklausel ein positives Literal und mindestens ein negatives Literal, dann heißt sie *Prozedurklausel* oder *Regel*.
- Hat die Hornklausel kein positives Literal, dann heißt sie *Zielklausel* oder *Frageklausel*, kurz *Ziel*.

Beispiel 8.1 (Hornklauseln und Hornformel).

- $\{P\}$ ist eine Tatsachenklausel,
- $\{\neg P, Q\}$ und $\{\neg P, \neg Q, R\}$ sind Regeln und
- $\{\neg P, \neg R\}$ ist eine Zielklausel.

Die Hornformel in diesem Beispiel ist also:

$$P \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee R) \wedge (\neg P \vee \neg R)$$

Wir können Hornformel auch (alternativ) definieren, indem wir eine Grammatik angeben:

¹ nach Alfred Horn (1918–2001), amerikanischer Mathematiker.

Definition 8.4 (Hornformel). Eine Hornformel ist eine Formel φ der Aussagenlogik, die folgender Grammatik genügt:

$$\begin{aligned} T &:= \top \rightarrow P \text{ für Aussagensymbole } P \text{ (Tatsache)} \\ P &:= P \mid P \wedge P \text{ für Aussagensymbole } P \\ Z &:= P \mid P \rightarrow \perp \text{ (Ziel)} \\ R &:= P \rightarrow Q \text{ für Aussagensymbole } Q \text{ (Regel)} \\ H &:= \varphi \wedge T \mid \varphi \wedge Z \mid \varphi \wedge R \end{aligned}$$

Die Bezeichnungen *Tatsache*, *Regel*, *Ziel* ergeben sich aus folgenden Überlegungen:

1. Jede Hornklausel ist entweder eine Tatsache, eine Regel oder ein Ziel.
2. Eine Tatsache muss wahr sein, d.h. $P \equiv \top \rightarrow P$.
3. Eine Prozedurklausel (Regel) $\neg P_1 \vee \dots \vee \neg P_n \vee Q$ ist äquivalent zu $P_1 \wedge \dots \wedge P_n \rightarrow Q$.
4. Eine Zielklausel $\neg P_1 \vee \dots \vee \neg P_n$ ist äquivalent zu $\neg(P_1 \wedge \dots \wedge P_n)$, also $P_1 \wedge \dots \wedge P_n \rightarrow \perp$.

Beispiel 8.2 (Hornformel). Die Hornformel aus Beispiel 8.1 ausgedrückt mit der Grammatik 8.4:

$$(\top \rightarrow P) \wedge (P \rightarrow Q) \wedge (P \wedge Q \rightarrow R) \wedge (P \wedge R \rightarrow \perp)$$

In der Hornlogik wird typischerweise ein *Widerlegungsverfahren* verwendet. In der Hornformel wird die angestrebte Aussage als *nicht* wahr angenommen (deshalb ist die Form der Zielklausel so wie oben definiert). Kann man nun die Nichterfüllbarkeit der Hornformel zeigen, sieht man, dass die Zielklausel zutrifft.

Wir formulieren einen Algorithmus für die Entscheidung der Erfüllbarkeit von Hornformeln (den sogenannten Markierungsalgorithmus):

```
function HORN( $\varphi$ ) {
    // pre:  $\varphi$  ist eine Hornformel
    // post: entscheidet die Frage, ob  $\varphi$  erfüllbar ist
    markiere alle  $\top$  in  $\varphi$ 
    while ( es gibt  $P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow Q$  mit
             $P_1, P_2, \dots, P_n$  markiert, aber  $Q$  nicht) {
        markiere  $Q$ 
```

```

    }
    if (  $\perp$  ist markiert ){
        return „unerfüllbar“
    } else {
        return „erfüllbar“
    }
}

```

Eigenschaften dieses Algorithmus

1. Er entscheidet, ob die Hornformel φ erfüllbar ist.
2. Ist φ erfüllbar, dann können wir eine erfüllende Belegung ablesen, nämlich

$$v(P_i) = \begin{cases} \text{T falls } P_i \text{ markiert ist} \\ \text{F sonst} \end{cases}$$

3. Der Algorithmus endet nach spätestens $n + 2$ Markierungsschritten, wenn n die Zahl der Atome in φ ist.

Beispiel 8.3 (Markierungsalgorithmus).

Gegeben sei die Aussage P , d.h. $\top \rightarrow P$, eine *Tatsache*.

Ferner sei bekannt, dass folgende *Regeln* gelten: $P \rightarrow Q$ und $P \wedge Q \rightarrow R$. Man möchte nun in dieser Situation wissen, ob $P \wedge R$ gilt.

In der Hornlogik klärt man die Frage durch die *Widerlegung* des Gegenstands: Man nimmt also an, dass $P \wedge R$ nicht gilt. Durch diese Annahme entsteht folgende Hornformel:

$$(\top \rightarrow P) \wedge (P \rightarrow Q) \wedge (P \wedge Q \rightarrow R) \wedge (P \wedge R \rightarrow \perp)$$

Wenn nun gezeigt werden kann, dass diese Formel *unerfüllbar* ist, dann hat man gezeigt, dass in der gegebenen Situation $P \wedge R$ gilt. Dies zeigt man nun mit Hilfe des Markierungsalgorithmus:

$$\begin{array}{ccccccccc} (\top \rightarrow P) \wedge & (P \rightarrow Q) \wedge & (P \wedge Q \rightarrow R) \wedge & (P \wedge R \rightarrow \perp) \\ * & & & & & & & & \\ * & * & * & * & & & * & & \\ * & * & * & * & * & * & & * & \\ * & * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * & * \end{array}$$

Der Algorithmus endet in diesem Beispiel, wenn \perp markiert ist. Das bedeutet, dass die Formel unerfüllbar ist, d.h. dass in der gegebenen Situation $P \wedge R$ gilt.

Kapitel 9

Erfüllbarkeit und SAT-Solver

Die Entscheidungsfragen der Aussagenlogik lassen sich alle auf das Erfüllbarkeitsproblem zurückspielen.

Ein Programm, das für eine Formel der Aussagenlogik das Erfüllbarkeitsproblem löst (und eine erfüllende Belegung ermittelt), heißt *SAT-Solver*.

Die Kunst in der Entwicklung von SAT-Solvern besteht darin, Algorithmen zu finden, die für große Klassen von Formeln das Problem effizient entscheiden, obgleich es \mathcal{NP} -vollständig ist.¹

Typischerweise setzen SAT-Solver voraus, dass eine Formel in CNF vorliegt. Das EingabefORMAT ist üblicherweise DIMACS (vorgeschlagen vom Center for Discrete Mathematics and Theoretical Computer Science <http://dimacs.rutgers.edu/>).

9.1 DIMACS-Format

SAT-Solver verwenden eine einfache Variante des DIMACS-Formats. Eine Formel in CNF wird in einer ASCII-Datei gespeichert, die in drei Sektionen aufgebaut ist:

Zuerst kommen optionale *Kommentarzeilen*, die durch ein kleines c an der ersten Position gekennzeichnet sind.

¹ Donald E. Knuth: “The story of satisfiability is the tale of a triumph of software engineering, blended with rich doses of beautiful mathematics. Thanks to elegant new data structures and other techniques, modern SAT solvers are able to deal routinely with practical problems that involve many thousands of variables, although such problems were regarded as hopeless just a few years ago.”[Knu15, S. iv]

Darauf folgt die *Präambel*, die aus einer Zeile der Form

p cnf v c

besteht, wobei v für die Zahl der Atome der Formel² und c für die Zahl der Klauseln steht.

Danach folgen die *Klauseln*. Die Literale werden durch Integers codiert. Dabei steht die positive Zahl für ein Atom, die negative Zahl für seine Negation. Jede Klausel nimmt eine Zeile der Datei ein, sie enthält die Literale getrennt durch Leerzeichen und wird abgeschlossen durch eine 0.

Beispiel φ sei die folgende Formel in CNF:

$$\begin{aligned}
 & (P_1 \vee P_2 \vee P_3) \wedge \\
 & (P_1 \vee \neg P_2 \vee \neg P_3) \wedge \\
 & (P_1 \vee \neg P_5) \wedge \\
 & (\neg P_2 \vee \neg P_3 \vee \neg P_5) \wedge \\
 & (\neg P_1 \vee \neg P_2 \vee P_3) \wedge \\
 & (P_4 \vee P_6) \wedge \\
 & (P_4 \vee \neg P_6) \wedge \\
 & (P_2 \vee \neg P_4) \wedge \\
 & (\neg P_3 \vee \neg P_4)
 \end{aligned}$$

Sie wird im DIMACS-Format so geschrieben:

```

c Beispiel DIMACS Vorlesung LfM
p cnf 6 9
1 2 3 0
1 -2 -3 0
1 -5 0
-2 -3 -5 0
-1 -2 3 0
4 6 0
4 -6 0
2 -4 0
-3 -4 0

```

Doch stop! Wir haben gesehen, dass die Umformung einer beliebigen Formel in eine äquivalente Formel in CNF im schlechtesten Fall eine exponentielle Laufzeit (in Bezug auf die Zahl der Atome der Formel) haben kann. Führt das nicht schon von vorneherein dazu, dass die Entscheidung der Erfüllbarkeit *nicht* effizient sein kann, noch ehe der SAT-Solver überhaupt startet, weil CNF als Eingabeform erwartet wird? Dies ist nicht der Fall:

² Sieht man die aussagenlogische Formel als eine Boolesche Funktion spricht man auch von *Variablen*, deshalb der Buchstabe v.

9.2 Tseitin-Transformation

Definition 9.1 (Erfüllbarkeitsäquivalenz). Zwei Formeln der Aussagenlogik φ und ψ heißen *erfüllbarkeitsäquivalent*, wenn gilt:

$$\varphi \text{ erfüllbar} \Leftrightarrow \psi \text{ erfüllbar}.$$

Die *Tseitin³-Transformation* besteht nun darin, eine beliebige Formel φ in eine *erfüllbarkeitsäquivalente* Formel in CNF umzuformen. Die Tseitin-Transformation ist linear in der Zahl der Atome der Formel.

```
function TSEITIN( $\varphi$ ) {
// post: Eine erfüllbarkeitsäquivalente Formel  $\varphi'$  in CNF
```

1. Führe für jede Subformel ψ von φ , die kein Atom ist, ein neues Atom T_ψ hinzu.
2. Setze $\varphi' = T_\varphi$.
3. Durchlaufe den Syntaxbaum von φ und füge φ' je nach Form der Subformel an einem inneren Knoten die Formel in CNF nach Tabelle 9.1 verbunden mit \wedge hinzu. (Atome werden analog zu den anderen Subformeln behandelt.)

}

Tabelle 9.1: Regeln für die Tseitin-Transformation

$\varphi = \neg\varphi_1$	$(\neg T_\varphi \vee \neg T_{\varphi_1}) \wedge (T_\varphi \vee T_{\varphi_1})$
$\varphi = \varphi_1 \wedge \varphi_2$	$(\neg T_\varphi \vee T_{\varphi_1}) \wedge (\neg T_\varphi \vee T_{\varphi_2}) \wedge (T_\varphi \vee \neg T_{\varphi_1} \vee \neg T_{\varphi_2})$
$\varphi = \varphi_1 \vee \varphi_2$	$(T_\varphi \vee \neg T_{\varphi_1}) \wedge (T_\varphi \vee \neg T_{\varphi_2}) \wedge (\neg T_\varphi \vee T_{\varphi_1} \vee T_{\varphi_2})$
$\varphi = \varphi_1 \rightarrow \varphi_2$	$(T_\varphi \vee T_{\varphi_1}) \wedge (T_\varphi \vee \neg T_{\varphi_2}) \wedge (\neg T_\varphi \vee \neg T_{\varphi_1} \vee T_{\varphi_2})$

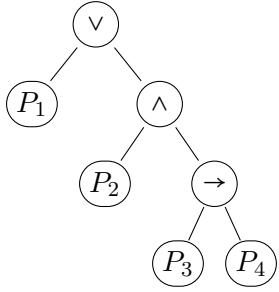
Die Tseitin-Transformation φ' einer Formel φ hat folgende Eigenschaften:

1. Die Menge der Atome von φ ist eine Teilmenge der Atome von φ' .
2. Wenn v eine erfüllende Belegung von φ ist, dann gibt es eine Erweiterung von v auf die Atome von φ' , so dass diese Belegung φ' erfüllt.
3. Ist v' eine erfüllende Belegung von φ' , dann erfüllt sie auch φ .

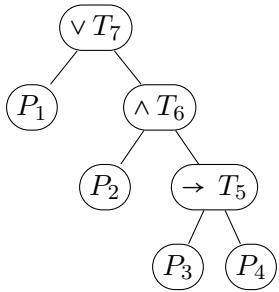
9.2.1 Beispiel

Betrachten wir die Formel $\varphi = P_1 \vee (P_2 \wedge (P_3 \rightarrow P_4))$.

³ nach Grigori S. Zeitin, russischer Mathematiker, [Tse83].



Im Schritt 1 legen wir pro Knoten, der nicht Blatt ist, ein neues Atom fest.



Im Schritt 2 erstellen wir φ' :

$$\begin{aligned}
 & T_7 \wedge \\
 & (T_7 \vee \neg P_1) \wedge (T_7 \vee \neg T_6) \wedge (\neg T_7 \vee P_1 \vee T_6) \wedge \\
 & (\neg T_6 \vee P_2) \wedge (\neg T_6 \vee T_5) \wedge (T_6 \vee \neg P_2 \vee \neg T_5) \wedge \\
 & (T_5 \vee P_3) \wedge (T_5 \vee \neg P_4) \wedge (\neg T_5 \vee \neg P_3 \vee P_4)
 \end{aligned}$$

9.3 DPLL und CDCL

Definition 9.2 (Wert-Propagation). Sei φ eine Formel in CNF und \hat{P} ein Literal. Dann bezeichnet $\varphi[\tau/\hat{P}]$ die Formel, die dadurch entsteht, dass \hat{P} durch den Wert τ ersetzt wird.

Durch die Wert-Propagation wird die gegebene Formel folgendermaßen vereinfacht:

- Jede Klausel der Formel, die \hat{P} enthält, fällt weg. Denn sie wird durch die Wertzuweisung von τ an \hat{P} wahr.

- In jeder Klausel der Formel, in der $\neg\hat{P}$ vorkommt, entfällt dieser Ausdruck, da er \perp ist und deshalb die Klausel nicht wahr machen kann.

Der Basis-Algorithmus von SAT-Solvern basiert wesentlich auf der Wert-Propagation.

Definition 9.3 (Einzelklausel). Eine Klausel einer Formel in CNF heißt *Einzelklausel*, wenn sie aus genau einem Literal besteht.

Definition 9.4 (Reines Literal). Ein Literal in einer Formel in CNF heißt *reines Literal*, wenn es in allen Klauseln nur als P oder $\neg P$ vorkommt.

Viele SAT-Solver verwenden (stark weiterentwickelte) Varianten des folgenden Algorithmus von Martin Davis⁴, Hilary Putnam⁵, George Logemann⁶ und Donald W. Loveland⁷ ([DP60], [DLL62]).

```
function DPLL( $\varphi, \mathcal{M}$ ) {
    // pre:  $\varphi$  eine Formel in CNF,  $\mathcal{M}$  eine partielle Belegung
    // return: true falls  $\varphi$  erfüllbar ist
    // post: Eine Belegung  $\mathcal{M} = \{\hat{P}_1, \hat{P}_2, \dots\}$  der Aussagensymbole
    // von  $\varphi$ , falls  $\varphi$  erfüllbar ist
    // modifies:  $\mathcal{M}$ 

    case {
         $\varphi = \top$ :
            return true;
         $\varphi = \perp$ :
            return false;
         $\varphi$  hat eine Einzelklausel ( $\hat{P}$ ):
            return DPLL( $\varphi[\top/\hat{P}], \mathcal{M} \cup \hat{P}$ );
         $\varphi$  hat ein reines Literal  $\hat{P}$ :
            return DPLL( $\varphi[\top/\hat{P}], \mathcal{M} \cup \hat{P}$ );
        otherwise:
            wähle zu einem Atom  $Q$  in der Formel mit  $\hat{Q} \notin \mathcal{M}$ 
            return DPLL( $\varphi[\top/Q], \mathcal{M} \cup Q$ )  $\vee$  DPLL( $\varphi[\perp/Q], \mathcal{M} \cup \neg Q$ );
    }
}
```

⁴ Martin Davis, amerikanischer Logiker und Informatiker.

⁵ Hilary Putnam (1926–2016), amerikanischer Philosoph.

⁶ George Logemann (1938–2012), amerikanischer Mathematiker.

⁷ Donald W. Loveland, amerikanischer Informatiker.

9.3.1 Idee des Algorithmus

1. Propagation von Einzelklauseln

Besteht eine Klausel nur aus einem Literal, also P oder $\neg P$, dann muss $P = \text{true}$ bzw. $\neg P = \text{true}$ sein, damit die Formel in CNF erfüllbar wird.

Folge: Man kann alle Klauseln weglassen, in denen \hat{P} vorkommt, und außerdem $\neg\hat{P}$ in allen Klauseln, in denen es vorkommt.

2. Elimination von reinen Literalen

Wenn in der Formel nur \hat{P} , niemals aber $\neg\hat{P}$ vorkommt, dann kann man $\hat{P} = \text{true}$ setzen. Denn dadurch fallen alle Klauseln weg, die \hat{P} enthalten, d.h. die Wahl kann niemals zu einem Widerspruch führen, weil ja weder P noch $\neg P$ noch vorkommt.

Folge: Man kann alle Klauseln weglassen, in denen \hat{P} vorkommt, denn sie sind dann erfüllt.

3. Rekursion (*Backtracking*)

Wenn keine der genannten Möglichkeiten bestehen, muss man ein Aussagensymbol Q der Formel φ wählen. Es gilt dann natürlich: φ erfüllbar $\Leftrightarrow \varphi \wedge Q$ erfüllbar oder $\varphi \wedge \neg Q$ erfüllbar.

9.3.2 Beispiele

Wir betrachten zunächst das einfache Beispiel der Formel, an der wir den Markierungsalgorithmus für Horn-Formeln demonstriert haben:

$$\begin{aligned} & P \wedge \\ & (\neg P \vee Q) \wedge \\ & (\neg P \vee \neg Q \vee R) \wedge \\ & (\neg P \vee \neg R) \end{aligned}$$

Im ersten Schritt setzt man $P \text{ true}$, da die erste Klausel eine Einzelklausel ist. Die Propagation dieses Werts führt dazu, dass diese Klausel selbst wegfällt und in allen anderen Klauseln $\neg P$ gestrichen werden kann. Man erhält also:

$$\begin{aligned} & Q \wedge \\ & (\neg Q \vee R) \wedge \\ & \neg R \end{aligned}$$

Nun ist Q auf **true** zu setzen und dieser Wert zu propagieren, also folgt:

$$\begin{array}{c} R \wedge \\ \neg R \end{array}$$

Und diese beiden Einzelklauseln bilden den Widerspruch, d.h. die Formel ist *unerfüllbar*.

Nach diesem einfachen Beispiel, das gezeigt hat, dass der Markierungsalgorithmus im Grunde mit der Idee der Propagation von Einzelklauseln arbeitet, folgt das Beispiel von 9.1:

φ sei wieder die folgende Formel in CNF:

$$\begin{aligned} & (P_1 \vee P_2 \vee P_3) \wedge \\ & (P_1 \vee \neg P_2 \vee \neg P_3) \wedge \\ & \quad (P_1 \vee \neg P_5) \wedge \\ & (\neg P_2 \vee \neg P_3 \vee \neg P_5) \wedge \\ & (\neg P_1 \vee \neg P_2 \vee P_3) \wedge \\ & \quad (P_4 \vee P_6) \wedge \\ & \quad (P_4 \vee \neg P_6) \wedge \\ & \quad (P_2 \vee \neg P_4) \wedge \\ & \quad (\neg P_3 \vee \neg P_4) \end{aligned}$$

Schritt 1 Es gibt keine Einzelklausel, aber $\neg P_5$ ist ein reines Literal, d.h. wir brauchen für P_5 nur den Fall $\neg P_5 = \text{true}$ zu betrachten.

D.h. $\mathcal{M} = \{\neg P_5\}$ und $\varphi[\top/\neg P_5]$ enthält die Klauseln mit $\neg p_5$ nicht mehr, also

$$\begin{aligned} & (P_1 \vee P_2 \vee P_3) \wedge \\ & (P_1 \vee \neg P_2 \vee \neg P_3) \wedge \\ & (\neg P_1 \vee \neg P_2 \vee P_3) \wedge \\ & \quad (P_4 \vee P_6) \wedge \\ & \quad (P_4 \vee \neg P_6) \wedge \\ & \quad (P_2 \vee \neg P_4) \wedge \\ & \quad (\neg P_3 \vee \neg P_4) \end{aligned}$$

Schritt 2 Probiere $P_1 = \text{true}$, also $\mathcal{M} = \{\neg P_5, P_1\}$. Die Klauseln mit P_1 sind **true** und in den Klauseln mit $\neg P_1$ kann man $\neg P_1$ weglassen. Dann bleibt

$$\begin{aligned} & (\neg P_2 \vee P_3) \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (P_2 \vee \neg P_4) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

Schritt 3 Probiere $P_2 = \text{true}$, also $\mathcal{M} = \{\neg P_5, P_1, P_2\}$. Die Klauseln mit P_2 sind **true** und in den Klauseln mit $\neg P_2$ kann man $\neg P_2$ weglassen. Dann bleibt

$$\begin{aligned} & P_3 \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

P_3 ist nun eine Einzelklausel, d.h. P_3 muss **true** sein, also $\mathcal{M} = \{\neg P_5, P_1, P_2, P_3\}$. Es bleibt:

$$\begin{aligned} & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & \neg P_4 \end{aligned}$$

$\neg P_4$ ist nun auch Einzelklausel, und es bleibt:

$$\begin{aligned} & P_6 \wedge \\ & \neg P_6 \end{aligned}$$

Dies ist aber der Widerspruch.

Schritt 4 Probiere $P_2 = \text{false}$, also $\mathcal{M} = \{\neg P_5, P_1, \neg P_2\}$. Dann bleibt

$$\begin{aligned} & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & \neg P_4 \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

$\neg P_4$ ist jetzt Einzelklausel, d.h. P_4 muss **false** sein, d.h.

$$\begin{array}{c} P_6 \wedge \\ \neg P_6 \end{array}$$

Erneut der Widerspruch.

Schritt 5 Probiere $P_1 = \text{false}$, also $\mathcal{M} = \{\neg P_5, \neg P_1\}$. Dann bleibt:

$$\begin{aligned} & (P_2 \vee P_3) \wedge \\ & (\neg P_2 \vee \neg P_3) \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (P_2 \vee \neg P_4) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

Schritt 5 Probiere $P_2 = \text{true}$, also $\mathcal{M} = \{\neg P_5, \neg P_1, P_2\}$. Dann bleibt:

$$\begin{aligned} & \neg P_3 \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

$\neg P_3$ ist jetzt Einzelklausel, d.h. P_3 muss **false** sein, also $\mathcal{M} = \{\neg P_5, \neg P_1, P_2, \neg P_3\}$. Dann bleibt:

$$\begin{aligned} & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \end{aligned}$$

P_4 ist jetzt reines Literal, d.h. P_4 muss **true** sein, dann bleibt nichts mehr übrig, alle Klauseln sind erfüllt. D.h. die Wahl der Belegung für P_6 ist beliebig.

Ergebnis $\mathcal{M} = \{\neg P_5, \neg P_1, P_2, \neg P_3, P_4, P_6\}$ ist eine erfüllende Belegung.

9.3.3 Von DPLL zu CDCL

Der vorgestellte DPLL-Algorithmus gibt die Idee wieder. Heutige SAT-Solver verwenden verbesserte Algorithmen mit folgenden Eigenschaften:

1. Oft wird keine Analyse bezüglich reiner Literale gemacht, weil diese Analyse aufwändig ist.
2. Stattdessen wird im Fall eines *Konflikts* genau analysiert, wodurch er entstanden ist und eine neue Klausel hinzugefügt, die dazu führt, dass der Algorithmus einmal gefundene Abhängigkeiten zwischen Atomen nicht „vergisst“ und deshalb erneut auswerten muss.

Die Idee kann man an unserem Beispiel oben sehen. Wir haben in Schritt 1 P_1 als `true` und in Schritt 2 P_2 als `true` gewählt. Diese beiden Entscheidungen haben zum Widerspruch geführt. Also muss gelten: $\neg P_1 \vee \neg P_2$. Diese Klausel können wir nun unserer ursprünglichen Formel hinzufügen. Der Algorithmus hat aus dem Konflikt „gelernt“.

Klausel-lernende SAT-Solver werden auch CDCL-Solver (*Conflict Driven Clause Learning*) genannt.

3. Diese Analyse kann noch weitergehen: Man merkt sich in welchem Level der Analyse man eine Entscheidung getroffen hat und macht bei einem Konflikt nicht einfach *Backtracking*, sondern *Backjumping*.
4. Außerdem werden Heuristiken eingesetzt, welche Variable beim Backjumping „ausprobiert“ wird. Ein Beispiel wäre etwa: DLCS (*Dynamic Largest Combined Sum*) – man wählt die Variable, die positiv oder negativ am häufigsten in der Formel vorkommt.

Würde man diese Strategie in unserem Beispiel verwenden, müsste man in Schritt 2 mit $P_2 = \text{true}$ starten, danach P_3 mit `true` probieren, was zum Konflikt führt. Doch $P_3 = \text{false}$ führt zu einer erfüllenden Belegung. Wir wären also etwas schneller.

Wie aktuelle SAT-Solver arbeiten, wird in [KS06] und [Knu15] beschrieben.

Kapitel 10

Anwendungen der Aussagenlogik in der Softwaretechnik

In der Programmierung spielt die Aussagenlogik eine prominente Rolle, wenn immer Bedingungen zu formulieren sind, also etwa bei bedingten Anweisungen oder Fallunterscheidungen. In diesem Kapitel werfen wir einen kurzen Blick auf einige andere Anwendungen der Aussagenlogik und von SAT-Solvern in der Softwaretechnik.

10.1 Anwendungen von SAT-Techniken

Einige Beispiele für den Einsatz von SAT-Solvern, um nicht-triviale Fragestellungen zu lösen.

- 2003 – Validierung von Produktkonfigurationen in der Automobilindustrie, Carsten Sinz et al, siehe [[SKK03](#)].
- 2009 – Formale Verifizierung des Intel Core 7 Prozessors, Kaivola et al, siehe [[KGN⁺09](#)]
- 2010 – Verifikation von Windows 7 Gerätetreibern mit einem SMT-Solver, De Moura, Björner, siehe [[dMB10](#)]. SMT (SAT modulo Theory) baut auf dem Konzept von SAT-Solvern auf. Microsoft Research hat einen SMT-Solver namens Z3 entwickelt, siehe <https://github.com/Z3Prover/z3/wiki>.
- 2014 – Analyse der Terminierung von Programmen, Jürgen Giesl et al, siehe [[GBE⁺14](#)].
- 2016/18 – Management von Abhängigkeiten innerhalb des Ökosystems der Eclipse Plattform, Le Berre, Rapicault, siehe [[BR18](#)].
- ...

10.2 Statische Codeanalyse

Das folgende Beispiel ist aus [KS06, Example 2.2], von mir zu einer kleinen Geschichte ausgebaut.

Bei einem Codereview fällt das Auge des Reviewers auf folgendes Codestück:

```
if (!a && !b) h();
else if (!a) g();
else f();
```

Der Entwickler versichert, dass der Code getestet wurde und in allen Testfällen das gewünschte Ergebnis ergab. Gleichwohl ist der Reviewer der Auffassung, dass dieses Codestück doch einfacher und damit für Menschen verständlicher geschrieben werden können müsste. Etwas unwillig erklärt sich der Entwickler dazu bereit und kommt am kommenden Tag wieder.

Er bringt zwei Fassungen mit:

Version 1:

```
if (a) h();
else if (b) g();
else f();
```

Version 2:

```
if (a) f();
else if (b) g();
else h();
```

und richtet milde lächelnd die Frage an den Reviewer: „Und welche der Versionen sollen wir nun nehmen?“

Der Reviewer freilich kennt seine Aussagenlogik und übersetzt die ursprüngliche Fassung sowie die beiden neuen Versionen indem er die Terme der Bedingungen sowie die Funktionen zu Aussagensymbolen macht:

Original:

```
if ( $\neg a \wedge \neg b$ ) then  $h$ 
else if ( $\neg a$ ) then  $g$ 
else  $f$ 
```

Version 1 (2 analog):

```
if  $a$  then  $h$ 
else if  $b$  then  $g$ 
else  $f$ 
```

Nun muss man nur noch **if-then-else** in eine Formel transformieren:

$$\text{if } x \text{ then } y \text{ else } z \rightsquigarrow (x \wedge y) \vee (\neg x \wedge z)$$

Und schon hat man drei Formeln:

$$\begin{aligned}\varphi_{orig} &\hat{=} ((\neg a \wedge \neg b) \wedge h) \vee (\neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g) \vee (a \wedge f)) \\ \varphi_{v_1} &\hat{=} (a \wedge h) \vee (\neg a \wedge ((b \wedge g) \vee (\neg b \wedge f))) \\ \varphi_{v_2} &\hat{=} (a \wedge f) \vee (\neg a \wedge ((b \wedge g) \vee (\neg b \wedge h)))\end{aligned}$$

Jetzt kann man einen SAT-Solver verwenden, um zu überprüfen, ob $\varphi_{orig} \leftrightarrow \varphi_{v_1}$ oder $\varphi_{orig} \leftrightarrow \varphi_{v_2}$ gilt.

Mit der Logic Workbench kann man sich die Sache noch einfacher machen, weil es in ihr den dreistelligen Operator `ite` gibt:

```
; Originaler Code
(def orig '(ite (and (not a) (not b)) h (ite (not a) g f)))
; Version 1
(def vers1 '(ite a h (ite b g f)))
; Version 2
(def vers2 '(ite a f (ite b g h)))

; Was ist richtig?
(valid? (list 'equiv orig vers1))
; => false
(valid? (list 'equiv orig vers2))
; => true
```

10.3 Feature-Modelle für (Software-)Produktlinien

In der Softwareentwicklung kommt es nicht selten vor, dass verwandte Anwendungen entwickelt werden, die Gemeinsamkeiten, aber auch Unterschiede haben. Wenn man dies auf *systematische* Weise tut, dann spricht man von einer *Softwarereproduktlinie*.

Die systematische Analyse von Gemeinsamkeit und Variabilität innerhalb einer Softwarereproduktlinie wird mittels *Feature-Modellierung* durchgeführt. Die Eigenschaften der verschiedenen Produkte in einer Produktlinie werden als *Features* bezeichnet.

Die *Features* werden im Feature-Modell in einem Baum angeordnet, wobei verschiedene Arten von Kanten zwischen Super- und Subfeatures die Variabilität beschreiben. Außerdem gibt es Integritätsbedingungen, die über Zweige des Baums hinweggehen, sogenannte *Cross Tree Constraints* (CTCs). Der Feature-Baum zusammen mit den CTCs ergibt das Feature-Modell.

Eine Auswahl von Features, die alle Bedingungen im Feature-Modell erfüllt, wird als eine gültige *Konfiguration* bezeichnet. Können noch weitere Features gewählt werden, spricht man von einer *partiellen* Konfiguration, besteht keine Wahlmöglichkeit mehr von einer *vollständigen* Konfiguration.

Typischerweise werden in Feature-Modelle folgende vier Arten der Beziehungen zwischen Super- und Subfeatures unterschieden:

- ① Ein Subfeature ist *obligatorisch*, d.h. wenn das Superfeature in einer Konfiguration gewählt wird, dann ist das Subfeature automatisch auch gewählt.
- ② Ein Subfeature ist *optional*, d.h. wenn das Superfeature in einer Konfiguration gewählt wird, dann kann das Subfeature gewählt werden oder auch nicht.
- ③ Eine Gruppe von Subfeatures wird als *Oder-Gruppe* dargestellt, wenn die Wahl des Superfeatures erzwingt, dass mindestens eines der Subfeatures gewählt werden muss.
- ④ Eine Gruppe von Subfeatures wird als *Alternativ-Gruppe* dargestellt, wenn die Wahl des Superfeatures erzwingt, dass genau eines der Subfeatures gewählt werden muss.

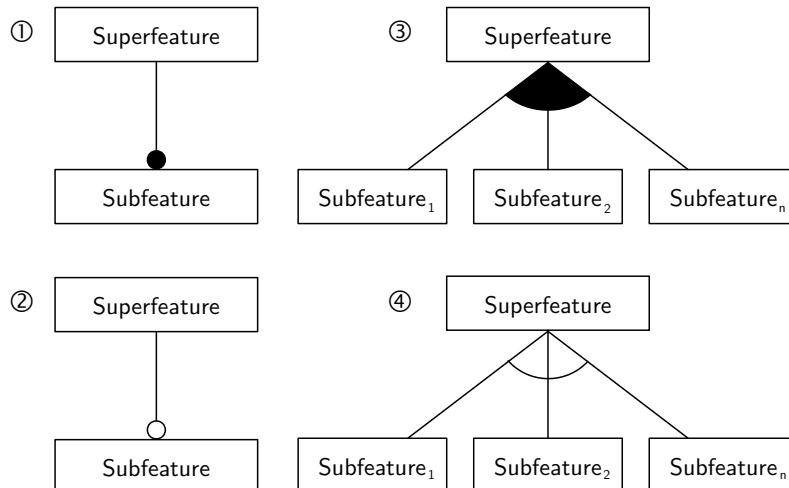


Abbildung 10.1: Beziehungen im Feature-Baum

Abbildung 10.1 stellt dar, wie diese vier Beziehungen im Feature-Diagramm graphisch dargestellt werden.

Integritätsbedingungen zwischen beliebigen Features, die unabhängig von der Baumstruktur sind, die CTCs, werden als aussagenlogische Formeln ausgedrückt, wobei die Features die Aussagensymbole sind. (Das setzt voraus, dass die Bezeichnungen der Features im Baum eindeutig sind.)

Die Feature-Modellierung wurde als Bestandteil der *Feature Oriented Domain Analysis* (FODA) 1990 von Kyo C. Kang et al. beim Software Engineering Institute SEI eingeführt [KCH⁺90]. Seither wurden viele Varianten des Feature-Modells entwickelt, einen Überblick findet man

in [MH07]. Ich verwende die Notation, die in der FeatureIDE, einem Werkzeug der Feature-Modellierung, eingesetzt wird, siehe [ABKS13] und [MTS⁺17].

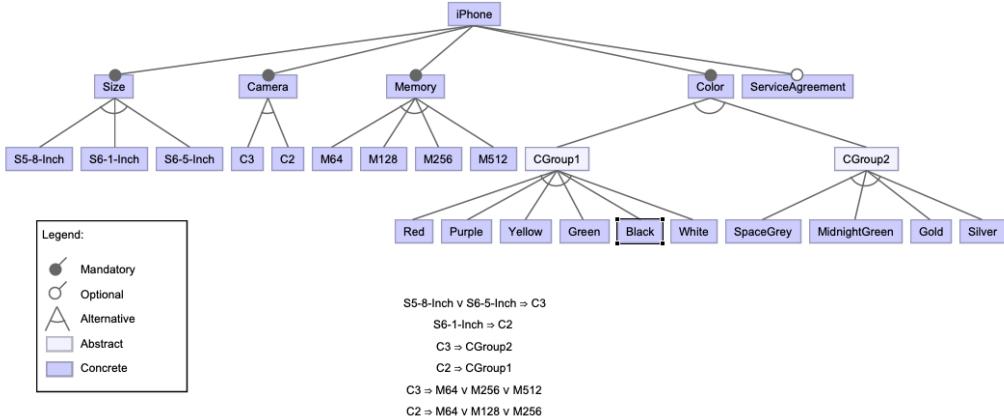


Abbildung 10.2: Feature-Modell für das iPhone

Abbildung 10.2 zeigt ein Feature-Modell für die Varianten des iPhones 2019. Das Modell wurde in FeatureIDE (siehe <https://featureide.github.io> und [MTS⁺17]) erstellt. In der Legende werden Features nach *abstract* und *concrete* unterschieden. Abstrakte Features dienen der Strukturierung des Feature-Baums, sie sind als solche nicht im Produkt konkret auffindbar. Die Aussagen unterhalb des Baums sind die CTCs.

Was nun hat das Feature-Modell mit der Aussagenlogik zu tun? Wir nehmen die Features als Aussagensymbole und verstehen zugeordnete Wahrheitswerte als Festlegungen, ob das Feature für die zu treffende Konfiguration gewählt wurde (also T ist) oder nicht (also F ist).

Das Feature-Modell entspricht auf diese Weise einer Formel der Aussagenlogik, die als Konjunktion folgender Teilformel gebildet wird:

- Jede mögliche Konfiguration soll ja tatsächlich ein Produkt definieren, dessen Varianten im Modell beschrieben werden. Das bedeutet, dass die Wurzel *Root* des Baums T sein muss, d.h. *Root* ist eine Teilformel:

$$Root$$

- Bei obligatorischen Subfeatures besteht eine logische Äquivalenz zum Superfeature: Ist das Superfeature *Super T*, dann muss auch das Subfeature *oblSub T* sein. Und wenn das Subfeature T ist, dann

ist das nur möglich, wenn das Superfeature gewählt ist, also

$$oblSub \leftrightarrow Super$$

- Beim optionalen Subfeature $optSub$ ist dieses bei Wahl des Super nicht zwingend erforderlich, also

$$optSub \rightarrow Super$$

- Bei einer Oder-Gruppe von Subfeatures $oSub_1, oSub_2, \dots, oSub_n$ gilt

$$oSub_1 \vee oSub_2 \vee \dots \vee oSub_n \leftrightarrow Super$$

- Bei einer Alternativ-Gruppe von Subfeatures $aSub_1, aSub_2, \dots, aSub_n$ muss exakt eines der Subfeatures der Wahl des Superfeatures entsprechen:

$$(aSub_1 \vee aSub_2 \vee \dots \vee aSub_n) \wedge \bigwedge_{i < j} (\neg aSub_i \vee \neg aSub_j) \leftrightarrow Super$$

- Cross Tree Constraints werden der Formel einfach hinzufügt.

Auf diese Weise (zuerst beschrieben in [Bat05]) entspricht ein Feature-Modell gerade einer Formel der Aussagenlogik und die erfüllenden Belegungen der Formel sind gerade die möglichen vollständigen Konfigurationen von Varianten.

Das bereits erwähnte Werkzeug FeatureIDE verwendet diese Korrespondenz und setzt den SAT-Solver **SAT4J** ein, um Feature-Modelle zu analysieren und Modellierer zu unterstützen.

Teil II

Prädikatenlogik

Kapitel 11

Objekte und Prädikate

Die Ausdruckskraft der Aussagenlogik ist beschränkt. Wir werden folgenden Schluss sicherlich für richtig halten:

Alle Quadratzahlen $\neq 0$ sind positiv (P)

16 ist eine Quadratzahl (Q)

Also folgt: 16 ist positiv (R)

Übertragen wir diesen Schluss etwas naiv in die Aussagenlogik, so ergibt sich $P \wedge Q \rightarrow R$ — und es gibt keinerlei Grund anzunehmen, dass diese Aussage zutrifft. Dies liegt daran, dass wir in der Aussagenlogik den Zusammenhang zwischen der ersten und der zweiten Aussage nicht erfassen können, nämlich, dass die 16 ein Exemplar von der Sorte *Quadratzahl* ist.

Also: Wir müssen unterscheiden können zwischen

Objekten, den Dingen eines Universums, einer „Welt“, wie z.B. Zahlen, Strings, Werten, Objekten usw. und

Prädikaten, Aussagen über die Objekte, die wahr oder falsch sein können.

Beispiel Ist etwa unser Universum die Welt der ganzen Zahlen \mathbb{Z} , dann könnten wir folgende Prädikate haben

$Sq(x)$ bedeutet „ x ist eine Quadratzahl $\neq 0$ “

$Pos(x)$ bedeutet „ x ist positiv“

Dann können wir das einleitende Beispiel so ausdrücken

$$\forall x(Sq(x) \rightarrow Pos(x)) \wedge Sq(16) \rightarrow Pos(16)$$

Und nun kann man formal argumentieren: Wenn für alle x die Implikation $Sq(x) \rightarrow Pos(x)$ zutrifft, dann ist dies sicherlich auch für ein bestimmtes x , die 16 der Fall, also $Sq(16) \rightarrow Pos(16)$. Ist weiter $Sq(16)$ gegeben, dann folgt mit dem Modus ponens, dass wir in der Tat $Pos(16)$ schließen dürfen.

11.1 Elemente der Sprache der Prädikatenlogik

Gegeben sei stets eine Menge \mathbb{U} , das Universum, auch genannt „Miniwelt“. In der Regel wird in der Literatur vorausgesetzt, dass das Universum nicht leer ist, d.h. $\mathbb{U} \neq \emptyset$ ¹

Wir verwenden zusätzlich zu den Junktoren der Aussagenlogik

Variablen

$x, y, z \dots$

Variablen sind Platzhalter für beliebige Objekte des Universums, z.B. steht in $Sq(x)$ die Variable x für ein Element des Universums, also in unserem Beispiel für eine Zahl in \mathbb{Z} .

Konstanten

$c, d, e \dots$

Konstanten sind bestimmte, benannte Elemente des Universums, z.B. 16, die Zahl $16 \in \mathbb{Z}$.

Funktionen

$f, g, h \dots$

Funktionen operieren auf den Elementen des Universums und ergeben wieder Elemente des Universums, z.B. $plus(16, 9)$ ergibt das Element $25 \in \mathbb{Z}$ mit der naheliegenden Definition von $plus$.

Prädikate

$P, Q, R \dots$

Prädikate sind Boolesche Funktionen, die Aussagen über Elemente der Welt machen. Die Wertemenge eines Prädikats ist also in $\mathbb{B} = \{\text{T}, \text{F}\}$. In unserem Beispiel ist Sq ein Prädikat der Arität 1, $Sq(x)$ ist genau dann wahr, wenn $x \in \mathbb{Z}$ eine Quadratzahl ist.

¹ Ist das Universum leer, dann ist jede Aussage der Form $\exists x \dots$ immer falsch und eine der Form $\forall x \dots$ immer wahr – diese Fälle möchte man gerne vermeiden.

Gleichheit

=

Gleichheit ist ein spezielles Prädikat, das wir in die Sprache von vorneherein aufnehmen.²

Quantoren

$\forall x \varphi, \exists y \varphi$

Quantoren machen Aussagen über *alle* Elemente des Universums oder darüber, ob ein Element des Universums mit einer bestimmten Eigenschaft *existiert*.

11.2 Prädikate und Relationen

Es besteht eine wichtige grundlegende Beziehung zwischen *Prädikaten und Relationen*:

Sei R eine n -wertige Relation über \mathbb{U} , d.h. $R \subseteq \mathbb{U}^n$. Dann kann man diese Relation repräsentieren durch die Boolesche Funktion $r : \mathbb{U}^n \rightarrow \mathbb{B}$ definiert durch

$$r(a_1, \dots, a_n) = \text{T} \text{ genau dann, wenn } (a_1, \dots, a_n) \in R$$

Ist etwa $Sq \subseteq \mathbb{Z} = \{x / \exists y \text{ mit } x = y^2 \text{ und } x > 0\}$, also

$$Sq = \{1, 4, 9, 16, 25, \dots\}$$

dann entspricht diese einstellige Relation gerade dem oben verwendeten Prädikat Sq .

Offenbar sind also Prädikate und Relationen austauschbare Konzepte, deshalb spricht man bei der Prädikatenlogik erster Ordnung auch manchmal von *relationaler Logik*.

Bemerkung. Man könnte in unserem Beispiel auch eine binäre Relation $Sq \subset \mathbb{N}^2$ auf folgende Weise definieren:

Sei $square : \mathbb{N} \rightarrow \mathbb{N}$ die Funktion, die einem $x \in \mathbb{N}$ sein Quadrat als Funktionswert zuordnet, also $x \mapsto x^2$. Diese Funktion kann man als binäre Relation auffassen, nämlich $Sq = \{(x, x^2) / x \in \mathbb{N}\}$. Ein Tupel (x, y) ist also genau dann in Sq , wenn y das Quadrat von x ist. Auf diese Weise entspricht jeder n -stelligen Funktion eine $n + 1$ -stellige Relation.

² Man kann auch Prädikatenlogik ohne Gleichheit machen, für Anwendungen ist es jedoch nützlich, die Gleichheit als *logisches* Symbol aufzufassen.

11.3 Beispiele von Aussagen in der Prädikatenlogik

In diesem Abschnitt wollen wir einige typische Aussagen in der Prädikatenlogik an einfachen Beispielen formulieren, um ein Gefühl für diese formale Sprache zu bekommen. Die Bedeutung der dabei verwendeten Prädikate und Konstanten ergibt sich aus dem Kontext.

Alle Dinge sind rot	$\forall x Rot(x)$
Es gibt ein rotes Ding	$\exists x Rot(x)$
Kein Ding ist rot	$\neg \exists x Rot(x)$ oder $\forall x \neg Rot(x)$

Oft formuliert man Aussagen über Typen/Klassen von Objekten:

Alle Feuerwehrautos sind rot	$\forall x (FA(x) \rightarrow Rot(x))$
Es gibt ein rotes Fahrrad	$\exists x (FR(x) \wedge Rot(x))$
Kein Feuerwehrauto ist blau	$\neg \exists x (FA(x) \wedge Blau(x))$ oder $\forall x (FA(x) \rightarrow \neg Blau(x))$

Wenn man die Aussage über die Feuerwehrautos so formalisieren würde: $\forall x (FA(x) \wedge Rot(x))$, dann würde das bedeuten, dass alle Objekte rote Feuerwehrautos sind, was sicherlich nicht gemeint ist.

Man könnte versucht sein, die zweite Aussage über das Fahrrad so auszudrücken: $\exists x (FR(x) \rightarrow Rot(x))$. Wenn es nun kein rotes Fahrrad gibt, könnten wir irgendein Objekt a , das kein Fahrrad ist, nehmen. Dann gilt $\neg FR(a)$, also wäre $FR(a) \rightarrow Rot(x)$ wahr, obwohl es gar kein rotes Fahrrad gibt.

Mit mehrstelligen Prädikaten und auch mit Funktionen kann man Beziehungen zwischen Objekten herstellen:

Hans ist Fahrer eines Feuerwehrautos	$\exists x (FA(x) \wedge FahrerVonFA(hans, x))$
Jedes Feuerwehrauto hat einen Fahrer	$\forall x (FA(x) \rightarrow \exists y (Person(y) \wedge FahrerVonFA(y, x)))$

Man kann auch Aussagen über die Anzahl von Objekten mit bestimmten Eigenschaften machen:

Es gibt ein Fahrrad	$\exists x FR(x)$
Es gibt genau ein Fahrrad	$\exists x (FR(x) \wedge \forall y (FR(y) \rightarrow y = x))$
Es gibt mindestens zwei Fahrräder	$\exists x \exists y (FR(x) \wedge FR(y) \wedge \neg (x = y))$

In den folgenden Abschnitten wollen wir die Syntax und die Semantik der Prädikatenlogik präzise erörtern.

Kapitel 12

Die formale Sprache der Prädikatenlogik

12.1 Signatur, Terme, Formeln

In der Sprache der Prädikatenlogik verwendet man Funktions- und Konstantensymbole sowie Prädikatssymbole. Genau genommen bezieht man sich auf eine bestimmte Wahl dieser Symbole und spricht dann von einer Sprache \mathcal{L} der Prädikatenlogik.

Definition 12.1 (Signatur). Die *Signatur* einer Sprache \mathcal{L} besteht aus einer Menge von Prädikatssymbolen $\{P_1, \dots, P_n\}$, von Funktionssymbolen $\{f_1, \dots, f_m\}$ und von Konstantensymbolen $\{c_1, \dots, c_k\}$.

Man schreibt die Signatur so:

$$\{P_1^{r_1}, \dots, P_n^{r_n}; f_1^{a_1}, \dots, f_m^{a_m}; c_1, \dots, c_k\}$$

wobei die r und die a die Arität der Prädikatssymbole bzw. Funktionssymbole bezeichnet.

Bemerkungen

- Die Signatur besteht aus den „nicht-logischen“ Symbolen der Sprache \mathcal{L} .
- Manche Autoren definieren die Signatur dadurch, dass sie nur die Arität von Prädikatssymbolen, Funktionssymbolen und die Zahl der Konstanten angeben. Denn „Name ist Schall und Rauch“¹. In dieser Sicht wird die Signatur so angegeben:

$$< r_1, \dots, r_n; a_1, \dots, a_m; k >$$

¹ J.W. von Goethe, Faust I, Marthens Garten

wobei r_i die Arität eines Prädikatensymbols, a_i die Arität eines Funktionssymbols und k die Zahl der Konstanten ist.

- Man kann Konstantensymbole auch als Funktionssymbole der Arität 0 auffassen.
- Prädikatssymbole der Arität 0 kann man als Aussagensymbole auffassen. Oder anders: Wenn in der Signatur nur Prädikatssymbole der Arität 0 vorkommen und keine anderen Symbole und wir uns auf die Junktoren $\neg, \wedge, \vee, \rightarrow$ beschränken, dann erhalten wir gerade die Definition einer Sprache der Aussagenlogik wie in Teil I.

Im Folgenden sei eine Signatur gegeben, sowie eine Menge von Variablen $\{x, y, \dots\}$.

Definition 12.2 (Terme). Die *Terme* sind Zeichenketten, die nach folgenden Regeln gebildet werden:

- (i) Jede Variable ist ein Term.
- (ii) Jedes Konstantensymbol ist ein Term.
- (iii) Sind t_1, \dots, t_n Terme und f ein Funktionssymbol mit der Arität n , dann ist auch $f(t_1, \dots, t_n)$ ein Term.

Als *Grammatik* in Backus-Naur-Darstellung können wir diese Regeln so ausdrücken:

$$t ::= x \mid c \mid f \mid f(t, \dots, t)$$

mit Variablen x , Konstantensymbolen c und Funktionssymbolen f .

Definition 12.3 (Formeln). Die *Formeln* sind die Zeichenketten, die durch folgende Regeln generiert werden können:

- (i) \perp ist eine Formel (genannt: der Widerspruch) und \top ist eine Formel (genannt: die Wahrheit).
- (ii) Ist P ein Prädikatssymbol der Arität 0, dann ist P eine Formel.
Ist P ein Prädikatssymbol der Arität $n \geq 1$ und sind t_1, \dots, t_n Terme, dann ist $P(t_1, \dots, t_n)$ eine Formel.
Sind t_1 und t_2 Terme, dann ist $(t_1 = t_2)$ eine Formel.
- (iii) Ist φ eine Formel, dann auch $(\neg\varphi)$.
Sind φ und ψ Formeln, dann auch $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ und $(\varphi \rightarrow \psi)$.
- (iv) Ist φ eine Formel und x eine Variable, dann sind $\forall x \varphi$ und $\exists x \varphi$ Formeln.

In Backus-Naur-Darstellung:

$$\begin{aligned}\varphi ::= & \perp \mid \top \mid P \mid P(t, \dots, t) \mid (t = t) \mid \\ & (\neg\varphi), \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \\ & \forall x \varphi \mid \exists x \varphi\end{aligned}$$

mit Termen t , Variablen x und Prädikatssymbolen P .

Bemerkung. In der Logic Workbench (lwb) hat man als Junktoren die Junktoren aus der Aussagenlogik, siehe Tabelle 3.1, darüber hinaus:

- das ausgezeichnete Symbol $=$ für die Gleichheit,
- den Allquantor, z.B. `(forall [x y] (and (P x) (Q y)))` mit unären Prädikaten P und Q sowie
- den Existenzquantor, z.B. `(exists [x] (= (f x) (g x)))` mit den unären Funktion f und g .

Bemerkungen

- Die Argumente von Prädikatssymbolen dürfen *nur* Terme sein, keine anderen Prädikate – wir definieren hier nämlich die Prädikatenlogik *erster* Stufe.
- In unserer Definition der Sprachen der Prädikatenlogik, haben wir die Gleichheit = als *logisches* Symbol mit aufgenommen. In vielen Büchern wird unterschieden zwischen der Prädikatenlogik und der Prädikatenlogik mit Gleichheit. Man könnte denken, dass man die Gleichheit einfach dadurch einführen kann, dass man ein Prädikatsymbol für die Gleichheit definiert. Wir werden sehen, dass dies zu „Nebenwirkungen“ führen könnte, die wir vermeiden, wenn wir die Gleichheit als Teil der Logik betrachten (siehe Abschnitt 13.1).

Bindungsregeln

$\neg, \forall x, \exists x$ binden am stärksten, dann
 \wedge und \vee (linksassoziativ) und schließlich
 \rightarrow (rechtsassoziativ.)

Diese Bindungsregeln erlauben uns, sparsamer (und dadurch besser lesbar) mit Klammern umzugehen, ohne die Grammatik mehrdeutig zu machen.

Wie in der Aussagenlogik stellt der Syntaxbaum einer Formel ihren Aufbau dar.

Beispiel 12.1. Abbildung 12.1 zeigt den Syntaxbaum der Formel

$$\forall x((P(x) \rightarrow Q(f(x))) \wedge S(x, y))$$

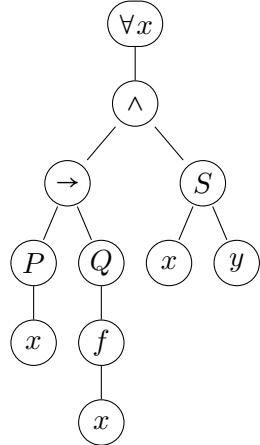


Abbildung 12.1: Syntaxbaum von $\forall x((P(x) \rightarrow Q(f(x))) \wedge S(x, y))$

12.2 Freie und gebundene Variablen

Sei φ eine Formel der Prädikatenlogik. Ein Vorkommen der Variablen x in φ heißt *frei*, wenn x im Syntaxbaum nicht Abkömmling eines Quantorknotens $\forall x$ oder $\exists x$ ist. Andernfalls heißt das Vorkommen *gebunden*.

Die Namen von gebundenen Variablen spielen keine Rolle, deshalb werden Formeln, die sich nur durch den Namen gebundener Variablen unterscheiden, identifiziert. $\forall x P(x, y)$ ist also dieselbe Formel wie $\forall z P(z, y)$, nicht jedoch $\forall x P(x, z)$, da die Bedeutung der freien Variablen vom Kontext abhängt.

Beispiel 12.2. Abbildung 12.2 zeigt den Syntaxbaum der Formel

$$\forall x(P(x) \wedge Q(x)) \rightarrow (\neg P(x) \vee Q(y))$$

mit der Angabe welche der Variablen frei und gebunden vorkommen. Die Variable x kommt sowohl frei als auch gebunden vor.

Bemerkung. Jede Formel lässt sich in eine semantisch äquivalente Formel in *bereinigter* Form umwandeln, d.h. in eine Formel, in der keine Variable sowohl frei als auch gebunden vorkommt und in der hinter jedem Quantor eine andere Variable steht. Man verwendet für die gebundenen Variablen neue Namen.

Im Beispiel 12.2 kann die Formel

$$\forall x(P(x) \wedge Q(x)) \rightarrow (\neg P(x) \vee Q(y))$$

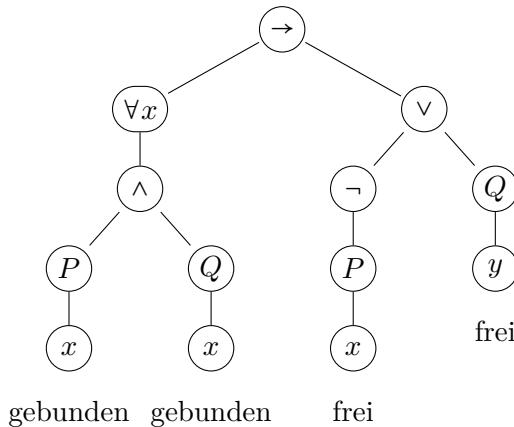


Abbildung 12.2: Gebundenes und freies Vorkommen von Variablen in einer Formel

in bereinigter Form so geschrieben werden:

$$\forall z(P(z) \wedge Q(z)) \rightarrow (\neg P(x) \vee Q(y))$$

Definition 12.4 (Satz). Eine Formel φ , die keine freien Variablen hat, heißt *Satz* oder *geschlossene Formel*.

Definition 12.5 (Grundterm). Ein Term t , der keine Variablen hat, heißt *geschlossener Term* oder *Grundterm*.

Bemerkung. Hat eine Formel freie Variablen, dann macht man sie zu einem Satz, in dem man sich alle freien Variablen durch den Allquantor gebunden denkt.

12.3 Substitution

Variablen sind Platzhalter für Terme. Substitution besteht darin, solche Platzhalter durch Terme zu ersetzen.

Ein Ersetzen von Variablen durch Terme ist nur möglich für Variablen, die *nicht* durch einen Quantor gebunden sind. Gebundene Variablen stehen für ein bestimmtes Element des Universums oder für alle Elemente des Universums, können somit nicht durch den Term ersetzt werden.

Definition 12.6 (Substitution). Gegeben eine Variable x und ein Term t . Die Formel $\varphi[t/x]$ ist die Formel, die aus φ entsteht, in dem jedes *freie* Vorkommen von x durch t ersetzt wird.

Dabei darf der eingesetzte Term t keine Variablen enthalten, die durch die Substitution in den Geltungsbereich eines Quantors kommen würden.

Die einschränkende Bemerkung zur Substitution sei an einem Beispiel erläutert:

Haben wir die Formel $\forall x P(x, y)$ und wollen y durch den Term $f(x)$ ersetzen, dann ergäbe sich durch einfaches Einsetzen $\forall x P(x, f(x))$ und plötzlich ist die Variable x im Term $f(x)$ in den Bereich des Allquantors gekommen. Dies ist nicht erlaubt.

Richtig wäre es, zunächst die gebundene Variable umzubenennen, also etwa $\forall z P(z, y)$, wodurch sich die Formeln nicht ändert. Jetzt ist die Substitution möglich:

$$(\forall x P(x, y))[f(x)/y] \text{ ergibt } \forall z P(z, f(x)), \text{ nicht aber } \forall x P(x, f(x)).$$

Man sagt, dass eine Term t frei ist für eine Variable x in einer Formel φ , wenn t keine Variable enthält, die beim Einsetzen für x in den Geltungsbereich eines Quantors käme.

Bei einer Substitution muss man also erst prüfen, ob der zu substituierende Term frei für die Variable in der Formel ist. Ist dies nicht der Fall, muss man gebundene Variablen in der Formel so umtaufen, dass keine „Kollision“ auftritt.

Beispiel 12.3. Gegeben sei die Formel

$$\varphi \stackrel{\text{def}}{=} \forall x (P(x) \rightarrow Q(x) \wedge S(x, y))$$

und es soll die Substitution von x mit dem Term $f(x)$ durchgeführt werden. Da alle Vorkommen von x gebunden sind, ergibt sich als Ergebnis für $\varphi[f(x)/x]$ die unveränderte Formel

$$\forall x (P(x) \rightarrow Q(x) \wedge S(x, y))$$

Beispiel 12.4. Gegeben sei die Formel

$$\varphi \stackrel{\text{def}}{=} (\forall x (P(x) \wedge Q(x)) \rightarrow (\neg P(x) \vee Q(y)))$$

In diesem Fall kommt x sowohl gebunden als auch frei vor. Somit ergibt sich für $\varphi[f(x)/x]$ die Formel

$$(\forall x (P(x) \wedge Q(x)) \rightarrow (\neg P(f(x)) \vee Q(y)))$$

Beispiel 12.5. Wir betrachten die Formel

$$\varphi \stackrel{\text{def}}{=} S(x) \wedge \forall y (P(x) \rightarrow Q(y))$$

und wollen $\varphi[f(y)/x]$ berechnen. Nun ist das zweite freie Vorkommen der Variablen x nicht durch $f(y)$ substituierbar, da dann der Term $f(x)$ in den Einflussbereich des Allquantors $\forall y$ kommen würde.

Zunächst muss also eine Form der Formel verwendet werden, die die Substitution erlaubt, also zum Beispiel

$$S(x) \wedge \forall z(P(x) \rightarrow Q(z))$$

Dann ist das Ergebnis der Substitution

$$S(f(y)) \wedge \forall z(P(f(y)) \rightarrow Q(z))$$

Kapitel 13

Semantik der Prädikatenlogik

13.1 Modell/Struktur

Definition 13.1 (Modell/Struktur). Ein *Modell* für die Sprache \mathcal{L} ist ein Paar $\mathcal{M} = \langle \mathbb{U}, I \rangle$ mit einer nichtleeren Menge \mathbb{U} und einer Funktion I , die jedem Symbol in \mathcal{L} eine Interpretation zuordnet nach folgenden Regeln:

- (i) Ist P ein 0-äres Prädikatensymbol, dann ist $I(P)$ ein Wahrheitswert.
- (ii) Ist P ein n -äres Prädikatensymbol für $n > 0$, dann ist $I(P) \subseteq \mathbb{U}^n$ eine n -äre Relation über \mathbb{U} .
- (iii) Ist c ein Konstantensymbol, dann ist $I(c) \in \mathbb{U}$, ein Element von \mathbb{U} .
- (iv) Ist f ein Funktionssymbol mit Arität n , dann ist $I(f) : \mathbb{U}^n \rightarrow \mathbb{U}$ eine Funktion.

Man nennt Modelle der Prädikatenlogik auch spezifischer *Strukturen*.

Die Menge \mathbb{U} nennt man auch das Universum. Die Interpretation schreibt man auch oft so:

- $P^{\mathcal{M}}$ für die Prädikate über \mathbb{U}
- $c^{\mathcal{M}}$ für die Elemente zu den Konstantensymbolen und
- $f^{\mathcal{M}}$ für die Funktionen zu den Funktionssymbolen

Bemerkung. Die Gleichheit in der Prädikatenlogik ist Teil der logischen Sprache und nicht definiert als binäres Prädikat. Der Grund liegt darin, dass die Definition der Gleichheit als Prädikat dazu führen würde, dass

es Modelle geben könnte, die sich *nur* durch die entsprechende Relation unterscheiden würden. Wir möchten aber, dass in *jedem* Modell die Gleichheit die Identitätsrelation ist, also $\{(x, x) \mid x \in \mathbb{U}\}$. (Siehe auch [Hof11, S.114f].)

Definition 13.2 (Variablenbelegung). Sei $\mathcal{M} = \langle \mathbb{U}, I \rangle$ eine Struktur für \mathcal{L} . Eine *Variablenbelegung* in \mathcal{M} ist eine Funktion l , die jeder Variablen x einen Wert $l(x) \in \mathbb{U}$ zuordnet.

Man schreibt $l[x \mapsto a]$ für die Variablenbelegung, die x auf a abbildet und alle anderen Variablen auf $l(y)$, d.h.

$$l[x \mapsto a](y) = \begin{cases} a & \text{falls } y = x. \\ l(y) & \text{falls } y \neq x. \end{cases}$$

Definition 13.3 (Interpretation der Terme). Sei $\mathcal{M} = \langle \mathbb{U}, I \rangle$ eine Struktur für \mathcal{L} und l eine Variablenbelegung. Für einen Term t von \mathcal{L} definiert man die Interpretation $I(t)$ bezüglich \mathcal{M} und der Variablenbelegung l induktiv über die Länge des Terms durch

- (i) $I(x) := l(x)$ für die Variablen x ,
- (ii) $I(c) := I(c)$ für die Konstanten c , und
- (iii) $I(f(t_1, \dots, t_n)) := I(f)(I(t_1), \dots, I(t_n))$ für die Funktionen f .

Definition 13.4 (Interpretation der Formeln). Sei $\mathcal{M} = \langle \mathbb{U}, I \rangle$ eine Struktur für \mathcal{L} und l eine Variablenbelegung, dann ist \mathcal{M} ein Modell für eine Formel φ , geschrieben

$$\mathcal{M} \vDash_l \varphi \quad \text{für die Formel } \varphi,$$

falls $\llbracket \varphi \rrbracket_l^{\mathcal{M}} = T$. Dabei wird $\llbracket \varphi \rrbracket_l^{\mathcal{M}}$ induktiv definiert über den strukturellen Aufbau von φ :

$$\begin{aligned} (i) \quad \llbracket P(t_1, \dots, t_n) \rrbracket_l^{\mathcal{M}} &:= \begin{cases} T & \text{falls } (I(t_1), \dots, I(t_n)) \in P^{\mathcal{M}} \\ F & \text{sonst} \\ & \text{bzw. } I(P) \text{ falls } P \text{ 0-är} \end{cases} \\ (ii) \quad \llbracket s = t \rrbracket_l^{\mathcal{M}} &:= \begin{cases} T & \text{falls } I(s) = I(t) \\ F & \text{sonst} \end{cases} \\ (iii) \quad \llbracket \neg \varphi \rrbracket_l^{\mathcal{M}} &:= \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_l^{\mathcal{M}} = F \\ F & \text{sonst} \end{cases} \\ (iv) \quad \llbracket \varphi \wedge \psi \rrbracket_l^{\mathcal{M}} &:= \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_l^{\mathcal{M}} = T \text{ und } \llbracket \psi \rrbracket_l^{\mathcal{M}} = T \\ F & \text{sonst} \end{cases} \end{aligned}$$

-
- (v) $\llbracket \varphi \vee \psi \rrbracket_l^{\mathcal{M}} := \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_l^{\mathcal{M}} = \text{T oder } \llbracket \psi \rrbracket_l^{\mathcal{M}} = \text{T} \\ \text{F sonst} \end{cases}$
- (vi) $\llbracket \varphi \rightarrow \psi \rrbracket_l^{\mathcal{M}} := \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_l^{\mathcal{M}} = \text{F oder } \llbracket \psi \rrbracket_l^{\mathcal{M}} = \text{T} \\ \text{F sonst} \end{cases}$
- (vii) $\llbracket \forall x \varphi \rrbracket_l^{\mathcal{M}} := \begin{cases} \text{T falls für alle } a \in \mathbb{U} \text{ gilt: } \llbracket \varphi \rrbracket_{l[x \mapsto a]}^{\mathcal{M}} = \text{T} \\ \text{F sonst} \end{cases}$
- (viii) $\llbracket \exists x \varphi \rrbracket_l^{\mathcal{M}} := \begin{cases} \text{T falls es existiert ein } a \in \mathbb{U} \text{ mit: } \llbracket \varphi \rrbracket_{l[x \mapsto a]}^{\mathcal{M}} = \text{T} \\ \text{F sonst} \end{cases}$
- (ix) $\llbracket \top \rrbracket_l^{\mathcal{M}} := \text{T}$
- (x) $\llbracket \perp \rrbracket_l^{\mathcal{M}} := \text{F}$

Bemerkung. Eine Formel φ der Prädikatenlogik ist eine Folge von Symbolen, die nach den Regeln der Grammatik der formalen Sprache der Prädikatenlogik gebildet werden. Außer den logischen Symbolen müssen die Prädikatssymbole, die Funktionssymbole sowie die Konstantensymbole, kurz die *Signatur* vorgegeben sein.

Was ist nun die *Semantik* dieser Formel? Wir betrachten eine „konkrete Welt“, ein Modell von Elementen, Prädikaten, Funktionen und Konstanten und können in diesem Modell auswerten, ob φ in diesem Modell tatsächlich zutrifft oder nicht.

In dieser Definition der Semantik ist die Wahl des Modells überaus „liberal“. Wenn wir uns fragen, ob eine gegebene Formel φ erfüllbar ist, dann dürfen wir die Frage bejahen, wenn es uns gelingt, ein Modell zu „erfinden“, in dem die Formel gilt. Diese Freiheit in der Wahl eines Modells soll des folgende Beispiel demonstrieren.

Beispiel 13.1. Wir geben zwei Formeln vor:

$$\begin{aligned}\varphi_1 &\stackrel{\text{def}}{=} \forall x \exists y K(x, y) \\ \varphi_2 &\stackrel{\text{def}}{=} \exists x \neg \exists y V(y, x)\end{aligned}$$

Als erstes Modell treffen wir folgende Wahl:

Das Universum \mathbb{U} seien die natürlichen Zahlen \mathbb{N} . Die Relation $K(x, y)$ sei definiert als $x < y$ und die Relation $V(y, x)$ als $y = x - 1$.

Dann trifft φ_1 in diesem ersten Modell zu, denn zu jeder natürlichen Zahl x existiert eine größere natürliche Zahl.

Auch die zweite Formel φ_2 ist in diesem Modell wahr, denn für die kleinste natürliche Zahl gibt es keine kleinere solche.

Wir können aber auch ein zweites (ganz anderes) Modell wählen:

Das Universum \mathbb{U} seien die Knoten endlicher binärer Bäume. Die Relation $K(x, y)$ sei definiert als x hat das Kind y und die Relation $V(y.x)$ sei definiert als y ist „Vater“ von x .

In diesem zweiten Modell trifft die Formel φ_1 nicht zu, denn die Blätter eines endlichen Baums haben keine Kinder.

Die zweite Formel φ_2 hingegen ist in diesem Modell wahr. Denn die Wurzel des Baums hat keinen „Vater“.

Beispiel 13.2. In diesem Beispiel wollen wir die Logic Workbench verwenden, um ein Modell zu definieren und Formeln der Prädikatenlogik in diesem Modell auszuwerten. Dazu nehmen wir ein Beispiel aus [Hal15, Chap.5]: Das Universum seien einige europäische Städte. Ferner haben wir zwei Relationen, nämlich $Q(x)$, die ausdrückt, dass die Stadt x auf dem Kontinent ist, sowie $R(x, y)$, die wahr wird, wenn x kleiner ist als y .

In der Logic Workbench könne wir das Modell folgendermaßen definieren:

```
(def cities
  {:univ #{:florence :stockholm :barcelona :london}
   'a    [:func 0 :florence]
   'b    [:func 0 :london]
   'Q    [:pred 1 (make-pred #{{:florence} [:stockholm] [:barcelona]})]
   'R    [:pred 2 (make-pred #{{:florence :stockholm}
                               [:florence :barcelona]
                               [:florence :london]
                               [:stockholm :barcelona]
                               [:stockholm :london]
                               [:barcelona :london]})])
  })
```

Das Universum besteht aus den vier angegebenen Städten. Wir haben zwei Konstanten a und b für Florenz und London. Das Makro `make-pred` verwenden wir, um die beiden Prädikate Q und R zu definieren.

Das Modell gegeben, können wir nun Formeln auswerten:

```
(def phi1 '(R a b))

(eval-phi phi1 cities)
; => true -- (R :florence :london)

(eval-phi '(R b a) cities)
; => false
```

```
(def phi2 '(forall [x] (impl (Q x) (R x b)))))

(eval-phi phi2 cities)
; => true -- all of our cities on the continent are smaller
; than London

(def phi3 '(forall [x] (exists [y] (or (R x y) (R y x)))))

(eval-phi phi3 cities)
; => true
```

13.2 Semantische Folgerung und Äquivalenz

Definition 13.5 (Semantische Folgerung). Sei Γ eine Menge prädikatenlogischer Formeln und φ eine prädikatenlogische Formel. Man sagt:

$\Gamma \vDash \varphi$ d.h. φ folgt semantisch aus Γ , genau dann wenn jedes Modell für Γ auch ein Modell für φ ist.

Besteht Γ nur aus einer Formel, sage ψ , dann schreibt man auch $\psi \vDash \varphi$.

Definition 13.6 (Semantische Äquivalenz). Zwei Formeln φ und ψ sind semantisch äquivalent, geschrieben $\varphi \equiv \psi$, genau dann, wenn $\varphi \vDash \psi$ und $\psi \vDash \varphi$ gilt.

In der Prädikatenlogik können wir nun dieselben Definitionen wie in der Aussagenlogik machen:

Definition 13.7 (Erfüllbarkeit). Eine prädikatenlogische Formel φ heißt erfüllbar, wenn sie ein Modell hat.

Definition 13.8 (Falsifizierbarkeit). Eine prädikatenlogische Formel φ heißt falsifizierbar, wenn es ein Modell \mathcal{M} gibt mit $\mathcal{M} \not\vDash \varphi$.

Definition 13.9 (Allgemeingültigkeit). Eine prädikatenlogische Formel φ heißt allgemeingültig, wenn sie in jedem Modell wahr ist.

Man schreibt dann $\models \varphi$ und nennt φ eine Tautologie.

Definition 13.10 (Unerfüllbarkeit). Eine prädikatenlogische Formel φ heißt unerfüllbar, wenn es kein Modell für sie gibt.

Man schreibt dann $\not\models \varphi$ und nennt φ eine Kontradiktion.

Auch in der Prädikatenlogik gilt das Dualitätsprinzip:

Satz 13.1 (Dualitätsprinzip). Eine prädikatenlogische Formel φ ist genau dann allgemeingültig, wenn $\neg\varphi$ unerfüllbar ist.

13.3 Fundamentale Äquivalenzen der Prädikatenlogik

Satz 13.2. φ , ψ und χ seien Formeln der Prädikatenlogik, wobei in χ die Variable x nicht vorkommt.

Es gelten folgende (semantische) Äquivalenzen:

$$\begin{aligned}
 \neg(\forall x \varphi) &\equiv \exists x (\neg\varphi) \\
 \neg(\exists x \varphi) &\equiv \forall x (\neg\varphi) \\
 \forall x \varphi \wedge \forall x \psi &\equiv \forall x (\varphi \wedge \psi) \\
 \exists x \varphi \vee \exists x \psi &\equiv \exists x (\varphi \vee \psi) \\
 \forall x (\forall y \varphi) &\equiv \forall y (\forall x \varphi) \\
 \exists x (\exists y \varphi) &\equiv \exists y (\exists x \varphi) \\
 (\forall x \varphi) \wedge \chi &\equiv \forall x (\varphi \wedge \chi) \\
 (\forall x \varphi) \vee \chi &\equiv \forall x (\varphi \vee \chi) \\
 (\exists x \varphi) \wedge \chi &\equiv \exists x (\varphi \wedge \chi) \\
 (\exists x \varphi) \vee \chi &\equiv \exists x (\varphi \vee \chi)
 \end{aligned}$$

Kapitel 14

Natürliches Schließen in der Prädikatenlogik

In Kapitel 5 wurde das Beweissystems des natürlichen Schließens nach Gerhard Gentzen für die Aussagenlogik eingeführt. Es beruht auf Regeln, wie Formeln (syntaktisch) umgeformt werden dürfen, um aus gegebenen Aussagen Schlussfolgerungen herzuleiten.

Ein wichtiges Ergebnis war dabei die Vollständigkeit des natürlichen Schließens als Beweissystem für die Aussagenlogik, d.h. dass für eine Menge Γ von gegebenen Formeln und eine Formel φ der Aussagenlogik gilt:

$$\Gamma \vdash \varphi \Leftrightarrow \Gamma \vDash \varphi$$

Die „syntaktische Sicht“ und die „semantische Sicht“ sind also äquivalent und wir können zwischen ihnen wechseln, je nachdem welche für eine konkrete Fragestellung besser geeignet ist.

Das Beweissystem des natürlichen Schließens hat Gerhard Gentzen für die Prädikatenlogik (mit Gleichheit) eingeführt. In diesem Kapitel werden die Regeln für die Quantoren und die Gleichheit vorgestellt.

Der Satz über die Vollständigkeit des natürlichen Schließens gilt auch für die Prädikatenlogik. Dies wurde zuerst von Kurt Gödel¹ in seiner Dissertation 1929 für das formale System gezeigt, das Bertrand Russell² und Alfred North Whitehead³ in ihrem Buch *Principia Mathematica* verwendet haben und das dieselbe Ausdruckskraft hat wie die Prädikatenlogik.⁴

¹ Kurt Gödel (1906–1978), österreichisch-amerikanischer Logiker.

² Bertrand Russell (1872–1970), britischer Philosoph, Mathematiker und Logiker.

³ Alfred North Whitehead (1861–1947), britischer Philosoph und Mathematiker.

⁴ „Der Hauptgegenstand der folgenden Untersuchungen ist der Beweis der Vollständigkeit des in Russell, *Principia mathematica*, P. I, Nr. 1 und Nr. 10, und ähnlich in Hilbert-Ackermann, *Grundzüge der theoretischen Logik* (zitiert als H. A.), Ill,

Gerhard Gentzen hat ihn für sein Beweissystem des Sequenzenkalküls bewiesen. Man muss den Vollständigkeitssatz so begreifen, dass er nicht für ein spezielles Beweissystem gilt, sondern eine Eigenschaft der Logik selbst ist, vorausgesetzt natürlich, dass das Beweissystem „vernünftig“ definiert ist.

14.1 Schlussregeln

Zusätzlich zu den Regeln des natürlichen Schließens der Aussagenlogik benötigen wir für das Beweissystem in der Prädikatenlogik die Regeln für die Quantoren und die Gleichheit.

14.1.1 Allquantor

	<i>Einführung</i>	<i>Elimination</i>
\forall	x_0 \vdots $\varphi[x_0/x]$	$\forall x \varphi$ $\frac{\forall x \varphi}{\varphi[t/x]}$ $\forall x e$ $\forall x \varphi$

Um den Allquantor einzuführen, hat man folgende Beweisverpflichtung: Gegeben sei ein beliebiges Objekt x_0 des Universums. Man muss dann zeigen, dass die Formel φ mit x_0 an Stelle der Variablen x gilt (dies schreibt man kurz als $\varphi[x_0/x]$). Dabei darf in dieser Herleitung keinerlei spezielle Eigenschaft von x_0 vorkommen, denn x_0 steht ja für ein *beliebiges* Objekt des Universums. Man sagt auch, dass x_0 ein *frisches* beliebiges Objekt ist, sein Name darf somit nicht außerhalb der Box vorkommen.

Die Entfernung des Allquantors ist ein naheliegender Schritt: Wenn φ für alle x gilt, dann kann man ein beliebiges konkretes t des Universums an Stelle von x in die Formel φ einsetzen. In der Substitutionen $\varphi[t/x]$ muss t frei für x in der Formel φ sein, d.h. keine freie Variable y in t darf durch das Einsetzen von x in φ in den Bereich eines Quantors $\forall y$ oder $\exists y$ gelangen.

§ 5, angegebenen Axiomensystems des sogenannten engeren Funktionenkalküls.“
[\[Gö29\]](#)

14.1.2 Existenzquantor

	<i>Einführung</i>	<i>Elimination</i>
∃	$\frac{\varphi[t/x]}{\exists x\varphi}$ ∃ x i	$\frac{\exists x\varphi \quad \begin{array}{ c c } \hline x_0 & \varphi[x_0/x] \\ \vdots & \chi \\ \hline \end{array}}{\chi}$ ∃ x e

Den Existenzquantor kann man einführen, indem man einen *Zeugen* vorweist: Gilt φ mit t an Stelle von x , dann gibt es offenbar ein x für das φ gilt, nämlich eben t .

Will man den Existenzquantor entfernen, muss man ein Objekt x_0 nehmen, das φ an Stelle von x erfüllt. Solch ein Objekt existiert, weil ja $\exists x\varphi$ gilt. Nun hat man die Beweisverpflichtung zu zeigen, dass daraus χ herleitbar ist. In dieser Herleitung darf man keine spezielle Aussage über x_0 verwenden, außer $\varphi[x_0/x]$.

14.1.3 Gleichheit

	<i>Einführung</i>	<i>Elimination</i>
=	$\frac{}{t = t}$ = i, ID	$\frac{t_1 = t_2 \quad \varphi[t_1/x]}{\varphi[t_2/x]}$ = e, SUB

Die Regel ID besagt, dass ein Symbol, das für ein Objekt steht, dieses eindeutig bestimmt. Dies ist gewissermaßen die Charakteristik der Gleichheit.

Die Entfernung der Gleichheit besteht darin, dass wenn t_1 und t_2 gleich sind, man in einer Formel φ t_1 durch t_2 ersetzen kann. Dies klingt wie selbstverständlich, muss aber mit Vorsicht gehandhabt werden. Es sind nur gültige Substitutionen erlaubt: In allen Substitutionen $\varphi[t/x]$ muss t frei für x in der Formel φ sein, d.h. keine freie Variable y in t gelangt durch das Einsetzen von x in φ in den Bereich eines Quantors $\forall y$ oder $\exists y$.

14.1.4 „Konstantenquantifizierung“

Im Folgenden schreiben wir $\varphi(x)$ für eine Formel, die als einzige freie Variable x hat und $\varphi(c)$ für die Formel, bei der in $\varphi(x)$ die Variable durch die Konstante c ersetzt wurde.

Man kann mit der Regel der Einführung des Allquantors $\forall i$ folgendes Lemma zeigen:

Lemma 14.1. Sei Γ eine Formelmenge und $\varphi(x)$ eine Formel, die die Konstante c nicht enthalten und es gebe eine Herleitung $\Gamma \vdash \varphi(c)$. Dann gibt es auch eine Herleitung $\Gamma \vdash \forall x\varphi(x)$.

Beweis. Die gegebene Herleitung $\Gamma \vdash \varphi(c)$ kann man in die folgende Herleitung einsetzen:

- | | | |
|----|-----------------------|---|
| 1. | Γ | gegebene Formeln |
| 2. | c | beliebig |
| 3. | ... | von $\Gamma \vdash \varphi(c)$ übernommen |
| 4. | $\varphi(c)$ | ... |
| 5. | $\forall x\varphi(x)$ | $\forall i$ |

Das „Einsetzen“ ist möglich, weil nach Voraussetzung die Konstante c nicht in Γ vorkommt, das heißt, sie hat keinerlei spezielle Eigenschaften, sondern ist beliebig. \square

14.2 Beispiele

Gentzen zeigt in [Gen35, S. 183] an drei Beispielen, wie das natürliche Schließen geht. Das erste Beispiel für eine Formel der Aussagenlogik wurde in Abschnitt 5.2 verwendet.

Für die Prädikatenlogik verwendet Gentzen die beiden folgenden Beispiele:

14.2.0.1 Beispiel: Vertauschen von Quantoren

Herleitung für

$$\exists x \forall y F(x, y) \rightarrow \forall y \exists x F(x, y)$$

- | | | |
|----|---|---------------------|
| 1. | $\exists x \forall y F(x, y)$ | angenommen |
| 2. | a | angenommen |
| 3. | $\forall y F(a, y)$ | angenommen |
| 4. | b | beliebig |
| 5. | $F(a, b)$ | $\forall e 3, 4$ |
| 6. | $\exists x F(x, b)$ | $\exists i 2, 5$ |
| 7. | $\forall y \exists x F(x, y)$ | $\forall i 4-6$ |
| 8. | $\forall y \exists x F(x, y)$ | $\exists e 1, 2-7$ |
| 9. | $\exists x \forall y F(x, y) \rightarrow \forall y \exists x F(x, y)$ | $\rightarrow i 1-8$ |

14.2.0.2 Negation und Quantoren

Als weiteres Beispiel folgt ein Beweis für

$$\neg \exists x G(x) \rightarrow \forall y \neg G(y)$$

Der Beweis folgt wieder der Argumentation von Gentzen in [Gen35, S. 183]:

1.	$\neg \exists x G(x)$	angenommen
2.	a	beliebig
3.	$G(a)$	angenommen
4.	$\exists x G(x)$	$\exists i 2, 3$
5.	\perp	$\neg e 1, 4$
6.	$\neg G(a)$	$\neg i 3-5$
7.	$\forall y \neg G(y)$	$\forall i 2-6$
8.	$\neg \exists x G(x) \rightarrow \forall y \neg G(y)$	$\rightarrow i 1-7$

14.2.0.3 Die verallgemeinerten Gesetze von De Morgan

In diesem Abschnitt verwenden wir das natürliche Schließen in der Logic WorkBench lwb, um die verallgemeinerten Gesetze von De Morgan herzuleiten.

Es geht um folgende Biimplikationen:

$$\begin{aligned} \vdash \neg \forall x \varphi &\leftrightarrow \exists x \neg \varphi \\ \vdash \neg \exists x \varphi &\leftrightarrow \forall x \neg \varphi \end{aligned}$$

Da die Regeln für das natürliche Schließen nur eine Regel für die Einführung bzw. Auflösung der Implikation, nicht jedoch der Biimplikation enthalten, müssen wir je Gesetz beide Richtungen herleiten.

Tatsächlich beweisen wir in lwb die Gesetze für unäre Prädikate. Da die Bezeichnung der Variablen und der Prädikate in der Herleitung aber keine Rolle spielt, kann man die Gesetze in lwb für beliebige Variablen und Prädikate anwenden (siehe auch [HR04, Section 2.3]).

- $\vdash \neg \forall x \varphi \rightarrow \exists x \neg \varphi$

Die Herleitung hat in lwb die folgenden Schritte:

```

1 (proof '(not (forall [x] (P x))) '(exists [x] (not (P x))))
2 (step-b :raa 3)
3 (step-b :not-e 4 1)
4 (step-b :forall-i 4)
5 (swap '?1 :i)
6 (step-b :raa 5)
7 (step-b :not-e 6 2)
8 (step-b :exists-i 6 3)

```

In Schritt 1 wird die Beweisverpflichtung formuliert.

```

(proof '(not (forall [x] (P x))) '(exists [x] (not (P x))))
-----
1: (not (forall [x] (P x))) :premise
2: ...
3: (exists [x] (not (P x)))
-----
```

Wir beginnen mit Schritt 2 einen Widerspruchsbeweis mit der Regel RAA, die in lwb als :raa geschrieben wird und rückwärts (step-b) auf Zeile 3 angewandt wird.

```

(step-b :raa 3)
-----
1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
3: | ...
4: | contradiction
-----
5: (exists [x] (not (P x))) :raa [[2 4]]
-----
```

Der dritte Schritt besteht darin, dass wir mit Zeile 4 und 1 die Regel für die Auflösung von \neg anwenden, was uns in folgende Situation bringt:

```

(step-b :not-e 4 1)
-----
1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
3: | ...
4: | (forall [x] (P x))
5: | contradiction :not-e [1 4]
-----
6: (exists [x] (not (P x))) :raa [[2 5]]
-----
```

Die Beweisverpflichtung besteht nun darin, den Allquantor in Zeile 4 einzuführen, dies bereiten wir im 4. Schritt vor:

```
(step-b :forall-i 4)
```

ergibt:

```
-----
1: (not (forall [x] (P x))) :premise
| -----
2: | (not (exists [x] (not (P x)))) :assumption
| |
3: | | (actual ?1) :assumption
4: | | ...
5: | | (P ?1)
| |
6: | (forall [x] (P x)) :forall-i [[3 5]]
7: | contradiction :not-e [1 6]
| -----
8: (exists [x] (not (P x))) :raa [[2 7]]
-----
```

Für die Einführung des Allquantors brauchen wir ein beliebiges Objekt des Universums. In Zeile 3 wurde uns ein solches beliebiges Objekt bereitgestellt, es hat noch keinen Namen, sondern wird markiert mit ?1. In Schritt 5 ersetzen wir ?1 durch einen Namen für das Objekt des Universums, wir nennen es :i.

```
(swap '?1 :i)
```

ergibt:

```
-----
1: (not (forall [x] (P x))) :premise
| -----
2: | (not (exists [x] (not (P x)))) :assumption
| |
3: | | (actual :i) :assumption
4: | | ...
5: | | (P :i)
| |
6: | (forall [x] (P x)) :forall-i [[3 5]]
7: | contradiction :not-e [1 6]
| -----
8: (exists [x] (not (P x))) :raa [[2 7]]
-----
```

Also müssen wir nun für das beliebige :i zeigen, dass (P :i) gilt. Und noch einmal die Regel für den Widerspruchsbeweis im 6. Schritt:

```
(step-b :raa 5)
```

ergibt:

```
-----
1: (not (forall [x] (P x))) :premise
| -----
2: | (not (exists [x] (not (P x)))) :assumption
| |
3: | | (actual :i) :assumption
| |
4: | | | (not (P :i)) :assumption
5: | | | ...
6: | | | contradiction
| |
7: | | (P :i) :raa [[4 6]]
| -----
8: | (forall [x] (P x)) :forall-i [[3 7]]
9: | contradiction :not-e [1 8]
| -----
10: (exists [x] (not (P x))) :raa [[2 9]]
-----
```

Wir kommen dem Ziel näher, denn in Zeile 2 der Herleitung haben wir ja die Aussage, dass kein Objekt x existiert für das $\neg P(x)$ gilt.

Also Schritt 7:

```
(step-b :not-e 6 2)
```

ergibt:

```
-----
1: (not (forall [x] (P x))) :premise
| -----
2: | (not (exists [x] (not (P x)))) :assumption
| |
3: | | (actual :i) :assumption
| |
4: | | | (not (P :i)) :assumption
5: | | | ...
6: | | | (exists [x] (not (P x)))
7: | | | contradiction :not-e [2 6]
| |
8: | | (P :i) :raa [[4 7]]
| -----
9: | (forall [x] (P x)) :forall-i [[3 8]]
10: | contradiction :not-e [1 9]
| -----
11: (exists [x] (not (P x))) :raa [[2 10]]
-----
```

Nun bleibt nur noch in Schritt 8 die Einführung des Existenzquantors durch die entsprechende Regel abzusichern:

```

-----
1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
| |
3: | | (actual :i) :assumption
| |
4: | | | (not (P :i)) :assumption
5: | | | (exists [x] (not (P x))) :exists-i [3 4]
6: | | | contradiction :not-e [2 5]
| |
7: | | (P :i) :raa [[4 6]]
| |
8: | (forall [x] (P x)) :forall-i [[3 7]]
9: | contradiction :not-e [1 8]
-----
10: (exists [x] (not (P x))) :raa [[2 9]]
-----
```

Für die drei weiteren Herleitung geben wir jeweils die Beweisschritte und das Ergebnis an:

- $\vdash \exists x \neg \varphi \rightarrow \neg \forall x \varphi$

Die Schritte der Herleitung:

```

1 (proof '(exists [x] (not (P x))) '(not (forall [x] (P x))))
2 (step-b :not-i 3)
3 (step-f :exists-e 1 4)
4 (swap '?1 :i)
5 (step-f :forall-e 2 3)
6 (step-f :not-e 4 5)
```

Das Ergebnis:

```

-----
1: (exists [x] (not (P x))) :premise
-----
2: | (forall [x] (P x)) :assumption
| |
3: | | (actual :i) :assumption
4: | | (not (P :i)) :assumption
5: | | (P :i) :forall-e [2 3]
6: | | contradiction :not-e [4 5]
| |
7: | contradiction :exists-e [1 [3 6]]
| |
8: (not (forall [x] (P x))) :not-i [[2 7]]
-----
```

- $\vdash \neg \exists x \varphi \rightarrow \forall x \neg \varphi$

Die Schritte der Herleitung:

```

1 (proof '(not (exists [x] (P x))) '(forall [x] (not (P x))))
2 (step-b :forall-i 3)
3 (swap '?1 :i)
4 (step-b :not-i 4)
5 (step-b :not-e 5 1)
6 (step-b :exists-i 5 2)

```

Das Ergebnis:

```

-----
1: (not (exists [x] (P x))) :premise
-----
2: | (actual :i) :assumption
| -----
3: | | (P :i) :assumption
4: | | (exists [x] (P x)) :exists-i [2 3]
5: | | contradiction :not-e [1 4]
| -----
6: | (not (P :i)) :not-i [[3 5]]
-----
7: (forall [x] (not (P x))) :forall-i [[2 6]]
-----
```

- $\vdash \forall x \neg \varphi \rightarrow \neg \exists x \varphi$

Die Schritte der Herleitung:

```

1 (proof '(forall [x] (not (P x))) '(not (exists [x] (P x))))
2 (step-b :not-i 3)
3 (step-f :exists-e 2 4)
4 (swap '?1 :i)
5 (step-f :forall-e 1 3)
6 (step-f :not-e 5 4)

```

Das Ergebnis:

```

-----
1: (forall [x] (not (P x))) :premise
-----
2: | (exists [x] (P x)) :assumption
| -----
3: | | (actual :i) :assumption
4: | | (P :i) :assumption
5: | | (not (P :i)) :forall-e [1 3]
6: | | contradiction :not-e [5 4]
| -----
7: | contradiction :exists-e [2 [3 6]]
-----
8: (not (exists [x] (P x))) :not-i [[2 7]]
-----
```

14.3 Vollständigkeit des natürlichen Schließens

Wie beim natürlichen Schließen in der Aussagenlogik stellen sich mit den zusätzlichen Regeln für die Prädikatenlogik mit Gleichheit die zwei Fragen:

- Sind die Regeln wahrheitserhaltend, leiten sie aus wahren Formeln auch immer nur wahre Formeln her? Dies ist die Frage nach der *Korrektheit* des Beweissystems:

$$\Gamma \vdash \varphi \Rightarrow \Gamma \vDash \varphi$$

- Sind alle Regeln vorhanden, die es gestatten jeden zutreffenden Zusammenhang auch tatsächlich herzuleiten? Dies ist die Frage nach der *Vollständigkeit* des Beweissystems:

$$\Gamma \vDash \varphi \Rightarrow \Gamma \vdash \varphi$$

14.3.1 Die Korrektheit des natürlichen Schließens in der Prädikatenlogik

Satz 14.1 (Korrektheit). *Gegeben eine Menge Γ von geschlossenen Formeln sowie eine geschlossene Formel φ . Dann gilt:*

$$\Gamma \vdash \varphi \Rightarrow \Gamma \vDash \varphi$$

Beweisskizze. Der Beweis geht durch Induktion über die Länge der gegebenen Herleitung. Man kann also voraussetzen, dass die Aussage für kürzere Herleitungen zutrifft. Also muss man nur zeigen, dass der jeweils letzte Schritt wahrheitserhaltend ist.

Dazu muss man sich davon überzeugen, dass dieses für jede der Regeln des natürlichen Schließens zutrifft, denn jede könnte ja die letzte der in der Herleitung verwendeten Regeln sein.

Was die Regeln für \neg, \wedge usw. angeht, haben wir dies schon beim Beweis der Korrektheit des natürlichen Schließens für die Aussagenlogik gesehen. Also muss man nur noch die Regeln für die Quantoren und die Gleichheit untersuchen. Nun sind die Regeln aber eben so konstruiert, dass dies der Fall ist (für einen systematischen und detaillierten Beweis dafür siehe [vD13, Lemma 3.8.2]). \square

14.3.2 Die Vollständigkeit des natürlichen Schließens in der Prädikatenlogik

In der Aussagenlogik ist es uns gelungen einen Beweis für eine Tautologie zu konstruieren, in dem wir in der Wahrheitstafel alle Modelle für die

Formel betrachtet haben und durch die fortwährende Anwendung des Regel TND eine Situation geschaffen haben, wo wir aus jeder Zeile der Wahrheitstafel, also aus jedem möglichen Modell die Formel durch die Regeln des natürlichen Schließens herzuleiten hatten.

Ein solches Vorgehen ist in der Prädikatenlogik nicht möglich, weil wir ja unmöglich alle Modelle hinschreiben könnten. Also müssen wir den Weg des indirekten Beweises gehen und wie im Fall der Aussagenlogik nehmen wir für den Beweis von $\Gamma \vDash \varphi \rightarrow \Gamma \vdash \varphi$ an, dass die Folgerung nicht richtig ist. Wir wissen dann, dass $\Gamma \cup \{\neg\varphi\}$ konsistent ist. Können wir nun aus dieser konsistenten Menge von Formeln ein Modell herleiten, dann haben wir gezeigt, dass $\Gamma \vDash \neg\varphi$ gilt, was der Voraussetzung widerspricht. Ganz analog zum indirekten Beweis des Vollständigkeitssatz in der Aussagenlogik benötigen wir also den Modellexistenzsatz, der besagt, dass eine konsistente Menge von geschlossenen Formeln ein Modell hat.

Für den Beweis wird die sogenannte Henkin-Konstruktion verwendet nach Leon Henkin⁵. In Lehrbüchern findet man viele Varianten des Beweise, die meist die Henkin-Konstruktion in der einen oder anderen Form verwenden.

Die Grundidee besteht darin, dass man als Universum des zu konstruierenden Modells die variablenfreien Terme der Sprache selbst nimmt und sich so wie Münchhausen am eigenen Schopf aus dem Sumpf zieht.⁶

Das geht aber nicht so ohne weiteres. Es könnte ja sein, dass die gegebene Sprache der Prädikatenlogik gar keine Konstanten hat, die Menge der variablenfreien Terme also leer wäre. Oder wir hätten eine Sprache mit einer Konstanten c und einem unären Prädikat P und Γ wäre die Formelmenge $\{\exists x P(x), \neg P(c)\}$. Diese Formelmenge ist konsistent, aber die Menge der variablenfreien Terme ist $\{c\}$. Da ja aber $\neg P(c) \in \Gamma$ gibt es kein Modell mit diesem Universum für Γ .

Dieses Beispiel motiviert eine Idee in der Henkin-Konstruktion: Für jede Formel der Form $\exists\varphi(x)$ (wir erinnern uns, dass in dieser Schreibweise x die einzige freie Variable in φ ist) muss es eine Konstante in der Sprache geben. Wir müssen die gegebene Sprache also um solche Konstanten erweitern. Man nennt diese Konstanten auch die „Zeugen“ für die Existenzaussage.

Einen weiteren Aspekt muss man berücksichtigen. Wir betrachten Logik mit Gleichheit. Deshalb kann es natürlich vorkommen, dass wir verschieden ausgedrückte Terme haben, die aber tatsächlich identisch sind.

⁵ Leon Henkin (1921–2006), amerikanischer Logiker.

⁶ Hieronymus Carl Friedrich Freiherr von Münchhausen (1720 - 1797) werden die Geschichten vom Baron Münchhausen zugeschrieben.

Stellen wir uns etwa vor, wir haben in unserer Sprache die Konstante 0 und zwei Funktionen, die unäre Funktion s sowie eine binäre Funktion p . Dann können wir zum Beispiel die Terme $s(0)$ und $p(0, s(0))$ bilden. Entspricht nun die Konstante 0 der natürlichen Zahl 0, die Funktion s der Nachfolger-Funktion und die Funktion p der Addition, dann sind die beiden Terme identisch. Das bedeutet, dass wir in der Konstruktion des Modells nicht die variablenfreien Terme als Elemente des Universums nehmen können, sondern Äquivalenzklassen (oder Repräsentanten davon) von beweisbar identischen Termen.

14.3.3 Der Modellexistenzsatz

Wir setzen in Folgenden voraus, dass unsere Sprache abzählbar ist, es demzufolge abzählbar viele geschlossene Formeln gibt. Der Modellexistenzsatz gilt auch ohne diese Einschränkung, benötigt dann aber das Wohlordnungsaxiom, siehe [Hed04, Abschnitt 4.3].

Satz 14.2 (Modellexistenzsatz). *Sei Γ eine konsistente Menge geschlossener Formeln der Sprache \mathcal{L} , dann gibt es ein Modell \mathcal{M} für Γ , d.h. $\mathcal{M} \models \Gamma$.*

Beweis.

Schritt 1 Erweiterung von Γ zu einer *maximal* konsistenten Menge Γ^* mit „Zeugen“.

Wir erweitern zunächst die Sprache \mathcal{L} um eine Menge $C = \{c_1, c_2, c_3, \dots\}$ frischer Konstanten.

Nun sei $\{\varphi_1, \varphi_2, \varphi_3, \dots\}$ die Menge der Sätze, die man in $\mathcal{L} \cup C$ bilden kann.

Wir bilden nun Γ^* durch folgenden Konstruktionsprozess, der Lindenbaums Lemma für die Prädikatenlogik darstellt:

Sei $\Gamma_0 = \Gamma$. Angenommen wir haben Γ_m bereits konstruiert, und zwar so, dass Γ_m konsistent ist und nur endlich viele Konstanten aus C in Γ_m vorkommen. Nun konstruieren wir Γ_{m+1} indem wir die Formel φ_{m+1} betrachten. Dabei können folgende Fälle auftreten:

- (1) $\Gamma_m \cup \{\varphi_{m+1}\}$ ist *nicht* konsistent. Dann sei $\Gamma_{m+1} = \Gamma_m \cup \{\neg\varphi_{m+1}\}$.
- (2) $\Gamma_m \cup \{\varphi_{m+1}\}$ ist konsistent und φ_{m+1} hat nicht die Form $\exists x\theta(x)$. Dann sei $\Gamma_{m+1} = \Gamma_m \cup \{\varphi_{m+1}\}$.
- (3) $\Gamma_m \cup \{\varphi_{m+1}\}$ ist konsistent und φ_{m+1} hat die Form $\exists x\theta(x)$. Dann sei $\Gamma_{m+1} = \Gamma_m \cup \{\varphi_{m+1}\} \cup \{\theta(c_i)\}$ für ein c_i , das in $\Gamma_m \cup \{\varphi_{m+1}\}$ nicht vorkommt.

Γ_{m+1} hat höchstens zwei Formeln mehr als Γ_m , also endlich viele Konstanten aus C , weil dies in Γ_m auch nur endlich viele waren.

Es muss nun gezeigt werden, dass Γ_{m+1} konsistent ist:

Ist Γ_{m+1} durch die Fälle (1) oder (2) entstanden, dann ist Γ_{m+1} qua Konstruktion konsistent. Also müssen wir noch Fall (3) analysieren.

Angenommen Γ_{m+1} wäre im Fall (3) *nicht* konsistent.

Das bedeutet

$$\Gamma_{m+1} \vdash \perp$$

d.h.

$$\Gamma_m, \exists x \theta(x), \theta(c_i) \vdash \perp$$

Daraus kann man mit der Regel $\neg i$ folgende Herleitung machen

$$\Gamma_m, \exists x \theta(x) \vdash \neg \theta(c_i)$$

Da c_i in den Voraussetzungen nicht vorkommt, können wir die „Konstantenquantifizierung“ (Lemma 14.1) machen

$$\Gamma_m, \exists x \theta(x) \vdash \forall x \neg \theta(x)$$

Andererseits gilt aber

$$\exists x \theta(x) \vdash \neg \forall x \neg \theta(x)$$

Dies ist ein Widerspruch, der die Annahme, Γ_{m+1} sei nicht konsistent, widerlegt.

Also haben wir eine Kette konsistenter Satzmengen $\Gamma_0 \subset \Gamma_1 \subset \Gamma_2 \subset \dots$. Wir setzen

$$\Gamma^* = \bigcup \{\Gamma_n \mid n \geq 0\}$$

Γ^* hat folgende Eigenschaften:

(1) Γ^* ist konsistent.

Wenn Γ^* nicht konsistent wäre, gäbe es eine *endliche* Herleitung von \perp aus Γ^* . D.h. eine endliche Teilmenge von Γ^* würde genügen, diese müsste aber in einer der Mengen Γ_m enthalten sein, die aber konsistent sind, also niemals den Schluss auf \perp erlauben.

(2) Γ^* ist maximal konsistent.

Sei $\Gamma^* \subseteq \Delta$ mit konsistenter Obermenge Δ . Wenn $\psi \in \Delta$, dann ist ψ ja einer Formel der Sprache $\mathcal{L} \cup C$, also eines der φ_m , d.h. $\Gamma^* = \Delta$.

(3) Jede geschlossene Formel in Γ ist in Γ^* .

Das ist der Fall, weil ja $\Gamma = \Gamma_0$ der Startpunkt der Konstruktion war.

(4) Für jeden Satz in Γ^* der Form $\exists x\theta(x)$ ist auch $\theta(c_i)$ für ein $c_i \in C$ in Γ^* , d.h. für jede dieser Existenzaussagen gibt es einen „Zeugen“.

Schritt 2 Das Universum des Modells \mathcal{M} für Γ^*

Das Universum des Modells für Γ^* sei die Menge aller variablenfreier Terme der Sprache $\mathcal{L} \cup C$, allerdings modulo Gleichheit.

Dabei sind zwei Terme gleich, wenn diese Aussage in Γ^* vorkommt, d.h. zwei Terme t_1 und t_2 sind gleich, wenn $\Gamma^* \vdash t_1 = t_2$ gilt. Da Γ^* maximal konsistent und folglich negationstreu ist (ganz analog in der Prädikatenlogik zu Lemma 5.6), gilt für jedes Paar t_1, t_2 von Termen entweder $\Gamma^* \vdash t_1 = t_2$ oder $\Gamma^* \vdash \neg(t_1 = t_2)$.

Es ist leicht zu überprüfen, dass dadurch eine Äquivalenzrelation definiert ist.

Im Folgenden gehen wir davon aus, dass mit einem Term immer ein fixer Repräsentant seiner Äquivalenzklasse gemeint ist.

Schritt 3 Definition des Modells \mathcal{M} für Γ^* , auch *Term-Modell* genannt.

Für jede Konstante c unserer Sprache gibt es einen eindeutigen Term t im Universum mit der Eigenschaft $\Gamma^* \vdash t = c$.

Für jede n -äre Relation R der Sprache und ein Tupel t_1, \dots, t_n von Elementen des Universums setzen wir

$$\mathcal{M} \models R(t_1, \dots, t_n) \Leftrightarrow \Gamma^* \vdash R(t_1, \dots, t_n)$$

Für jede n -äre Funktion f der Sprache, jedes Element s und ein Tupel t_1, \dots, t_n von Elementen des Universums setzen wir

$$\mathcal{M} \models f(t_1, \dots, t_n) = s \Leftrightarrow \Gamma^* \vdash f(t_1, \dots, t_n) = s$$

Schritt 4 Das so konstruierte \mathcal{M} ist in der Tat ein Modell für Γ^* .

Dazu müssen wir zeigen, dass für jede geschlossene Formel φ gilt:

$$\mathcal{M} \models \varphi \Leftrightarrow \Gamma^* \vdash \varphi.$$

Um diesen Nachweis etwas kürzer zu machen, gehen wir davon aus, dass nur die logischen Symbole $\neg, \wedge, \exists, =$ in φ vorkommen. Der Beweis geht als Induktion über die Zahl n des Vorkommens dieser Symbole in φ .

Wenn keines der Symbole in φ vorkommt, dann handelt es sich um eine Primformel und die zu beweisende Aussage gilt wegen der Definition von \mathcal{M} .

Wir nehmen nun an, dass die Aussage für alle Sätze gilt, in denen es n oder weniger Vorkommen der logischen Symbole hat und wir betrachten einen Satz mit $n + 1$ Vorkommen.

Wenn φ die Form $\varphi_1 \wedge \varphi_2$ hat, dann gilt $\Gamma^* \vdash \varphi \Leftrightarrow \Gamma^* \vdash \varphi_1 \text{ und } \Gamma^* \vdash \varphi_2$, da Γ^* maximal konsistent ist. Das ist aber nach Induktionsvoraussetzung genau dann der Fall, wenn $\mathcal{M} \models \varphi_1$ und $\mathcal{M} \models \varphi_2$ gilt. Dann gilt aber auch $\mathcal{M} \models \varphi$.

Wenn φ die Form $\neg\varphi_1$ hat, dann gilt $\Gamma^* \vdash \varphi \Leftrightarrow \Gamma^* \not\vdash \varphi_1$, wieder weil Γ^* maximal konsistent ist. Nach Induktionsvoraussetzung ist dann $\mathcal{M} \not\models \varphi_1$, was aber wegen der Semantik von \neg gerade äquivalent zu $\mathcal{M} \models \varphi$ ist.

Nun betrachten wir noch den Fall, dass φ die Form $\exists x\varphi(x)$ hat. Hier verwenden wir nun Eigenschaft (4) von Γ^* , die Existenz des „Zeugen“: $\Gamma^* \vdash \varphi \Leftrightarrow \Gamma^* \vdash \theta(t)$ für einen Term t des Universums. Nach Induktionsvoraussetzung gilt $\Gamma^* \vdash \theta(t) \Leftrightarrow \mathcal{M} \models \theta(t)$. Nach der Semantik des Existenzquantors ist dies aber gerade äquivalent zu $\mathcal{M} \models \varphi$.

Damit ist der Beweis des Modellexistenzsatzes abgeschlossen. \square

Aus dem Modellexistenzsatz folgt die Vollständigkeit des natürlichen Schließens als Beweissystem für die Prädikatenlogik sowie der Kompaktheitssatz:

Satz 14.3 (Kompaktheitssatz). *Eine (abzählbare) Menge von Formeln ist genau dann erfüllbar, wenn jede endliche Teilmenge erfüllbar ist.*

Beweis. Sei Γ eine (abzählbare) Menge von Formeln. Man zeigt, dass Γ genau dann unerfüllbar ist, wenn es eine endliche Teilmenge gibt, die unerfüllbar ist.

Wenn eine endliche Teilmenge von Γ unerfüllbar ist, dann kann selbstredend auch Γ nicht erfüllbar sein.

Die andere Richtung: Sei Γ unerfüllbar, das bedeutet aber nach dem Modellexistenzsatz, dass Γ nicht konsistent ist, d.h. $\Gamma \vdash \perp$. Formale Beweise sind endlich, d.h. es gibt eine endliche Teilmenge Δ und $\Delta \vdash \perp$, wegen der Korrektheit des natürlichen Schließens bedeutet das aber $\Delta \models \perp$ und Δ ist unerfüllbar. \square

Bemerkung. Kompaktheit der Prädikatenlogik (und der Aussagenlogik, wie wir schon mit derselben Argumentation gesehen haben) ist nicht so selbstverständlich, wie es scheinen mag.

Wenn wir uns etwa folgende Aussagen ansehen:

$$\begin{aligned}
 A_0 &\stackrel{\text{def}}{=} \text{„Es gibt nur endlich viele Objekte im Universum“} \\
 A_1 &\stackrel{\text{def}}{=} \text{„Es gibt mindestens ein Objekt im Universum“} \\
 A_2 &\stackrel{\text{def}}{=} \text{„Es gibt mindestens zwei Objekte im Universum“} \\
 A_3 &\stackrel{\text{def}}{=} \text{„Es gibt mindestens drei Objekte im Universum“} \\
 &\dots \quad \dots \\
 A_n &\stackrel{\text{def}}{=} \text{„Es gibt mindestens } n \text{ Objekte im Universum“} \\
 &\dots \quad \dots
 \end{aligned}$$

Dann ist die Menge dieser Aussagen widersprüchlich, jede endliche Teilmenge ist jedoch erfüllbar. Jede Logik, in der man diese Aussagen formal ausdrücken kann, hätte nicht die Eigenschaft der Kompaktheit.

In der Prädikatenlogik lässt sich A_n leicht ausdrücken:

$$\exists x_1 \exists x_2 \dots \exists x_n (\bigwedge_{i < j} \neg(x_i = x_j)),$$

jedoch kann man A_0 nicht ausdrücken, denn man müsste die Zahl der Objekte auf ein *beliebiges* n begrenzen können, nicht auf ein *bestimmtes*.

Übrigens ist es in der Prädikatenlogik auch nicht möglich, den transitiven Abschluss einer (beliebigen) binären Relation auszudrücken. Für Fragestellungen in der Softwaretechnik benötigen wir aber oft Aussagen über den transitiven Abschluss, weshalb die Sprache *Alloy*, die wir als ein Werkzeug der Anwendung der Prädikatenlogik in der Softwaretechnik in Abschnitt 16.2 behandeln, nicht nur die Ausdrucksstärke der Prädikatenlogik hat, sondern auch einen Operator für den transitiven Abschluss.

Auch die Datenbankabfragesprache SQL kann übrigens mit dem Konstrukt `with recursive ...` den transitiven Abschluss einer binären Relation bilden.

Kapitel 15

Unentscheidbarkeit der Prädikatenlogik

Eine der Entscheidungsfragen in der Logik ist die nach der Allgemeingültigkeit einer Formel. Wir haben in der Aussagenlogik gesehen, dass diese Frage *entscheidbar* ist: es gibt einen Algorithmus, der für eine *beliebige* Formel der Aussagenlogik ermitteln kann, ob sie allgemeingültig ist. Wir können als Algorithmus das Aufstellen der Wahrheitstafel nehmen.

In der Prädikatenlogik ist die Sache anders: Das Gültigkeitsproblem in der Prädikatenlogik ist *nicht entscheidbar*. Präziser: es ist *semi-entscheidbar*, was bedeutet, dass es einen Algorithmus gibt, der für eine allgemeingültige Formel ein positives Ergebnis ergibt, bei einer nicht allgemeingültigen Formel jedoch möglicherweise nicht terminiert. Ein solcher Algorithmus wird zum Beispiel in [Sch00, Kapitel 2.4] vorgestellt.

Wir werden in diesem Kapitel die Unentscheidbarkeit des Gültigkeitsproblems der Prädikatenlogik zeigen, in dem wir die Fragestellung (wie in [Sch00, Kapitel 2.3] oder [HR04, Kapitel 2.5]) zurückführen auf ein anderes Entscheidungsproblem, das Postsche Korrespondenzproblem¹, von dem man weiß, dass es unentscheidbar ist, siehe z.B. [Sip13, Chap 5.2].

Aus der Unentscheidbarkeit des Gültigkeitsproblems für die Prädikatenlogik folgt auch die Unentscheidbarkeit des Erfüllbarkeitsproblems. Mit der Vollständigkeit des natürlichen Schließens ergibt sich damit auch, dass es kein Programm geben kann, das für jede beliebige Fragestellung $\Gamma \vdash \varphi?$ eine Herleitung automatisch erstellen kann.

¹ nach **Emil Leon Post** (1897–1954), polnisch-amerikanischer Mathematiker und Logiker.

Das Erfüllbarkeitsproblem für die Prädikatenlogik ist *nicht* semi-entscheidbar. Denn angenommen dies wäre der Fall, dann könnte man Allgemeingültigkeit entscheiden: Für eine Formel φ ist sie selbst allgemeingültig oder ihre Negation $\neg\varphi$ ist erfüllbar. Nun könnte man sowohl das semi-entscheidbare Entscheidungsverfahren für die Allgemeingültigkeit von φ als auch das semi-entscheidbare Verfahren für die Erfüllbarkeit von $\neg\varphi$ durchführen und hätte damit ein Entscheidungsverfahren für die Allgemeingültigkeit von φ , was es aber nicht geben kann, wie in Satz 15.1 gezeigt wird.

15.1 Das Postsche Korrespondenzproblem

Definition 15.1 (Postsches Korrespondenzproblem).

Gegeben eine endliche Folge von Paaren $(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)$ von Strings über dem Alphabet $\{0, 1\}$ positiver Länge, gibt es dann eine Folge von Indices i_1, i_2, \dots, i_n mit $n \geq 1$, so dass die Konkatenation der Strings $s_{i_1} s_{i_2} \dots s_{i_n}$ und $t_{i_1} t_{i_2} \dots t_{i_n}$ identisch sind.

Man kann sich die Paare von Strings als Dominosteine vorstellen. Zum Beispiel könnten wir folgende Steine haben:

$$\left[\begin{array}{c} 1 \\ 101 \end{array} \right] \quad \left[\begin{array}{c} 10 \\ 00 \end{array} \right] \quad \left[\begin{array}{c} 011 \\ 11 \end{array} \right]$$

Man beachte, dass man die Steine mehrfach verwenden kann. Gesucht ist also eine Folge von Dominostenen, so dass der String oben identisch mit dem String unten ist. In unserem Beispiel gibt es folgende Lösung:

$$\left[\begin{array}{c} 1 \\ 101 \end{array} \right] \left[\begin{array}{c} 011 \\ 11 \end{array} \right] \left[\begin{array}{c} 10 \\ 00 \end{array} \right] \left[\begin{array}{c} 011 \\ 11 \end{array} \right]$$

$$\left[\begin{array}{c} 1 \cdot 011 \cdot 10 \cdot 011 \\ 101 \cdot 11 \cdot 00 \cdot 11 \end{array} \right]$$

15.2 Die Unentscheidbarkeit des Gültigkeitsproblems in der Prädikatenlogik

Die Reduktion der Frage nach der Entscheidbarkeit des Gültigkeitsproblems in der Prädikatenlogik auf das Postsche Korrespondenzproblem besteht darin, dass wir letzteres durch eine Formel so ausdrücken, dass $\models \varphi$ genau dann gilt, wenn das durch φ kodierte Korrespondenzproblem eine Lösung hat.

Wir starten mit einem (beliebigen) Korrespondenzproblem C

$$\left[\frac{s_1}{t_1} \right] \left[\frac{s_2}{t_2} \right] \dots \left[\frac{s_k}{t_k} \right]$$

und wollen nun das Problem als eine Formel der Prädikatenlogik darstellen.

Für die Kodierung in der Prädikatenlogik brauchen wir eine Sprache \mathcal{P} mit geeigneter Signatur:

- Wir nehmen eine Konstante e , die für den leeren String steht.
- Ferner sehen wir für unsere Sprache zwei unäre Funktionen f_0 und f_1 vor. Die Idee dahinter ist, dass die Funktion f_0 für das Anfügen der 0 und f_1 der 1 zu einem String ist. Zum Beispiel: $f_1(e)$ ergibt den String 1, $f_0(f_1(e))$ ergibt den String 10 Achtung: umgekehrte Reihenfolge, deshalb vereinbaren wir, dass für einen binären String s die Funktion $f_s(e)$ dadurch definiert ist, dass man die Buchstaben von s nacheinander auf e anwendet.
- Schließlich soll unsere Sprache noch ein binäres Prädikat P haben. Hier ist die Idee, dass das Prädikat zwei Terme s und t als Argumente hat und T wird, wenn es eine Folge von Indices (i_1, i_2, \dots, i_n) gibt und die beiden Terme erfüllen gerade die Anforderung des Postschen Korrespondenzproblems.

Die Sprache \mathcal{P} hat also die Signatur $\{P^2; f_0^1, f_1^1; e\}$.

In dieser Sprache formulieren wir nun eine Formel, die das Postsche Korrespondenzproblem kodiert:

$$\begin{aligned}\varphi_1 &\stackrel{\text{def}}{=} \bigwedge_{i=1}^k P(f_{s_i}(e), f_{t_i}(e)) \\ \varphi_2 &\stackrel{\text{def}}{=} \forall x \forall y \left(P(x, y) \rightarrow \bigwedge_{i=1}^k P(f_{s_i}(x), f_{t_i}(y)) \right) \\ \varphi_3 &\stackrel{\text{def}}{=} \exists z P(z, z) \\ \varphi &\stackrel{\text{def}}{=} \varphi_1 \wedge \varphi_2 \rightarrow \varphi_3\end{aligned}$$

Nun ist zu zeigen:

$\models \varphi \Leftrightarrow$ Das Postsche Korrespondenzproblem C hat eine Lösung.

Beweis.

Wir nehmen an, dass $\models \varphi$ gilt. Das bedeutet, dass die Formel in allen

Modellen der Sprache gilt. Wir müssen also nur ein Modell finden, in dem man die Lösung der Korrespondenzproblems C aus dem Zutreffen der Formel einfach ablesen kann.

Das Universum \mathbb{U} sei die Menge aller endlichen binären Strings. Die Interpretation der Konstanten e sei der leere String. Die beiden Funktionen sind definiert durch Anhängen ihres Index an ihr Argument. Schließlich sei noch die Interpretation von P definiert:

$$\begin{aligned} P^{\mathcal{M}} &\stackrel{\text{def}}{=} \{(s, t) \mid \text{es gibt eine Folge von Indices } (i_1, i_2, \dots, i_n) \text{ so dass} \\ &\quad s = s_{i_1} s_{i_2} \cdots s_{i_n} \text{ und } t = t_{i_1} t_{i_2} \cdots t_{i_n}\} \end{aligned}$$

wobei die s_i und die t_i die Vorgaben aus dem Korrespondenzproblem C sind.

Da φ allgemeingültig ist nach Voraussetzung, gilt auch $\mathcal{M} \models \varphi$. Es gilt auch $\mathcal{M} \models \varphi_2$. Warum? Wenn $P(s, t)$ gilt, dann kann man für jedes $i = 1, 2, \dots, k$ die Folge der Indices einfach verlängern. Außerdem gilt auch $\mathcal{M} \models \varphi_1$, denn $f_{s_i}(e)$ ist ja gerade s_i , für t analog.

Also gilt $\mathcal{M} \models \varphi_1 \wedge \varphi_2 \rightarrow \varphi_3$ sowie $\mathcal{M} \models \varphi_1 \wedge \varphi_2$, also $\mathcal{M} \models \varphi_3$. Das bedeutet aber gerade, dass das Korrespondenzproblem C eine Lösung hat.

Nun ist die umgekehrte Richtung zu zeigen. Wir nehmen also an, dass C eine Lösung hat und müssen zeigen, dass in einem beliebigem Modell \mathcal{M} zu unserer Sprache die Formel φ wahr wird.

Wir machen eine Fallunterscheidung:

Angenommen $\mathcal{M} \not\models \varphi_1 \wedge \varphi_2$, dann gilt $\models \varphi$.

Angenommen also, dass gilt $\mathcal{M} \models \varphi_1 \wedge \varphi_2$. In diesem Fall ist zu zeigen, dass dann auch $\mathcal{M} \models \varphi_3$ gilt. Die Idee ist nun, dass man endliche binäre Strings im Universum von \mathcal{M} , d.h. man definiert eine Abbildung ι induktiv durch

$$\begin{aligned} \iota(e) &\stackrel{\text{def}}{=} e^{\mathcal{M}} \\ \iota(s0) &\stackrel{\text{def}}{=} f_0^{\mathcal{M}}(\iota(s)) \\ \iota(s1) &\stackrel{\text{def}}{=} f_1^{\mathcal{M}}(\iota(s)) \end{aligned}$$

Da es nach Voraussetzung eine Lösung $s = s_{i_1} s_{i_2} \cdots s_{i_n}$ und $t = t_{i_1} t_{i_2} \cdots t_{i_n}$ von C gibt, sind s und t gleich, das bedeutet aber auch ihre Interpretationen sind gleich: $\iota(s) = \iota(t)$ und damit existiert ein Element z des Universums von \mathcal{M} mit $P(z, z)$, nämlich $\iota(s)$. Also $\mathcal{M} \models \varphi_3$ \square

Damit haben wir gezeigt:

Satz 15.1. Das Allgemeingültigkeitsproblem in der Prädikatenlogik ist unentscheidbar. \square

15.3 Der Sonderfall endlicher Universen

Im vorherigen Abschnitt haben wir gesehen, dass Erfüllbarkeit in der Prädikatenlogik unentscheidbar ist. Die Situation sieht anders aus, wenn wir zu einer gegebenen Sprache \mathcal{L} nur Modelle mit endlichen Universen einer vorgegebenen Größe n betrachten.

Zunächst beobachtet man, dass bei einem Universum $\mathbb{U} = \{u_1, u_2, \dots, u_n\}$ die Quantoren durch \wedge bzw. \vee ausgedrückt werden können:

$$\begin{aligned}\forall x \varphi(x) &= \varphi(u_1) \wedge \varphi(u_2) \wedge \dots \wedge \varphi(u_n) \\ \exists x \varphi(x) &= \varphi(u_1) \vee \varphi(u_2) \vee \dots \vee \varphi(u_n)\end{aligned}$$

Was Prädikate angeht, können sie in einem endlichen Universum jeweils durch endlich viele Symbole ausgedrückt werden. Ein einstelliges Prädikat P etwa durch $P_{u_1}, P_{u_2}, \dots, P_{u_n}$, ein zweistelliges Prädikat durch $P_{(u_1, u_1)}, P_{(u_1, u_2)}, \dots, P_{(u_n, u_n)}$ usw.

Auch Funktionen können durch explizite Wertetabellen dargestellt werden, die angeben, zu welchen Elementen des Universums sie auswerten.

Das bedeutet, dass wir jedes der von den Prädikaten herrührenden Symbole als atomare Aussagen auffassen können. Auf diese Weise ist es möglich jede Formel zu einer Formel in der Aussagenlogik zu machen.

Aus dieser Überlegung kann man ein (beschränktes) Widerlegungsverfahren für das Gültigkeitsproblem machen:

Sei φ eine beliebige Formel der Prädikatenlogik. φ ist allgemeingültig, wenn $\neg\varphi$ unerfüllbar ist. Für $n = 1, 2, 3, \dots$ prüfe man $\neg\varphi$ in einem Universum der Größe n durch die Frage nach der Erfüllbarkeit der zu $\neg\varphi$ korrespondierenden Formel in der Aussagenlogik. Wenn sich erweist, dass in einem dieser Universen die Formel erfüllbar ist, dann ist sie nicht allgemeingültig.

Ob eine Formel *nicht* allgemeingültig ist, können wir auf diese Weise jedoch nicht sicher sagen. Wenn wir in den untersuchten endlichen Universen kein Modell finden, dann bedeutet das nicht, dass die Formel allgemeingültig ist. Es könnte ja sein, dass es ein Gegenbeispiel für ein noch größeres, eventuell unendliches Universum gibt.

Ein Beispiel für Erfüllbarkeit in endlichen Universen

In diesem Abschnitt wollen wir die Logic Workbench als *Model Finder* in der Prädikatenlogik verwenden. Wir machen uns auf die Suche nach einer Gruppe, die nicht kommutativ ist.

Wir legen die Sprache fest, in der wir arbeiten wollen, in dem wir eine Signatur definieren:

```
(def grp-sig
  {'unit [:func 0]
   'op   [:func 2]
   'inv  [:func 1]})
```

Die binäre Funktion **op** ist die Operation der Gruppe, **inv** ist die Inversenbildung und die Konstante **unit** ist das neutrale Element der Gruppe.

In dieser Sprache kann man die Axiome der Gruppentheorie formulieren:

```
;; Assoziativität
(def grp-ass
  '(forall [x y z] (= (op x (op y z)) (op (op x y) z))))
;; Neutrales Element
(def grp-unit
  '(forall [x] (= (op x unit) x)))
;; Inverses Element
(def grp-inv
  '(forall [x] (= (op x (inv x)) unit)))
```

Darüberhinaus formulieren wir die Eigenschaft der Kommutativität:

```
(def grp-comm '(forall [x y] (= (op x y) (op y x))))
```

Auf der Suche nach einer Gruppe, die nicht kommutativ ist, brauchen wir ein Modell, das die Gruppenaxiome erfüllt, jedoch nicht das Kommutativgesetz:

```
(def grp
  (list 'and grp-ass grp-unit grp-inv))
(def grp-na
  (list 'and grp (list 'not grp-comm)))
```

Nun können wir in der Logic Workbench die Funktion **sat** mit Angabe der Größe des Universums dazu verwenden, um Modelle zu generieren, die die Formel **grp-na** erfüllen, also gerade nicht-kommutative Gruppen sind.

Wir probieren einfach beginnend mit einem Universum mit 2 Elementen aus und zählen alle gefundenen Modelle:

```
(count (sat grp-na grp-sig 2 :all))
(count (sat grp-na grp-sig 3 :all))
(count (sat grp-na grp-sig 4 :all))
(count (sat grp-na grp-sig 5 :all))
; => 0
```

Das Ergebnis ist 0 für Universen mit weniger als 6 Elementen. Also probieren wir:

```
(sat grp-na grp-sig 6)
```

Wir bekommen folgendes Ergebnis

```
{unit #{{:unit}},
op #{{:e2 :e5 :e3] [:e5 :e1 :e2] [:unit :unit :unit]
      [:unit :e5 :e5] [:e5 :e5 :unit] [:e2 :e4 :e1]
      [:e4 :e5 :e1] [:e5 :unit :e5] [:e3 :e4 :e5]
      [:e1 :e2 :e5] [:e3 :e1 :unit] [:e1 :e1 :e3]
      [:unit :e4 :e4] [:e1 :unit :e1] [:unit :e2 :e2]
      [:e3 :e3 :e1] [:e2 :e2 :unit] [:e2 :unit :e2]
      [:e3 :e5 :e2] [:e4 :e3 :e2] [:e3 :e2 :e4]
      [:e2 :e3 :e5] [:e4 :unit :e4] [:e2 :e1 :e4]
      [:e1 :e5 :e4] [:e4 :e1 :e5] [:e4 :e2 :e3]
      [:unit :e1 :e1] [:unit :e3 :e3] [:e4 :e4 :unit]
      [:e1 :e3 :unit] [:e1 :e4 :e2] [:e3 :unit :e3]
      [:e5 :e3 :e4] [:e5 :e4 :e3] [:e5 :e2 :e1]},  

inv #{{:e5 :e5] [:e2 :e2] [:unit :unit] [:e1 :e3] [:e3 :e1] [:e4 :e4]},  

:univ #{{:e1 :e2 :e3 unit :e4 :e5}}
```

Die Relation `op` ist gerade die Multiplikationstabelle der Gruppe und wir können direkt ablesen, dass zum Beispiel `:e3 op :e5 = :e2`, aber `:e5 op :e3 = :e4` ist.

Es handelt sich um die symmetrische Gruppe S_3 , siehe [Group Explorer](#).

Kapitel 16

Anwendungen der Prädikatenlogik in der Softwaretechnik

16.1 Spezifikation und Analyse von Eigenschaften von Programmen

Eine wichtige Anwendung der Prädikatenlogik in der Softwaretechnik besteht in der Spezifikation von Funktionen und der Möglichkeit der systematischen Analyse der Spezifikationen und ihrer Implementierungen.

Dies kann man durchaus auf sehr unterschiedlichen Leveln der Präzision und Formalität tun. Um mit einem sehr einfachen Beispiel zu beginnen: Man kann Mehrdeutigkeiten vermeiden.

Wenn in einer Spezifikation zum Beispiel steht: „Alle Elemente des Arrays A der Länge n sind entweder 0 oder 1“, dann kann man diesen Satz ganz unterschiedlich auffassen:

$$\forall i ((0 \leq i \wedge i < n) \rightarrow (A[i] = 0 \vee A[i] = 1))$$

oder

$$\forall i ((0 \leq i \wedge i < n) \rightarrow A[i] = 0) \vee \forall i ((0 \leq i \wedge i < n) \rightarrow A[i] = 1)$$

Die Formulierung mit Hilfe der Prädikatenlogik stiftet Klarheit, und wenn man die Syntax etwas anreichert, werden die Bedingungen noch besser lesbar:

$$\forall i \in [0, n - 1] : A[i] = 0 \vee A[i] = 1$$

$$\forall i \in [0, n - 1] : A[i] = 0 \vee \forall i \in [0, n - 1] : A[i] = 1$$

Ein bisschen Formalismus hilft Missverständnisse zu vermeiden.

Man kann die Anwendung der Prädikatenlogik natürlich weit konsequenter anwenden. Eine naheliegende Möglichkeit besteht darin für eine Funktion (oder ein Programm) zwei Prädikate zu formulieren:

- Eine *Vorbedingung* φ spezifiziert, was vor Ausführen der Funktion zutrifft.
- Die *Nachbedingung* ψ spezifiziert, was nach dem Ende der Funktion zutrifft.

Mit dieser Idee kann man über die Korrektheit von Programmen in imperativen Sprachen mit Mitteln der Prädikatenlogik argumentieren. Dieses Vorgehen wird als *Hoare-Kalkül*¹ bezeichnet.

Das Hoare-Tripel beschreibt wie ein Programmsegment P den Zustand der Berechnung so ändert, dass vorher die Vorbedingung φ gilt und hinterher die Nachbedingung ψ :

$$\langle\varphi\rangle P \langle\psi\rangle$$

Im Hoare-Kalkül geht es darum, zu *beweisen*, dass das Programmsegment P tatsächlich die Eigenschaft hat, dass $\varphi \rightarrow \psi$ gilt. Außerdem ist zu untersuchen, ob die Berechnung *terminiert*. Ist beides der Fall, dann spricht man von *totaler Korrektheit*.

Dieser Abschnitt soll nur auf diese Technik der Programmverifikation hinweisen, mehr findet man in [HR04, Kapitel 4].

*

Es gibt noch einen anderen Ansatz für die Verwendung der Prädikatenlogik in der Softwaretechnik, der hier erwähnt werden soll. Man kann mit dem Beweissystem *Coq* sogenannte zertifizierte Programme entwickeln. Im ersten Kapitel von [BC04] wird als Beispiel gezeigt, wie man zwei Prädikate formulieren kann, mit denen man beschreiben kann, was ein Sortierprogramm erreichen soll:

Gegeben eine Liste L von ganzen Zahlen. Sei $\sigma(L)$ die Formel, die beschreibt, dass L sortiert ist und sei \equiv die Relation, die für zwei Listen L und L' ausdrückt, dass sie die gleichen Elemente enthalten. Die Spezifikation eines Sortieralgorithmus ist dann der Typ einer Funktion, die jede Liste L auf eine Liste L' abbildet, die folgende Formel erfüllt:

$$\sigma(L') \wedge L' \equiv L$$

¹ nach C.A.R. „Tony“ Hoare britscher Informatiker.

Die Aufgabe ein zertifiziertes Sortierprogramm zu entwickeln ist dann identisch mit der Aufgabe einen Term des eben beschriebenen Typs zu finden. Und eben dieses kann man interaktiv in *Coq* tun und anschließend daraus Programmcode in den Sprachen OCaml, Haskell oder Scheme generieren.

Dieser kurze Hinweis soll als „Eyecatcher“ genügen. Wir werden uns im folgenden Abschnitt mit einer „leichtgewichtigen“ formalen Methode beschäftigen.

16.2 Analyse von Software mit Alloy

In der Prädikatenlogik kann man sich (wie in der Aussagenlogik auch) zwei Fragen stellen, zwei Konzepte betrachten:

1. Gegeben sei ein Modell \mathcal{M} , also gewissermaßen eine vorgegebene *Welt*, und eine Formel φ der Prädikatenlogik, gilt dann die Formel φ in \mathcal{M} , d.h. gilt $\mathcal{M} \models \varphi$?
2. Gegeben eine Menge Γ von Formeln und eine Formel φ , folgt dann φ aus Γ , d.h. gilt $\Gamma \vdash \varphi$ bzw. $\Gamma \vDash \varphi$?

Überträgt man diese beiden Fragestellung auf das Spezifizieren und Entwickeln von Software, dann führen die Konzepte zu:

1. *Model Checking*. Gegeben sei eine Implementierung ($\hat{=}$ Modell \mathcal{M}), erfüllt diese dann eine verlangte Anforderung ausgedrückt in der Formel φ ? Bei diesem Ansatz brauchen wir also ein Modell, d.h. eine Menge Details im Modell, die vielleicht gar nicht aus den Anforderungen an die Software stammen.
2. *Gültigkeit*. Der Ausgangspunkt ist nicht ein Modell, sondern eine *Spezifikation* des zu untersuchenden Systems, ausgedrückt in der Formelmenge Γ . Dann kann man fragen, ob in *allen* Implementierungen, die diese Spezifikation erfüllen, auch die Anforderung φ garantiert ist. Bei diesem Ansatz benötigt man also nicht Details eines Modells, hat aber andererseits den Nachteil, dass diese Frage – wie wir gesehen haben – nicht entscheidbar ist.

16.2.1 Die Idee hinter Alloy

Die Idee hinter der leichtgewichtigen formalen Methode Alloy von Daniel Jackson² besteht darin, die beiden Ansätze zu kombinieren:

² Daniel Jackson, Professor für Informatik am Massachusetts Institute of Technology (MIT).

1. Ausgangspunkt ist die Spezifikation Γ . Begrenzt man nun die Größe des Universums, so ist es möglich, Modelle \mathcal{M} zu finden, die dieser Spezifikation genügen. Man macht also gewissermaßen *Model Finding*.
2. Nun verwendet man diese Modelle um zu checken, ob sie die Formel φ erfüllen.

Diese Kombination kann man verwenden, um zu zeigen, dass eine Spezifikation Γ die Anforderung φ *nicht* erfüllt: Man sucht nach Modellen für die Formelmenge $\Gamma \cup \{\neg\varphi\}$. Wenn man ein (kleines) Modell dafür findet, dann hat man ein Gegenbeispiel. Und kann dann untersuchen, woran es liegt, dass $\Gamma \not\models \varphi$ gilt.

Wenn man kein Gegenbeispiel findet, dann kann man natürlich nicht sicher sein, dass $\Gamma \models \varphi$ gilt, weil für das Generieren der Modell eine maximale Größe des Universums vorgegeben werden muss. Es könnte ja sein, dass in einem größeren Universum doch ein Modell existiert.

Daniel Jackson geht von der sogenannten *Small Scope Hypothesis*³ aus, die besagt, dass viele (Denk-)Fehler schon im Kleinen auftauchen und mit Alloy sichtbar gemacht werden können. Unsere Erfahrung als Softwareentwickler bestätigt dies: es ist oft so, dass wenn ein Fehler in einem Stück Software erst mal gefunden ist, sich einfache Beispiele bilden lassen ihn zu demonstrieren.⁴

16.2.2 Beispiele für Analysen mit Alloy

Auf der [Webseite zu Alloy](#) findet man eine Vielzahl von Analysen, die mit Alloy gemacht wurden. Darunter:

- Kritische Systeme in der *Medizintechnik*. Eine Gruppe an der University of Washington hat mit Alloy ein System für die Neutronenstrahlentherapie systematisch untersucht und dabei mehrere kritische Fehler gefunden, die vor dem Einsatz des Systems korrigiert werden konnten, siehe [University of Washington PLSE Neutrons](#).

³ “Most flaws in models can be illustrated by small instances, since they arise from some shape being handled incorrectly, and whether the shape belongs to a large or small instance makes no difference. So if the analysis considers all small instances, most flaws will be revealed. This observation, which I call the small scope hypothesis, is the fundamental premise that underlies Alloy’s analysis.” [Jac12, S.15]

⁴ Das ist gewissermaßen das $\mathcal{P} \neq \mathcal{NP}$ der Softwareentwicklung: Fehler finden ist extrem schwer, hinterher zu sehen, worin der Fehler besteht, oft überraschend einfach.

- *Netzwerk-Protokolle.* Pamela Zave⁵ hat die Spezifikation für Chord, eine Peer-to-Peer verteilte Hashtabelle, mit Alloy und Spin analysiert und dabei verschiedene Fehler entdeckt [Zav12].
- *Programmverifikation.* Die Idee hierbei besteht darin, aus Java-Code, der mit JML (Java Modeling Language) annotiert ist, also mit Klasseninvarianten, Vorbedingungen, Nachbedingungen und Schleifen-Invarianten formal spezifiziert ist, automatisch Spezifikationen in Alloy zu erzeugen, die man dann für die Verifikation von Programmeigenschaften verwenden kann. Greg Dennis aus der Arbeitsgruppe von Daniel Jackson hat das Tool **Forge** entwickelt, mit dem solche Analysen möglich sind. Forge wurde verwendet, um Implementationen von Listen in Java zu untersuchen. Dabei wurden sowohl Fehler in den JML-Spezifikationen als auch in den Implementierungen gefunden⁶. Außerdem wurde Forge eingesetzt, um eine Wahlsoftware in den Niederlanden zu analysieren. Auch bei dieser Analyse konnten Fehler in Spezifikationen und in der Implementierung aufgedeckt werden⁷.
- *IT-Sicherheit.* Eine Gruppe an der University of California Berkeley und an der Stanford University hat verschiedene Sicherheitsmechanismen für das Web analysiert: WebAuth, HTML5 forms und das Cross-Origin Resource Sharing Protokoll u.a. Dabei wurden bekannte Schwachstellen bestätigt und drei weitere entdeckt⁸.

16.2.3 Die Sprache Alloy und das Werkzeug Alloy Analyzer

Alloy ist nicht auf die Untersuchung vordefinierter Artefakte der Softwareentwicklung festgelegt. Man kann Ideen, Spezifikationen, Entwürfe, Implementierung usw. mit Alloy untersuchen. In Abbildung 16.1 zeigt die Ziffer ① dass zunächst aus dem zu untersuchenden Gegenstand eine Spezifikation in Alloy erstellt werden muss. Dies betrifft sowohl die Struktur der Situation wie auch die Dynamik, also Veränderungen an der Situation. Was die Dynamik angeht, ist zu beachten, dass man nicht nur spezifizieren muss, was sich ändert, sondern auch was im veränderten Zustand gleich geblieben ist, die sogenannten *frame conditions*.

⁵ Pamela Zave, amerikanische Informatikerin, lange Zeit in der Forschung und Entwicklung bei AT&T tätig, seit 2017 Forscherin an der Princeton University.

⁶ *Modular Verification of Code with SAT* von Greg Dennis, Felix Chang und Daniel Jackson.

⁷ *Bounded Verification of Voting Software* von Greg Dennis, Kuat Yessenov und Daniel Jackson.

⁸ *Towards a Formal Foundation of WebSecurity* von Devdatta Akhawe, Adam Arth, Peifung E. Lam, John Mitchell und Dawn Song.

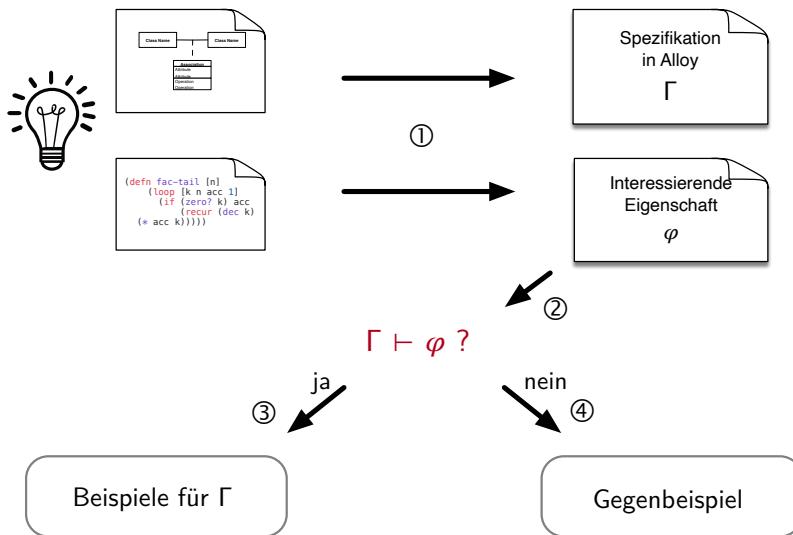


Abbildung 16.1: Verwendung von Alloy

Den Entwurf der Spezifikation macht man interaktiv im *Alloy Analyzer* ②, indem man schrittweise die Spezifikation erstellt und immer wieder den Alloy Analyzer Modelle für die Spezifikation erzeugen lässt. Dies ist eine hoch interessante, sehr agile Erfahrung. Typischerweise wird man feststellen, dass bei der Spezifikation Bedingungen zunächst vergessen werden, etwa weil man sie für selbstverständlich hält. Oder man erhält Beispiele, an denen man erkennt, dass es Spezialfälle gibt, die man nicht berücksichtigt hat.

Mit dem Kommando `run` kann man zur Spezifikation Γ Beispiele (siehe ③) erzeugen lassen zu einer vorgegebenen Maximalzahl von Elementen der definierten Sigs (Default ist 3). Die Formel φ formuliert man in Alloy als `assert` und das Kommando `check` sucht nach einem Gegenbeispiel. Wird dies gefunden wird es (④) angezeigt und kann analysiert werden.

16.2.4 Die Sprache Alloy

Wir haben die Korrespondenz von Prädikaten und Relationen in Abschnitt 11.2 diskutiert. Alloy ist eine Sprache, in der man Relationen deklarieren kann, also eine Sprache der relationalen Logik.

Für die Elemente der Sprache mag hier ein Hinweis auf den Text *Kurze Einführung in Alloy* genügen, den ich gemeinsam mit Nils Asmussen verfasst habe.

16.2.5 Der Alloy Analyzer — unter der Haube

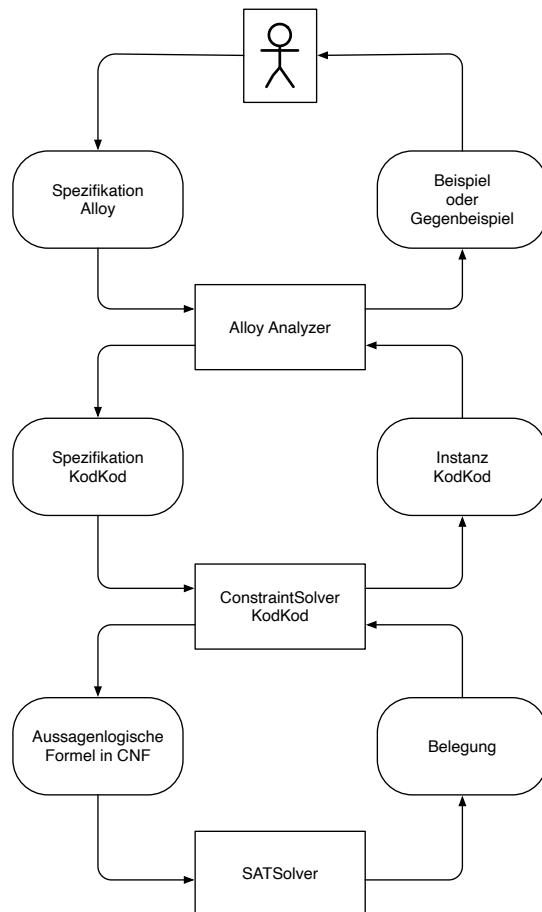


Abbildung 16.2: Arbeitsweise des Alloy Analyzers

Die grundlegende Idee für die Arbeitsweise des Alloy Analyzers ergibt sich aus der Tatsache, dass man für endliche Universen eine Formel der Prädikatenlogik in eine Formel der Aussagenlogik transformieren kann, wie in Abschnitt 15.3 diskutiert. Tatsächlich wird diese Transformation nicht direkt vom Alloy Analyzer durchgeführt, sondern, wie in Abbildung 16.2 dargestellt, vom *Kodkod Constraint Solver*⁹. Der Alloy Analyzer übersetzt die Spezifikation in Alloy in eine Spezifikation in Kodkod. Dieses Werkzeug leistet nun die eigentliche Transformation der Spezifikation in die Aussagenlogik und verwendet dann SAT-Solver, als Default SAT4J, um die Erfüllbarkeit der Formel zu prüfen. Wird eine

⁹ Der *Kodkod Constraint Solver* wurde von Emina Torlak im Rahmen ihrer Promotion bei Daniel Jackson entwickelt.

Belegung gefunden, wird daraus eine Instanz von Relationen in Kodkod gebildet und diese vom Alloy Analyzer in ein Beispiel oder Gegenbeispiel (je nach Fragestellung) in Alloy zurückinterpretiert. Besonders wichtig ist das sogenannte *Symmetry Breaking*: Permutationen der Bezeichnung von Atome in Formeln führen zu Lösungen, die im Prinzip bis auf die Benennung der Atome identisch sind. Kodkod setzt deshalb automatisch spezielle zusätzliche Bedingungen ein, die solche symmetrischen Modelle ausschließen.

16.2.6 Ein (kleines) Beispiel

Michael Jackson demonstriert die Möglichkeiten von Alloy am Beispiel einer Idee zu einem E-Mail-Programm [Jac06]. Dieses Beispiel wollen wir in diesem Abschnitt nachvollziehen.

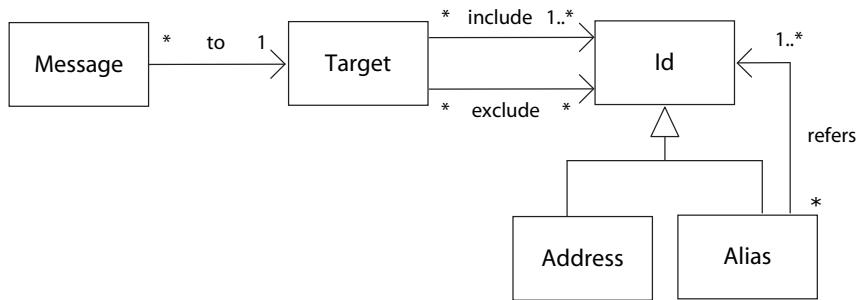


Abbildung 16.3: Klassendiagramm E-Mail-Programm

Das Klassendiagramm in Abbildung 16.3 visualisiert das Konzept der Adressierung von E-Mails: Jede Nachricht (*Message*) hat genau ein Ziel (*Target*), nämlich die Menge der Adressaten. Diese können in Gruppen eingeteilt werden. Die Konstruktion der 3 Klassen *Id*, *Alias* und *Address* zeigt, dass *Ids* entweder eine Adresse sein können oder ein Alias, der für eine Menge weiterer *Ids* steht, die selbst wieder Adressen oder Aliase sein können.

Die Eigenschaft, die wir mit Alloy analysieren wollen, besteht darin, dass im geplanten E-Mail-Programm es sowohl die Möglichkeit geben soll, *Ids* dem Ziel hinzuzufügen (*include*), als auch auszuschließen (*exclude*). Dieses Feature könnte für Anwender sinnvoll sein, wenn jemand zum Beispiel eine Mitteilung an alle Kolleg:innen verschicken möchte, jedoch bestimmte Personen ausschließen möchte.

Nun stellt sich die Frage, wie werden die wirklichen Adressaten ermittelt: Spielt es eine Rolle, ob man die Differenzmenge zwischen den eingeschlossenen und den ausgeschlossenen Adressen *vor* oder *nach* der Auflösung

der Alias-Referenzen bilden muss. Und genau diese Frage wollen wir nun mit Alloy untersuchen.

Zunächst spezifizieren wir die im Klassendiagramm dargestellte Struktur in Alloy:

```

1 module lfm/email
2
3 sig Message {
4   to: Target
5 }
6 sig Target {
7   include: some Id,
8   exclude: set Id
9 }
10 abstract sig Id {}
11 sig Address extends Id {}
12 sig Alias extends Id {
13   refers: set Id
14 }
15
16 run {
17   #Alias = 3
18 }
```

In Zeile 3 der Spezifikation wird eine *Signature* definiert. man kann damit die Vorstellung verbinden, dass damit ein Typ definiert wird. Die *Signatures* **Message** und **Target** kommen in der Relation **to** vor. Dabei beinhaltet die Syntax, dass es zu jedem Element in **Message** genau ein Element von **Target** in der Relation **to** steht.

In Zeile 7 wird festgelegt, dass in **include** zu einem **Target** mehrere Ids, aber mindestens eine, gehören. In Zeile 8 wird eine Menge von Ids zugeordnet, die auch leer sein könnte.

Ein abstrakte *Signature* wie **Id** in Zeile 10 bedeutet eine Partitionierungen des Typs in die beiden Typen **Address** sowie **Alias**.

Wie man sieht, ist es recht geradlinig, wie man das Klassendiagramm in die Spezifikation transformiert. Das liegt auch daran, dass Daniel Jackson beim Entwurf der Sprache ihr ein objekt-orientiertes Flair gegeben hat.

in Zeile 16 formulieren wir ein Kommando an den Alloy Analyzer: er soll Modelle zur Spezifikation erstellen und zwar solche, bei denen der Typ **Alias** genau drei Elemente hat. Auf diese Weise können wir prüfen, ob unsere Spezifikation auch dem entspricht, wie wir uns die Struktur der Ids vorstellen.

Und siehe da: Es gibt Beispiele, die gar nicht so ausschauen, wie wir dachten. Es kommen Beispiele vor, bei denen ein Alias gar nicht auf eine wirkliche Adresse verweist und es kommen Zyklen im Graph der Adressen vor. Das ist ein Beispiel dafür, dass das Klassendiagramm nicht alle Integritätsbedingungen ausdrückt, die eigentlich gemeint sind. Wir erweitern die Spezifikation und sorgen dafür, dass keine Zyklen auftreten können. Für unsere Fragestellung ist es unerheblich, dass es vorkommen kann, dass ein Alias gar keine Referenz hat, deshalb brauchen wir die Spezifikation bezüglich dieser Frage nicht zu erweitern. Das ist ein Beispiel dafür, dass man Alloy als „leichtgewichtig“ bezeichnen kann. Man muss nicht alle denkbaren Möglichkeiten spezifizieren, sondern kann sich auf das für die konkrete Fragestellung relevanten Eigenschaften konzentrieren.

```
20 fact AliasingIsAcyclic {
21   no a: Alias | a in a.^refers
22 }
```

In Zeile 20 wird spezifiziert, dass es keine Zyklen geben darf. `a.^refers` bezeichnet in Alloy den transitiven Abschluss der binären Relation `refers`. Ein *Fact* ist eine Bedingung, die immer gelten muss, hier also, dass kein Alias in seinem transitiven Abschluss enthalten sein darf, die Struktur also keine Zyklen haben darf.

Nun können wir an die eigentliche Fragestellung gehen. Dazu definieren wir in Alloy zwei Funktionen:

```
24 fun diffThenRefers(t: Target): set Id {
25   t.(include - exclude).*refers - Alias
26 }
27 fun refersThenDiff(t: Target): set Id {
28   (t.include.*refers - t.exclude.*refers) - Alias
29 }
```

Die erste Funktion hat ein `Target` als Argument und ergibt die Menge der Ids, die man erhält, wenn man erst die Differenz bildet und dann dereferenziert. Dabei wird der Operator `.`, der sogenannte *Dot Join* verwendet, sowie der Operator `*`, der für den reflexiven transitiven Abschluss steht. Bei der zweiten Funktion wird erst dereferenziert und dann die Differenz gebildet.

Nun können wir den Alloy Analyzer prüfen lassen, ob die beiden Vorgehensweisen übereinstimmen. Dazu verwenden wir das Kommando `check`, mit dem wir den Analyzer auf die Suche nach einem Gegenbeispiel schicken:

```
35 assert OrderIrrelevant{
36   all t: Target | diffThenRefers[t] = refersThenDiff[t]
```

```
37 }
38 check OrderIrrelevant for 3 but 1 Target
```

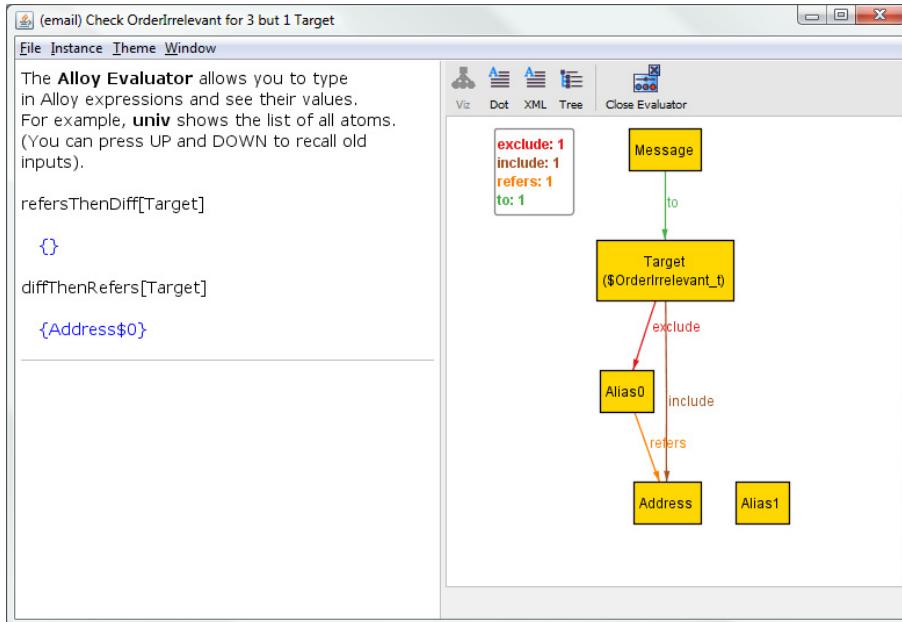


Abbildung 16.4: Alloy Analyzer Überprüfung des E-Mail-Programms

Ein mögliches Ergebnis der Analyse ist in Abbildung 16.4 dargestellt. Dabei wird auch der *Alloy Evaluator* verwendet: Im linken Teil des Fensters kann man Ausdrücke eingeben, die im aktuell im rechten Teil angezeigten Modell ausgewertet werden. Und nicht nur am Modell kann man so ablesen, dass die beiden Konzepte nicht übereinstimmen, man sieht explizit im linken Teil, dass nach der ersten Vorgehensweise jemand die E-Mail bekommen hätte, der eigentlich ausgeschlossen sein sollte, indirekt über ein Alias.

Doch wir erhalten nicht nur das Ergebnis, dass sicherlich die Vorgehensweise zuerst zu dereferenzieren und dann die Differenz zu bilden, das gewünschte Verhalten ist, sondern auch, dass es dabei vorkommen kann, dass die Adressatenmenge leer ist, was man also bei der Entwicklung des E-Mail-Programms berücksichtigen muss.

Teil III

Lineare Temporale Logik

Kapitel 17

Dynamische Modelle

Bisher sind die beiden Logiken von ihrem Ansatz her eher statisch. Sie erlauben es, eine bestimmte Situation deklarativ zu einem bestimmten Zeitpunkt zu beschreiben. Softwaressysteme haben jedoch eine zeitliche Dimension: ihr Zustand zu einem Zeitpunkt ändert sich und führt zu einem anderen Zustand in einem späteren Zeitpunkt.

Natürlich kann man diese zeitliche Veränderung des Zustands auch in der Prädikatenlogik formulieren. In Alloy tut man dies zum Beispiel dadurch, dass man eine Signatur `State` oder `Time` einführt und sie als Komponente von Relationen zu all den Signaturen verwendet, deren Zustand sich ändern kann. Dieses Vorgehen wird als eine der Techniken, Dynamik in Alloy zu spezifizieren, beschrieben im Technischen Bericht von Nils Asmussen: [Ansätze zur Modellierung von Dynamik mit Alloy](#).

Man kann aber auch Modelle verwenden, in denen die Dimension der Zeit direkt und explizit vorgesehen ist: Transitionssysteme.

17.1 Konzept der Transitionssysteme

Die Grundidee:

1. Ein System befindet sich zu einem bestimmten Zeitpunkt in einem bestimmten *Zustand*.
2. Es treten Ereignisse auf, die *Transitionen* (Zustandsübergänge) auslösen, die zu einem anderen, späteren Zustand führen.

Man kann unterscheiden zwischen

1. *diskreten* und *kontinuierlichen* Transitionssystemen. In diskreten Systemen geht man von einer diskreten linearen Zeit aus: t_1, t_2, t_3, \dots . Solche Systeme werden wir verwenden.

2. *deterministischen* und *nicht deterministischen* Transitionssystemen.

In deterministischen Systemen führt ein bestimmtes Ereignis dazu, dass das System in genau einen bestimmten Folgezustand wechselt.

Am Beispiel einer Ampelschaltung in Abbildung 17.1 kann jeder Zustand durch die Belegung der aussagenlogischen Atome *Rot*, *Gelb*, *Grün* beschrieben werden:

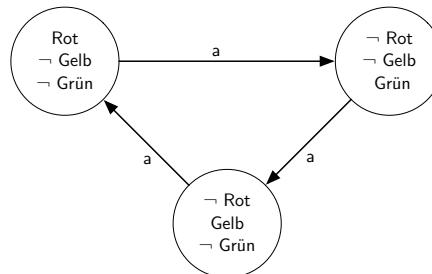


Abbildung 17.1: Transitionssystem Ampel

Geht man von einer fixen Menge von aussagenlogischen Atomen aus, wird man die Zustände des Transitionssystems oft nur mit den Atomen beschriften, die in diesem Zustand mit T belegt sind.

17.2 Beispiel eines Programms mit zwei Threads

Ein Entwickler möchte zwei Threads T1 und T2 synchronisieren, in dem er zwei Variablen `t1gesperrt` und `t2gesperrt` verwendet. Das Schlüsselwort `volatile` garantiert, dass Änderungen an den Variablen von allen Threads „gesehen“ werden. Dadurch möchte er sicherstellen, dass sich die beiden Threads in ihren Aktionen nicht in die Quere kommen.

Hier der Pseudocode des Konzepts:

```

volatile boolean t1gesperrt = false;
volatile boolean t2gesperrt = false;

// Thread T1:
run() {
T1.1  t1gesperrt = true;
T1.2  istT2Frei();
      // hier sind die kritischen Aktionen von T1
  
```

```

T1.3    t1gesperrt = false;
T1.e  }

// Thread T2:
run() {
T2.1    t2gesperrt = true;
T2.2    istT1Frei();
        // hier sind die kritischen Aktionen von T2
T2.3    t2gesperrt = false;
T2.e  }

```

Die Funktion `istT1Frei()` ist folgendermaßen implementiert, `istT2Frei()` analog.

```

void istT1Frei() {
    forever {
        if (t1gesperrt == false) return;
        sleep(10);
    }
}

```

Um dieses Konzept in ein Transitionssystem umzusetzen, vereinbaren wir die Notation in Abbildung 17.2.

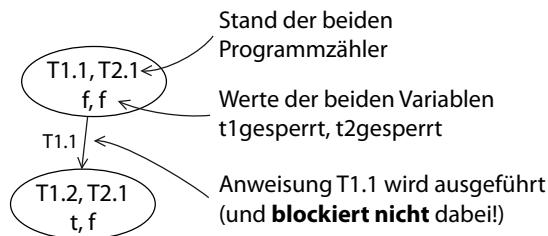


Abbildung 17.2: Notation für das Beispiel eines Programms mit zwei Threads

Nun kann man das Transitionssystem für dieses Programm wie in Abbildung 17.3 darstellen. Man sieht in dem Graphen alle möglichen Berechnungspfade. Und ebenso ist mühelos sichtbar, dass wir einen Zustand haben, der ein „schwarzes Loch“ darstellt und in dem beide Threads keinen Fortschritt mehr machen können; sie sind verklemmt.

17.3 Temporale Logik

Um Transitionssysteme analysieren zu können, kommt *temporale Logik* ins Spiel. Man will Aussagen über die Zustände in den verschiedenen Berechnungspfaden von Programmen machen können. Ein solcher Pfad ist dann eine Folge von Zuständen. Die diskrete Zeit ist einfach die lineare

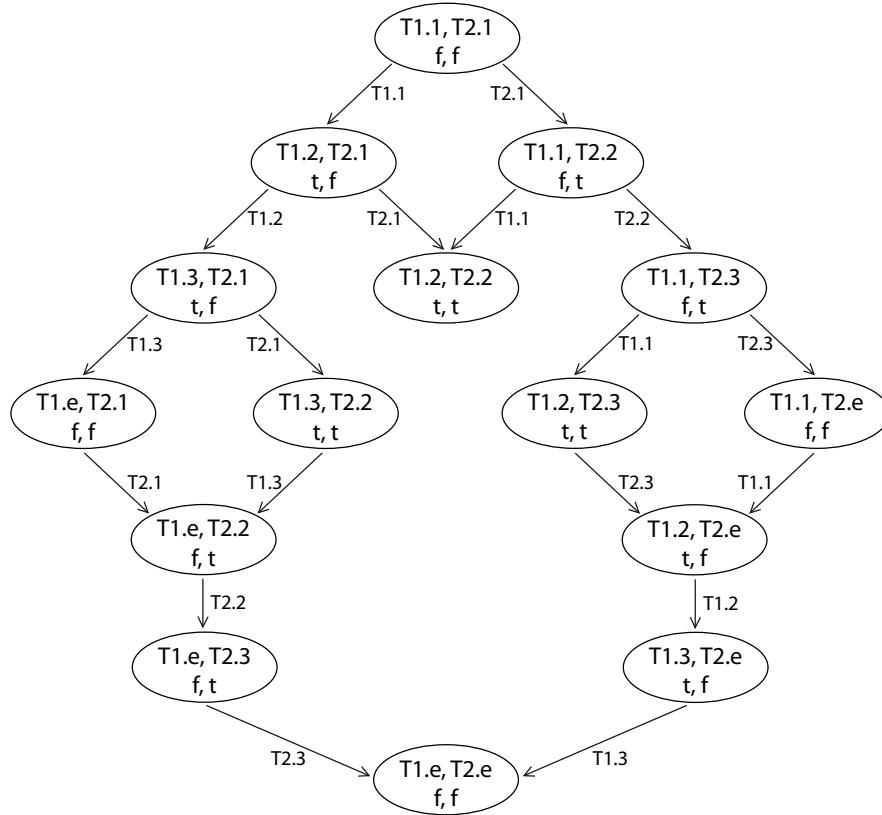


Abbildung 17.3: Transitionssystem für das Beispiel

Ordnung der Zustände entlang des Pfads. Man möchte dann Aussagen machen können, wie „irgendwann gilt ...“ oder „es kann niemals sei, dass ...“.

Es gibt einen ganzen „Zoo“ temporaler Logiken:

- *Lineare temporale Logik (LTL)* erlaubt Aussagen, die sich auf *alle* möglichen Pfade in einem Transitionssystem beziehen.
- *Computational Tree Logic (CTL)* berücksichtigt die Möglichkeit von Verzweigungen, Dazu hat CTL temporale Operatoren, die mit Pfadquantoren kombiniert sind, d.h. man bildet Formeln wie „Für alle Pfade gilt im nächsten Zustand ...“ oder „Es gibt einen Pfad, auf dem immer ... gilt“.
- *CTL**: LTL und CTL haben unterschiedliche Ausdrucksstärke (siehe etwa [CGP99, S. 30]), CTL* ist eine temporale Logik, die beider Ausdrucksstärke kombiniert.

Wir werden in diesem Teil der Veranstaltung LTL betrachten.

Kapitel 18

Die formale Sprache der linearen temporalen Logik (LTL)

In der Sprache der linearen temporalen Logik (LTL) erweitert man die formale Sprache der Aussagenlogik durch weitere Junktoren, mit denen temporale Eigenschaften formuliert werden können.

Definition 18.1 (Alphabet der LTL). Das *Alphabet* der Sprache der linearen temporalen Logik (LTL) besteht aus

- (i) einer Menge \mathcal{P} von Aussagensymbolen,
- (ii) den (aussagenlogischen) Junktoren: $\neg, \wedge, \vee, \rightarrow$
- (iii) den (temporalen) Junktoren: \circ, \mathcal{U}
- (iv) der Konstanten: \perp
- (v) den zusätzlichen Symbolen: $(,)$

Bemerkungen

- Wie im Falle der Aussagenlogik, definieren wir *eine* Sprache der linearen temporalen Logik durch die Vorgabe der Menge \mathcal{P} und der eben definierten Junktoren.
- Die formale Sprache der LTL ist eine Erweiterung der Aussagenlogik, in der zwei neue Junktoren vorkommen:
 - \circ steht für „zum nächsten Zeitpunkt“ (*next*)
 - \mathcal{U} steht für „bis“ (*until*)

Definition 18.2 (Formeln der LTL). Die *Formeln* der linearen temporalen Logik sind Zeichenketten, die nach folgenden Regeln gebildet werden:

- (i) Jedes Symbol $P \in \mathcal{P}$ ist eine Formel und auch \perp ist eine Formel.
- (ii) Ist φ eine Formel, dann auch $\neg\varphi$.
- (iii) Sind φ und ψ Formeln, dann auch $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ und $(\varphi \rightarrow \psi)$
- (iv) Ist φ eine Formel, dann auch $\circ \varphi$.
- (v) Sind φ und ψ Formeln, dann auch $(\varphi \mathcal{U} \psi)$

Als *Grammatik* in Backus-Naur-Darstellung können wir diese induktive Definition der Formeln der LTL so ausdrücken:

$$\varphi ::= P \mid \perp \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \circ \varphi \mid (\varphi \mathcal{U} \varphi)$$

mit Variablen $P \in \mathcal{P}$ und (bereits gebildeten) Formeln φ .

Die Präzedenz der Junktoren wird folgendermaßen definiert: Die unären Junktoren \neg und \circ binden stärker als die binären, sie selbst binden gleich stark. Binäre Junktoren binden in folgender Reihenfolge, die stärkste Bindung zuerst: $\mathcal{U}, \wedge, \vee, \rightarrow$. Außerdem sind \wedge und \vee linksassoziativ, \mathcal{U} und \rightarrow sind rechtsassoziativ.

Bemerkung. In Formeln der LTL werden wir oft vier weitere Junktoren verwenden, die folgendermaßen definiert werden:

$$\begin{aligned}\diamond \varphi &\stackrel{\text{def}}{=} \neg \perp \mathcal{U} \varphi \\ \Box \varphi &\stackrel{\text{def}}{=} \neg \diamond \neg \varphi \\ \varphi \mathcal{W} \psi &\stackrel{\text{def}}{=} (\varphi \mathcal{U} \psi) \vee \Box \varphi \\ \varphi \mathcal{R} \psi &\stackrel{\text{def}}{=} \neg(\neg \varphi \mathcal{U} \neg \psi)\end{aligned}$$

Wir lesen sie so:

- \diamond steht für „irgendwann“ (*eventually*),
- \Box steht für „immer“ (*always*),
- \mathcal{W} steht für „sofern nicht“ (*unless, weak until*) und
- \mathcal{R} steht für „löst ab“ (*release*).

\diamond und \Box haben dieselbe Bindungspräzedenz wie \circ und \mathcal{W} und \mathcal{R} die von \mathcal{U} .

Kapitel 19

Die Semantik der linearen temporalen Logik (LTL)

19.1 Kripke-Struktur

Modelle in der temporalen Logik enthalten ein implizites Konzept einer diskreten Zeit: Man denkt sich die „Welt“ des Modells als bestehend aus Zuständen, in denen gewissen Aussagen wahr sind und einer Übergangsrelation der Zustände. Jeder Zustandsübergang entspricht dann gerade einem Zeitschritt. Präziser definiert man die Kripke-Struktur¹:

Definition 19.1 (Kripke-Struktur). Eine *Kripke-Struktur* \mathcal{K} ist ein Tupel (S, s_0, \rightarrow, L) bestehend aus

- einer Menge von Zuständen S ,
- einem ausgezeichneten Startzustand $s_0 \in S$,
- einer Übergangsrelation $\rightarrow \subseteq S \times S$, die jedem Zustand s einen Folgezustand s' zuordnet (d.h. $\forall s \exists s' \text{ mit } s \rightarrow s'$) und
- einer Beschriftungsfunktion $L : S \rightarrow \mathbb{P}(\mathcal{P})$ von S in die Potenzmenge von \mathcal{P} , die jedem Zustand eine Menge von (in diesem Zustand wahren) Aussagenatomen zuordnet.

Bemerkungen

1. Man könnte in die Definition der Kripke-Struktur auch die Wahl von \mathcal{P} explizit aufnehmen. In den Beispielen, die wir betrachten, ergibt sich die Menge der Atome aus der Beschriftungsfunktion.
2. Die Beschriftungsfunktion L ordnet jedem Zustand die in diesem Zustand wahren Aussagen aus \mathcal{P} zu. Dies kann man auch so sehen: L ordnet jedem Zustand s eine Belegung $v_s : \mathcal{P} \rightarrow \mathbb{B}$ zu.

¹ Saul A. Kripke, amerikanischer Logiker.

3. Eine Kripke-Struktur kann man als gerichteten Graphen sehen, in dem die Zustände S die Knoten sind und die Übergangsrelation gerade die gerichteten Kanten. Zudem wird jeder Zustand mit den in ihm gültigen Aussagen gemäß der Beschriftungsfunktion L markiert. Der Startzustand wird durch einen eingehenden Pfeil ohne Startknoten markiert.
4. Manchmal zeichnet man in einer Kripke-Struktur keinen Startzustand aus, man bezeichnet sie dann als Übergangssystem (siehe [HR04, Abschnitt 3.2]).
5. Manche Autoren lassen in Kripke-Strukturen auch mehrere Startzustände zu.
6. Wenn in einem konkreten System die Übergangsrelation nicht die Eigenschaft hat, dass es zu jedem Zustand einen Folgezustand gibt, kann man den Graph um einen Zustand erweitern, der einen Übergang auf sich selbst hat und hat damit die Definition einer Kripke-Struktur erfüllt.

Beispiele In Abb. 19.1 und 19.2 werden die Graphen zu zwei Beispielen dargestellt.

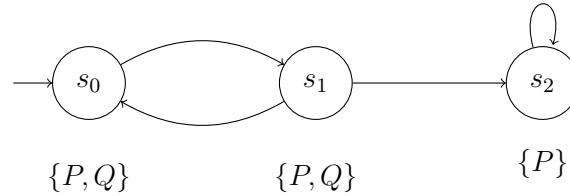


Abbildung 19.1: Beispiel einer Kripke-Struktur

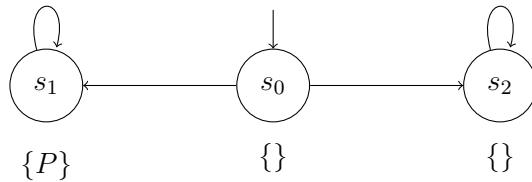


Abbildung 19.2: Beispiel für die Semantik der Negation in Kripke-Strukturen

Definition 19.2 (Pfad und Berechnung). Sei $\mathcal{K} = (S, s_0, \rightarrow, L)$ eine Kripke-Struktur.

Ein *Pfad* π ist eine unendliche Folge s_1, s_2, s_3, \dots von Zuständen $s_i \in S$ mit $s_i \rightarrow s_{i+1}$ für alle $i \geq 1$. Man schreibt einen Pfad gerne so:

$$\pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$$

Hat man einen Pfad $\pi = s_1 \rightarrow s_2 \rightarrow s_3 \dots$ gegeben, dann bezeichnet man mit π^i den Pfad, der im i -ten Zustand von π beginnt, also z.B. $\pi^2 = s_2 \rightarrow s_3 \rightarrow s_4 \dots$

Eine *Berechnung* ist ein Pfad, der mit dem Startzustand $s_0 \in S$ beginnt.

Nun haben wir alle Notation, um Semantiken der LTL definieren zu können:

Definition 19.3 (Semantik der LTL für einen Pfad). Sei \mathcal{K} eine Kripke-Struktur und $\pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$ ein Pfad. Für eine Formel φ der linearen temporalen Logik definiert man

$$\pi \models \varphi,$$

falls $\llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = T$. Dabei wird $\llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}}$ induktiv definiert über den strukturellen Aufbau von φ

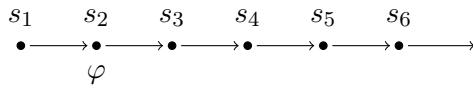
- (i) $\llbracket \perp \rrbracket_{\pi}^{\mathcal{K}} := F$
- (ii) $\llbracket P \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } P \in L(s_1) \\ F & \text{sonst} \end{cases}$
- (iii) $\llbracket \neg \varphi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = F \\ F & \text{sonst} \end{cases}$
- (iv) $\llbracket \varphi \wedge \psi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = T \text{ und } \llbracket \psi \rrbracket_{\pi}^{\mathcal{K}} = T \\ F & \text{sonst} \end{cases}$
- (v) $\llbracket \varphi \vee \psi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = T \text{ oder } \llbracket \psi \rrbracket_{\pi}^{\mathcal{K}} = T \\ F & \text{sonst} \end{cases}$
- (vi) $\llbracket \varphi \rightarrow \psi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = F \text{ oder } \llbracket \psi \rrbracket_{\pi}^{\mathcal{K}} = T \\ F & \text{sonst} \end{cases}$

$$(vii) \quad \llbracket \circ \varphi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_{\pi^2}^{\mathcal{K}} = \text{T} \\ \text{F sonst} \end{cases}$$

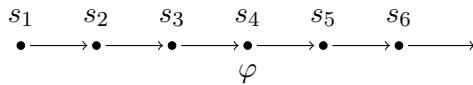
$$(viii) \quad \llbracket \varphi \mathcal{U} \psi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} \text{T falls } \exists i \geq 1 \text{ mit } \llbracket \psi \rrbracket_{\pi^i}^{\mathcal{K}} = \text{T} \\ \text{und } \forall j = 1, \dots, i-1 \llbracket \varphi \rrbracket_{\pi^j}^{\mathcal{K}} = \text{T} \\ \text{F sonst} \end{cases}$$

Veranschaulichung der Semantik der temporalen Operatoren der LTL

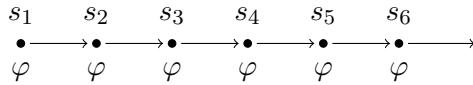
- $\circ \varphi$ bedeutet, dass φ im nächsten Zustand gilt:



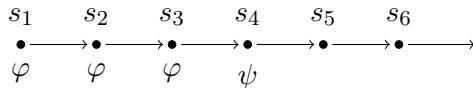
- $\Diamond \varphi$ bedeutet, dass φ irgendwann auf dem Pfad gilt:



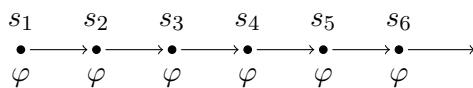
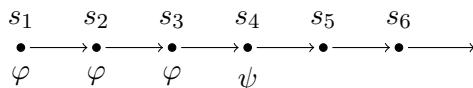
- $\Box \varphi$ bedeutet, dass φ immer auf dem Pfad gilt:



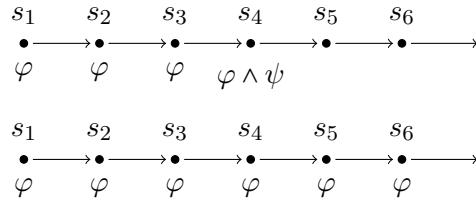
- $\varphi \mathcal{U} \psi$ bedeutet, dass ψ irgendwann auf dem Pfad gilt, und dass bis dahin auf jeden Fall φ wahr ist:



- $\varphi \mathcal{W} \psi$ bedeutet, dass φ gilt bis ψ gilt oder für immer, wenn ein solcher Zustand nicht existiert:



- $\varphi \mathcal{R} \psi$ bedeutet, dass φ gilt einschließlich dem ersten Zustand, in dem ψ gilt oder immer, wenn ein solcher Zustand nicht existiert:



Definition 19.4 (Semantik für Pfade, Zustände und Strukturen). Sei \mathcal{K} eine Kripke-Struktur. Es gilt dann

- Sei π ein *Pfad* über \mathcal{K} . Dann sagt man, dass π eine Formel φ erfüllt, geschrieben $\pi \models \varphi$, wenn gilt $\llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = T$.
- Sei s ein *Zustand* von \mathcal{K} . Dann sagt man, dass s eine Formel φ erfüllt, geschrieben $s \models \varphi$, wenn für alle Pfade π , die mit s beginnen, gilt $\pi \models \varphi$.
- Man sagt, dass die *Kripke-Struktur* \mathcal{K} eine Formel φ erfüllt, geschrieben $\mathcal{K} \models \varphi$, wenn für den Startzustand s_0 gilt: $s_0 \models \varphi$, d.h. φ gilt für alle Berechnungen in der Kripke-Struktur.

Beispiele Wenn wir zunächst das obige Beispiel 19.1 betrachten, ist leicht zu sehen, dass gilt:

- $\mathcal{K} \models \Box P$
denn in allen Zuständen ist P true.
- $s_0 \models \Diamond(P \vee Q)$
denn in s_1 gilt $P \vee Q$.
- $s_0 \models \Diamond(P \wedge Q)$
denn in s_1 gilt $P \wedge Q$.
- $s_1 \not\models \Diamond(P \wedge Q)$
denn zwar gilt in s_0 gilt $P \wedge Q$, nicht aber in s_2 .
- $\mathcal{K} \models \Box(\neg Q \rightarrow \Box(P \wedge \neg Q))$
denn der einzige Zustand mit $\neg Q$ ist s_2 , ab dann gilt aber immer $P \wedge \neg Q$.

An Beispiel 19.2 kann man sehen, dass obgleich für Pfade gilt $\pi \models \varphi \Leftrightarrow \pi \not\models \neg \varphi$ gilt, dies für Kripke-Strukturen nicht der Fall ist:

- $\mathcal{K} \not\models \Diamond P$
denn auf dem Pfad $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ ist P niemals true.
- $\mathcal{K} \not\models \neg \Diamond P$
denn auf dem Pfad $s_0 \rightarrow s_1 \rightarrow s_1 \rightarrow \dots$ ist P schließlich true.

Bemerkung. Die Semantik der linearen temporalen Logik unterscheidet sich grundlegend je nach Definition. Betrachtet man die Semantik der LTL für Pfade, dann gilt der Satz vom ausgeschlossenen Dritten, d.h. $\models_{\pi} \varphi \vee \neg\varphi$ für eine beliebige Formel φ der LTL. Für die Semantik der LTL für Kripke-Strukturen ist dies nicht der Fall. Obiges Beispiel zeigt, dass es Kripke-Strukturen geben kann, in denen weder φ noch $\neg\varphi$ gilt. Das liegt daran, dass es in einer Kripke-Struktur Pfade π_1 und π_2 geben kann, für die $\pi_1 \models \varphi$ und $\pi_2 \models \neg\varphi$ gilt.²

Für die Definition von Erfüllbarkeit, Allgemeingültigkeit und semantischer Äquivalenz wird die Pfad-Semantik zugrundegelegt.

Definition 19.5 (Erfüllbarkeit, Allgemeingültigkeit).

- Eine Formel φ heißt *erfüllbar*, wenn es eine Kripke-Struktur gibt mit einem Pfad π so dass gilt $\pi \models \varphi$.
- Eine Formel φ heißt *allgemeingültig*, wenn für alle Pfade in allen Kripke-Strukturen gilt: $\pi \models \varphi$.

Definition 19.6 (Semantische Äquivalenz). Zwei Formeln φ und ψ sind *semantisch äquivalent*, geschrieben $\varphi \equiv \psi$, wenn für alle Pfade π gilt: $\pi \models \varphi \Leftrightarrow \pi \models \psi$.

² Dass der Satz vom ausgeschlossenen Dritten (*Tertium non datur*) *nicht* richtig ist, gilt auch für die Kripke-Semantik der intuitionistischen Logik. Diese Semantik wird über Kripke-Strukturen definiert, die zusätzlich zu unserer Definition die Eigenschaft der *Monotonie* haben. Diese Eigenschaft bedeutet, dass in jedem Folgezustand s' eines beliebigen Zustands s gilt: $L(s) \subseteq L(s')$. D.h. also, dass mit jedem Zustandsübergang mehr atomare Aussagen wahr werden.

Die Semantik der Junktoren der intuitionistischen Aussagenlogik werden dann in den Begriffen der LTL so definiert:

- $P \wedge Q \stackrel{\text{def}}{=} \mathcal{K} \models P \wedge Q$
- $P \vee Q \stackrel{\text{def}}{=} \mathcal{K} \models P \vee Q$
- $\neg P \stackrel{\text{def}}{=} \mathcal{K} \models \neg \Diamond P$
- $P \rightarrow Q \stackrel{\text{def}}{=} \mathcal{K} \models \Box(P \rightarrow Q)$

Siehe [Bor05, Chap. 9] sowie [vd13, Section 6.3].

19.2 Äquivalenzen von Formeln der LTL (in der Pfad-Semantik)

- Dualität

$$\begin{aligned}\neg \circ \varphi &\equiv \circ \neg \varphi \\ \neg \diamond \varphi &\equiv \square \neg \varphi \\ \neg \square \varphi &\equiv \diamond \neg \varphi\end{aligned}$$

- Idempotenz

$$\begin{aligned}\diamond \diamond \varphi &\equiv \diamond \varphi \\ \square \square \varphi &\equiv \square \varphi \\ \varphi \mathcal{U} (\varphi \mathcal{U} \psi) &\equiv \varphi \mathcal{U} \psi \\ (\varphi \mathcal{U} \psi) \mathcal{U} \psi &\equiv \varphi \mathcal{U} \psi\end{aligned}$$

- Absorption

$$\begin{aligned}\diamond \square \diamond \varphi &\equiv \square \diamond \varphi \\ \square \diamond \square \varphi &\equiv \diamond \square \varphi\end{aligned}$$

- Expansion

$$\begin{aligned}\varphi \mathcal{U} \psi &\equiv \psi \vee (\varphi \wedge \circ (\varphi \mathcal{U} \psi)) \\ \diamond \varphi &\equiv \varphi \vee \circ \diamond \varphi \\ \square \varphi &\equiv \varphi \wedge \circ \square \varphi\end{aligned}$$

- Distributiv-Gesetze

$$\begin{aligned}\diamond (\varphi \vee \psi) &\equiv \diamond \varphi \vee \diamond \psi \\ \square (\varphi \wedge \psi) &\equiv \square \varphi \wedge \square \psi \\ \circ (\varphi \mathcal{U} \psi) &\equiv \circ \varphi \mathcal{U} \circ \psi\end{aligned}$$

19.3 Typische Aussagen in der LTL

Die lineare temporale Logik eignet sich besonders gut dafür, Eigenschaften von Programmen, insbesondere mit Nebenläufigkeit, auszudrücken. In diesem Abschnitt sollen die wichtigsten dieser Eigenschaften vorgestellt und als Formeln in der LTL ausgedrückt werden.

Als Beispiel für die Formulierung der Eigenschaften werden wir eine Ampelschaltung an einer einspurigen Baustelle verwenden, wir verwenden

zwei Ampeln, die sich verhalten wie die Ampel in Abbildung 17.1. Als zweites Beispiel stellen wir uns ein Programm mit einem kritischen Abschnitt (*critical section*) vor, in dem wechselseitiger Ausschluss zweier Transaktionen, Prozesse oder Threads erforderlich ist.

19.3.1 Sicherheitseigenschaft

Eine *Sicherheitseigenschaft* φ ist erfüllt, wenn sie in jedem Zustand jedes Berechnungspfads eines Programms erfüllt ist:

$$\Box \varphi$$

Oder mit Negation informell ausgedrückt: „etwas Unerwünschtes darf niemals passieren“, also $\neg\Diamond\neg\varphi$.

Für das Beispiel der beiden Ampeln A und B sind Sicherheitseigenschaften:

$$\begin{aligned}\Box\neg(Rot_A \wedge Gruen_A) \\ \Box\neg(Gruen_A \wedge Gruen_B)\end{aligned}$$

Wenn wir Symbole $Crit_1$ und $Crit_2$ für den Aufenthalt zweier Prozesse im kritischen Abschnitt $Crit$ nehmen, dann wird die Sicherheitseigenschaft des wechselseitigen Ausschlusses so formuliert:

$$\Box(\neg Crit_1 \vee \neg Crit_2)$$

19.3.2 Lebendigkeitseigenschaft

Eine *Lebendigkeitseigenschaft* φ ist erfüllt, wenn sie schließlich eintritt.

$$\Diamond \varphi$$

Oder informell ausgedrückt: „Etwas Erwünschtes wird passieren“.

Lebendigkeitseigenschaften an unseren Beispielen:

$$\begin{aligned}\Diamond Gruen_A \wedge \Diamond Gruen_B \\ \Diamond Crit_1 \wedge \Diamond Crit_2\end{aligned}$$

19.3.3 Fairness

Während man Sicherheits- und Lebendigkeitseigenschaften offensichtlich auch für eine einzelnen Prozess (ohne Nebenläufigkeit) formulieren kann,

geht es bei der Fairness darum, dass es fair dabei zugeht, wenn mehrere Prozesse darum konkurrieren, als nächstes einen Schritt ausführen zu können.

Im Beispiel des wechselseitigen Ausschlusses zweier Prozesse nehmen wir an, dass binäre Semaphoren verwendet werden und wir geben der Aktion, bei der ein Prozess darauf wartet, den kritischen Abschnitt zu betreten, das Aussagensymbol $Wait_1$ bzw. $Wait_2$ (siehe [BK08, Abschnitt 2.2.2]).

Im Allgemeinen soll ψ die Formel bezeichnen, die ausdrückt, dass versucht wird φ zu erreichen.

Man unterscheidet drei Formen der Fairness:

19.3.3.1 Unbedingte Fairness

Unbedingte Fairness ist gegeben, wenn φ immer wieder erfüllt ist:

$$\square \diamond \varphi$$

Oder informell ausgedrückt: „Etwas kommt immer wieder dran“.

In unseren Beispielen:

$$\begin{aligned} & \square \diamond Gruen_A \wedge \square \diamond Gruen_B \\ & \square \diamond Crit_1 \wedge \square \diamond Crit_2 \end{aligned}$$

19.3.3.2 Schwache Fairness

Schwache Fairness ist gegeben, wenn schließlich ψ immer gilt, dass dann auch φ immer wieder erfüllt ist:

$$\diamond \square \psi \rightarrow \square \diamond \varphi$$

Oder informell ausgedrückt: „Wenn schließlich etwas immerzu versucht wird, dann wird es unendlich oft gelingen“.

Im Beispiel des wechselseitigen Ausschlusses:

$$(\diamond \square Wait_1 \rightarrow \square \diamond Crit_1) \wedge (\diamond \square Wait_2 \rightarrow \square \diamond Crit_2)$$

19.3.3.3 Starke Fairness

Starke Fairness ist gegeben, wenn immer wieder ψ gilt, dass dann auch φ immer wieder erfüllt ist:

$$\square \diamond \psi \rightarrow \square \diamond \varphi$$

Oder informell ausgedrückt: „Wenn etwas immer wieder mal versucht wird, dann wird es unendlich oft gelingen“.

Im Beispiel des wechselseitigen Ausschlusses:

$$(\square \diamond \text{Wait}_1 \rightarrow \square \diamond \text{Crit}_1) \wedge (\diamond \square \text{Wait}_2 \rightarrow \square \diamond \text{Crit}_2)$$

19.4 Büchi-Automaten

Formeln der linearen temporalen Logik und Kripke-Strukturen hängen eng zusammen mit Büchi³-Automaten. Deshalb soll es in diesem Abschnitt um Büchi-Automaten gehen.

19.4.1 Automaten und unendliche Wörter

Büchi-Automaten sind endliche Automaten, bei denen es darum geht, dass sie *unendliche* Folgen von Symbolen erkennen. Dabei verwendet man für die Akzeptanz für einen unendlichen Lauf auf dem endlichen Automaten die sogenannte *Büchi-Bedingung*, die besagt, dass der Lauf akzeptiert wird, wenn er unendlich oft akzeptierende Zustände durchläuft.

Definition 19.7. Ein *Büchi-Automat* ist ein Tupel $\mathcal{A} = (\Sigma, Q, \Delta, q_I, F)$ mit

- Σ ist ein endliches *Alphabet*,
- Q ist eine endliche Menge von *Zuständen*,
- $\Delta \subseteq Q \times \Sigma \times Q$ ist die *Transitionsrelation*,
- q_I ist ein ausgezeichneter *Initialzustand* und
- $F \subseteq Q$ ist eine ausgezeichnete Menge *akzeptierender Zustände*.

Sei v ein unendliche Folge von Symbolen des Alphabets, d.h. $v = a_0a_1a_2\dots$ mit $a_i \in \Sigma$. Man schreibt dann $v \in \Sigma^\omega$ und bezeichnet die Folge als ω -Wort. Ein *Lauf* von \mathcal{A} auf einem Wort v ist eine unendliche Folge von Zuständen $\rho = q_0q_1q_2\dots$ beginnend mit dem Initialzustand q_I und die Transitionsrelation respektierend, d.h. für alle Paare von aufeinanderfolgenden Zuständen gilt $(q_i, a_i, q_{i+1}) \in \Delta$.

Mit $\text{Inf}(\rho)$ bezeichnet man die Menge der Zustände, die unendlich oft im Lauf ρ vorkommen. Weil der Lauf unendlich ist, der Automat aber nur endlich viele Zustände hat, muss $\text{Inf}(\rho) \neq \emptyset$ gelten.

Ein Lauf ρ ist *akzeptierend*, wenn $\text{Inf}(\rho) \cap F \neq \emptyset$ ist, also wenn mindestens ein akzeptierender Zustand unendlich oft durchlaufen wird.

³ nach Julius Richard Büchi (1924–1984), Schweizer Logiker und Mathematiker.

Die Sprache $L(\mathcal{A})$ des Büchi-Automaten \mathcal{A} ist die Menge aller ω -Wörtern, für die es einen akzeptierenden Lauf gibt.

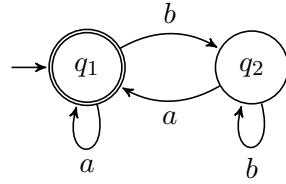


Abbildung 19.3: Beispiel eines Büchi-Automaten

Beispiel 19.1. Abbildung 19.3 zeigt einen Büchi-Automaten, der alle Worte über dem Alphabet $\{a, b\}$ akzeptiert, die unendliche viele as enthalten, wie etwa das Wort $(ab)^\omega$, die unendliche alternierende Sequenz der beiden Buchstaben. Die Sprache, die der Automat akzeptiert, wird durch den ω -regulären Ausdruck $(b^*a)^\omega$ beschrieben.

Zu einer Kripke-Struktur kann man einen Büchi-Automaten auf kanonische Weise finden. Die Zustände sind gerade die Zustände der Kripke-Struktur und die Übergänge in einen Zustand sind genau mit den Teilmengen der Potenzmenge der Atome in der Kripke-Struktur markiert, die in diesem Zustand gelten müssen. Da die Semantik der LTL in einer Kripke-Struktur so definiert ist, dass eine Formel auf allen Berechnungspfaden gelten muss, sind im korrespondierenden Büchi-Automaten alle Zustände akzeptierend sind. Präzise:

Sei \mathcal{K} die Kripke-Struktur (S, s_o, \rightarrow, L) dann korrespondiert dazu der Büchi-Automat \mathcal{A} mit

- dem Alphabet $\Sigma = \mathbb{P}((P))$, der Potenzmenge der Menge der Aussagensymbole \mathcal{P} der Kripke-Struktur,
- der Menge der Zustände $S \cap \{\iota\}$, wobei ι ein zusätzlicher Zustand ist, der für den Eintritt in den Startzustand der Kripke-Struktur steht,
- der Transitionsrelation Δ mit

$$(s, \alpha, s') \in \Delta \text{ für } s, s' \in S \Leftrightarrow (s, s') \in \rightarrow \text{ und } \alpha = L(s)$$

$$(\iota, \alpha, s) \in \Delta \Leftrightarrow s = s_o \text{ und } \alpha = L(s_o)$$

- dem Initialzustand $q_I = \iota$ und
- alle Zuständen $F = S \cap \{\iota\}$ als akzeptierende Zustände.

Beispiel 19.2. Abbildung 19.4 zeigt links eine Kripke-Struktur und daneben den entsprechenden Büchi-Automaten.

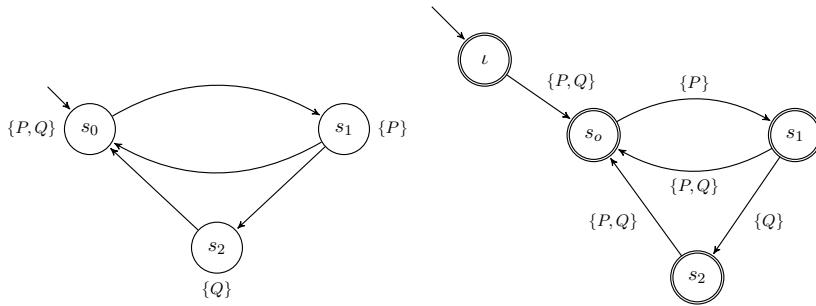


Abbildung 19.4: Kripke-Struktur und entsprechender Büchi-Automat

Wesentlich ist nun die Beobachtung, dass einer Formel φ , die in der Kripke-Struktur erfüllt ist, gerade die Läufe entsprechen, die vom korrespondierenden Büchi-Automat akzeptiert werden. Wenn man im Beispiel von Abbildung 19.4 die Formel $\square \diamond P$ betrachtet, so ist sie in der Kripke-Struktur wahr, denn auf allen Berechnungspfaden wird immer wieder der Zustand s_1 erreicht, in dem $P = T$ gilt. Wenn man nun andererseits einen beliebigen Lauf im korrespondierenden Büchi-Automat betrachtet, dann ist der Zustand s_1 ein akzeptierender Zustand, der in jedem möglichen Lauf unendlich oft durchlaufen wird, das bedeutet, dass in einem Lauf immer wieder der Übergang vorkommt, der mit $\{P\}$ markiert ist.

19.4.2 Eigenschaften von Büchi-Automaten

Man unterscheidet zwischen *deterministischen* und *nichtdeterministischen* Büchi-Automaten. In der Definition 19.7 können in der Transitionsrelation Δ zwei Tupel (q, a, r) und (q, a, r') vorkommen, was bedeutet dass Nichtdeterminismus in der Definition erlaubt ist. Wenn es zu einem Buchstaben in jedem Zustand nur einen möglichen Übergang gibt, dann ist der Automat deterministisch.

Handelt es sich um endliche Worte und endliche Automaten, dann kann zu jedem nichtdeterministischen Automat ein deterministischer Automat konstruiert werden, der dieselbe Sprache erkennt. Bei Büchi-Automaten ist dies nicht so [HL11, Abschnitt 1.1.3 und 5.2].

Büchi-Automaten sind abgeschlossen bezüglich der Bildung des Durchschnitts. Das bedeutet, dass es einen Büchi-Automaten \mathcal{A} gibt, der exakt den Durchschnitt der Sprachen zweier Automaten $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ akzeptiert. Der Büchi-Automat \mathcal{A} wird konstruiert als sogenanntes *synchronisiertes Produkt* von \mathcal{A}_1 und \mathcal{A}_2 , siehe [HL11, Satz 5.14].

Hat man zwei Büchi-Automaten $\mathcal{A}_1 = (\Sigma, Q_1, \Delta_1, q_1, F_1)$ und $\mathcal{A}_2 = (\Sigma, Q_2, \Delta_2, q_2, F_2)$ gegeben, dann definiert man das synchronisierte Produkt $\mathcal{A}_1 \cap \mathcal{A}_2$ folgendermaßen:

- $\mathcal{A}_1 \cap \mathcal{A}_2 = (\Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta, \langle q_1, q_2, 0 \rangle, F_1 \times F_2 \times \{2\})$
- $(\langle q_i, q_j, z \rangle a \langle q_k, q_l, z' \rangle) \in \Delta \Leftrightarrow (q_i, a, q_k) \in \Delta_1 \text{ und } (q_j, a, q_l) \in \Delta_2$ und für z und z' gilt
 - $z = 0 \wedge q_k \in F_1 \Rightarrow z' = 1$
 - $z = 1 \wedge q_l \in F_2 \Rightarrow z' = 2$
 - $z = 2 \Rightarrow z' = 0$
 - sonst $z' = z$

Die Idee besteht darin, dass man die beiden Automaten gewissermaßen parallel durchläuft und durch die Verbindungen sicherstellt, dass akzeptierende Zustände von *beiden* Automaten unendlich oft durchlaufen werden müssen.

Diese Idee wird in Abbildung 19.5 illustriert. Die beiden Automaten erkennen Läufe mit unendlich vielen *as*, bzw. *bs*. Das synchronisierte Produkt ist ein Automat für unendlich viele *as* und *bs*. In der Abbildung von $\mathcal{A}_1 \cap \mathcal{A}_2$ sind die Zustände, die gar nicht erreicht werden können, nicht eingezeichnet.

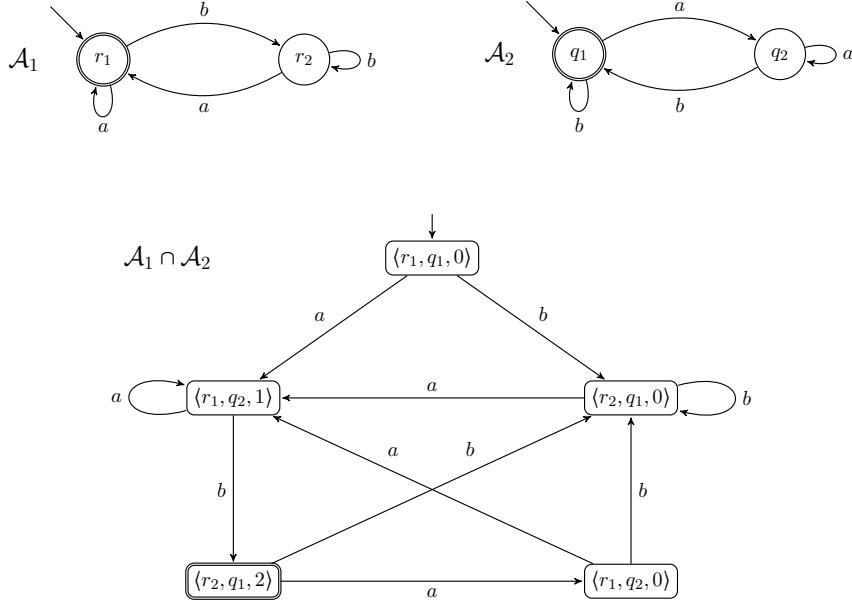


Abbildung 19.5: Beispiel für das synchronisierte Produkt

Nicht-deterministische Büchi-Automaten sind auch abgeschlossen unter der Komplementbildung. Das Komplement zum Automaten \mathcal{A} ist ein Büchi-Automat, der die Sprache $\mathcal{L}(\mathcal{A}) = \Sigma^\omega - \overline{\mathcal{L}(\mathcal{A})}$ akzeptiert. In [HL11, Abschnitt 6.2] wird zu einem Büchi-Automat \mathcal{A} ein Automat \mathcal{A}' konstruiert mit $\mathcal{L}(\mathcal{A}') = \overline{\mathcal{L}(\mathcal{A})}$.

19.4.3 Das Leerheitsproblem für Büchi-Automaten

Die Frage, ob die von einem Automaten erkannte Sprache leer ist, also ob $\mathcal{L}(\mathcal{A}) = \emptyset$ gilt, nennt man das *Leerheitsproblem*.

Ein gerichteter Graph heißt *stark zusammenhängend*, wenn es zwischen allen Knoten einen gerichteten Weg gibt. Anders ausgedrückt, jedes beliebige Paar von Knoten im Graphen ist in der reflexiv-transitiven Hülle des Graphen enthalten. Eine *starke Zusammenhangskomponente* ist ein Teilgraph, der stark zusammenhängend ist.

Die von einem Büchi-Automaten erkannte Sprache ist nicht leer, wenn es einen starke Zusammenhangskomponente gibt, die erreichbar ist und einen akzeptierenden Zustand enthält. Das bedeutet dasselbe wie: es gibt einen erreichbaren akzeptierenden Zustand, der Teil eines Zyklus ist.

Üblicherweise werden die starken Zusammenhangskomponenten eines Graphen mit dem Algorithmus von Tarjan⁴ ermittelt. In [CGP99, Abschnitt 9.3] wird ein Algorithmus beschrieben, der in einem Büchi-Automaten Zyklen mit einem akzeptierenden Zustand findet.

19.4.4 Verallgemeinerte Büchi-Automaten

Oft ist es praktisch mit verallgemeinerten Büchi-Automaten zu arbeiten.

Definition 19.8. Ein *verallgemeinerter Büchi-Automat* ist ein Tupel $(\mathcal{A}) = (\Sigma, Q, \Delta, I, F_1, F_2, \dots, F_k)$ wie bereits definiert, jedoch mit einer Menge $I \subseteq Q$ von Initialzuständen und mehreren Mengen $F_1, F_2, \dots, F_k \subseteq Q$ akzeptierender Zustände.

Ein Lauf eines verallgemeinerten Büchi-Automaten auf einem unendlichen Wort kann im Unterschied zu den bisher betrachteten Automaten in einem beliebigen Zustand aus der Menge I beginnen. Ein Lauf q_0, q_1, \dots heißt akzeptierend, wenn es für alle $i = 1, 2, \dots, k$ ein $q \in F_i$ gibt, so dass für unendlich viele j gilt: $q = q_j$.

Ein verallgemeinerter Büchi-Automat hat also mehrere Initialzustände und mehrere Mengen akzeptierender Zustände. Die Akzeptanzbedingung besteht darin, dass jede dieser Mengen unendlich oft besucht werden muss.

Man kann einen verallgemeinerten Büchi-Automaten leicht zu einem Büchi-Automaten mit einem Initialzustand und einer einzigen Menge von akzeptierenden Zuständen machen.

⁴ Robert Tarjan, amerikanischer Informatiker.

Gegeben sei $\mathcal{A} = (\Sigma, Q, \Delta, I, F_1, \dots, F_k)$. Man konstruiert daraus $\mathcal{A}' = (\Sigma, Q \times \{0, 1, \dots, k\}, \Delta', I \times \{0\}, Q \times \{k\})$ mit folgender Transitionsrelation Δ' :

$((q, z), a, (q', z')) \in \Delta'$, wenn $(q, a, q') \in \Delta$ ist und für z und z' gilt

- Ist $q' \in F_i$ und $z = i - 1$, dann ist $z' = i$.
- Ist $z = k$, dann ist $z' = 0$.
- In allen anderen Fällen ist $z' = z$.

Die Idee besteht darin, aus den bisherigen Zuständen Paare mit einem zusätzlichen Zähler z zu machen. Dieser Zähler gibt an, aus welcher der ursprünglichen Menge akzeptierender Zustände als nächstes ein akzeptierender Zustand durchlaufen werden soll.

Um zu einem einzigen Initialzustand zu kommen, erweitert man den Automaten durch einen neuen Initialzustand, der zu jedem der bisherigen Initialzuständen führt.

19.5 Erfüllbarkeit in der LTL

Wie wir gleich sehen werden, ist es möglich zu einer Formel der linearen temporalen Logik einen Büchi-Automaten zu konstruieren, der die Eigenschaft hat, dass die von diesem Automaten erkannte Sprache genau den Berechnungspfaden entspricht, die die Formel erfüllen.

Um die Erfüllbarkeit einer Formel der LTL zu entscheiden, konstruiert man also zu dieser Formel den korrespondierenden Büchi-Automat. Wenn dieser Automat nicht leer ist, ist die Formel erfüllbar.

Um eine Kripke-Struktur zu finden, in der die Formel wahr wird, nimmt man einen Pfad im Büchi-Automaten mit einem Zyklus, der eine akzeptierenden Zustand enthält. Man kann dann den Büchi-Automaten als Kripke-Struktur interpretieren, indem man aus den Aussagen, die die Übergänge des Automaten markieren, die Aussagen ermittelt, die im Zielzustand zutreffen müssen.

19.5.1 Negationsnormalform (NNF) in der LTL

Für die Konstruktion eines Büchi-Automaten zu einer Formel ist es nötig, in der Formel folgende Junktoren zu erlauben: $\neg, \wedge, \vee, \circ, \mathcal{U}, \mathcal{R}$ sowie die beiden Konstanten \perp und \top .

Definition 19.9. Eine Formel φ der LTL ist in *Negationsnormalform*, wenn die Negation \neg nur in Literalen vorkommt.

Um die Negationsnormalform von φ zu erreichen, müssen wir zunächst die Implikation beseitigen. Dazu eignet sich die Funktion `IMPL_FREE` aus Abschnitt 6.1.

Ist die Formel ohne Implikationen, erhält man die Negationsnormalform mit folgender Funktion:

```
function NNFLTL( $\varphi$ ) {
    // pre:  $\varphi$  hat keine Implikationen
    // post: äquivalente Umformung von  $\varphi$  in NNF
    case {
         $\varphi$  ist Literal oder eine Konstante:
            return  $\varphi$ ;
         $\varphi$  hat die Form  $\neg\neg\varphi_1$ :
            return NNFLTL( $\varphi_1$ );
         $\varphi$  hat die Form  $\varphi_1 \wedge \varphi_2$ :
            return NNFLTL( $\varphi_1$ )  $\wedge$  NNFLTL( $\varphi_2$ );
         $\varphi$  hat die Form  $\varphi_1 \vee \varphi_2$ :
            return NNFLTL( $\varphi_1$ )  $\vee$  NNFLTL( $\varphi_2$ );
         $\varphi$  hat die Form  $\neg(\varphi_1 \wedge \varphi_2)$ :
            return NNFLTL( $\neg\varphi_1$ )  $\vee$  NNFLTL( $\neg\varphi_2$ );
         $\varphi$  hat die Form  $\neg(\varphi_1 \vee \varphi_2)$ :
            return NNFLTL( $\neg\varphi_1$ )  $\wedge$  NNFLTL( $\neg\varphi_2$ );
         $\varphi$  hat die Form  $\neg\circ\varphi_1$ :
            return  $\circ$  NNFLTL( $\neg\varphi_1$ );
         $\varphi$  hat die Form  $\neg(\varphi_1 \mathcal{U} \varphi_2)$ :
            return NNFLTL( $\neg\varphi_1$ )  $\mathcal{R}$  NNFLTL( $\neg\varphi_2$ );
         $\varphi$  hat die Form  $\neg(\varphi_1 \mathcal{R} \varphi_2)$ :
            return NNFLTL( $\neg\varphi_1$ )  $\mathcal{U}$  NNFLTL( $\neg\varphi_2$ );
    }
}
```

19.5.2 Von LTL zum Büchi-Automat

In diesem Abschnitt wird der Algorithmus von Gerth, Peled, Vardi und Wolper [GPVW95] vorgestellt, der eine Formel der LTL in Negationsnormalform in einen Büchi-Automaten transformiert.

Die Grundidee der Konstruktion des Automaten aus dem Formel besteht darin, dass die Formel von ihrer Struktur her so zerlegt wird, dass sich ein Teil der Aussage auf den aktuellen Zustand und ein anderer Teil der Aussage auf den nächsten Zustand bezieht. Dieses Vorgehen erklärt auch, weshalb in der Negationsnormalform gerade die binären Junktoren \mathcal{U} und \mathcal{R} gewählt wurden. Denn für diese beiden Operatoren hat man die folgenden Expansionsstrategien, auch *Abwicklung* genannt:

$$\begin{aligned}\varphi_1 \mathcal{U} \varphi_2 &\leftrightarrow \varphi_2 \vee (\varphi_1 \wedge \circ(\varphi_1 \mathcal{U} \varphi_2)) \\ \varphi_1 \mathcal{R} \varphi_2 &\leftrightarrow \varphi_2 \wedge (\varphi_1 \vee \circ(\varphi_1 \mathcal{R} \varphi_2))\end{aligned}$$

Für den Algorithmus wird die Datenstruktur $node = [id, incoming, old, new, next]$ verwendet, wobei

- id ist eine eindeutige Id des Knotens.
- $incoming$ ist eine Liste von Ids von Knoten, die eine Kante zu diesem Knoten haben.
- old enthält die Teilformeln von φ , die bereits vom Algorithmus berechnet wurden.
- new enthält die Teilformeln von φ , die im aktuellen Schritt berechnet werden.
- $next$ enthält die Teilformeln, die im nächsten Schritt berechnet werden müssen.

Als Hilfsfunktion wird die Funktion `NEW_ID()` verwendet, die bei jedem Aufruf eine frische Id generiert.

Der Algorithmus wird initialisiert mit einem Knoten $node_0$, der eine Kante von einem speziellen Zustand namens *init* hat, welcher später der Initialzustand des Automaten werden wird:

$$\begin{aligned}node_0 := [id := \text{NEW_ID}(), \\ incoming := \{\text{init}\}, \\ old := \emptyset, \\ new := \{\varphi\}, \\ next := \emptyset]\end{aligned}$$

Die rekursive Funktion $nodes' = \text{EXPAND}(q, nodes)$ bearbeitet den Knoten q und die bisherige Liste $nodes$ von bereits konstruierten Knoten. Sie gibt eine Liste von Knoten zurück. D.h. der Graph wird erzeugt durch $\text{expand}(node_0, \emptyset)$.

In der Funktion $\text{expand}(q, nodes)$ sind folgende Möglichkeiten zu unterscheiden:

- Das Feld new von q ist leer.

- Es gibt einen Knoten $r \in nodes$ mit $r.old = q.old$ und $r.next = q.next$
dann tritt r an die Stelle von q , d.h.
 $r.incoming = r.incoming \cup q.incoming$
 $\text{return } nodes.$
- andernfalls muss man einen neuen Knoten einfügen:

$$\begin{aligned} node_{new} := & [id =: \text{NEW_ID}(), \\ & incoming := \{q.id\}, \\ & old := \emptyset, \\ & new := q.next, \\ & next := \emptyset] \end{aligned}$$

Die Rekursion geht weiter mit $\text{EXPAND}(node_{new}, nodes \cup \{q\})$.

- $q.new$ ist nicht leer.
Dann nehmen wir eine Formel $\psi \in q.new$ und setzen
 $q.new := q.new - \psi$.
 - Es kann sein, dass $\psi \in q.old$ gilt
Die Rekursion geht weiter mit $\text{EXPAND}(q, nodes)$
 - Es kann sein, dass $\neg\psi \in q.old$ ist oder $\psi = \perp$
Wir sind auf einen Widerspruch gestoßen, d.h. der aktuelle Knoten muss verworfen werden:
 $\text{return } nodes.$
 - andernfalls werden Knoten q' bzw. q_1 und q_2 erzeugt, je nach Struktur der Formel ψ und die Rekursion geht dann weiter mit
 $\text{EXPAND}(q', nodes)$ bzw. $\text{EXPAND}(q_2, \text{EXPAND}(q_1, nodes))$

Nun müssen wir noch sehen, wie im letzteren Fall der Knoten q' gebildet wird oder unser Knoten gespalten wird in q_1 und q_2 . Dazu müssen wir uns die Struktur der Formel ansehen:

- ψ ist ein Literal oder $\psi = \top$:

$$\begin{aligned} q' := & [id =: \text{NEW_ID}(), \\ & incoming := q.incoming, \\ & old := q.old \cup \{\psi\}, \\ & new := q.new, \\ & next := q.next] \end{aligned}$$

- ψ hat die Form $\psi = \psi_1 \vee \psi_2$:

$$\begin{array}{ll} q_1 := [id := \text{NEW_ID}(), & q_2 := [id := \text{NEW_ID}() \\ & \quad incoming := q.incoming, \quad incoming := q.incoming, \\ & \quad old := q.old \cup \{\psi\}, \quad old := q.old \cup \{\psi\}, \\ & \quad new := q.new \cup \{\psi_1\}, \quad new := q.new \cup \{\psi_2\}, \\ & \quad next := q.next] \quad next := q.next] \end{array}$$

- ψ hat die Form $\psi = \psi_1 \wedge \psi_2$:

$$q' := [id := \text{NEW_ID}(), \\ \quad incoming := q.incoming, \\ \quad old := q.old \cup \{\psi\}, \\ \quad new := q.new \cup \{\psi_1, \psi_2\}, \\ \quad next := q.next]$$

- ψ hat die Form $\psi = \psi_1 \mathcal{U} \psi_2$:

$$\begin{array}{ll} q_1 := [id := \text{NEW_ID}(), & q_2 := [id := \text{NEW_ID}() \\ & \quad incoming := q.incoming, \quad incoming := q.incoming, \\ & \quad old := q.old \cup \{\psi\}, \quad old := q.old \cup \{\psi\}, \\ & \quad new := q.new \cup \{\psi_1\}, \quad new := q.new \cup \{\psi_2\}, \\ & \quad next := q.next \cup \{\psi_1 \mathcal{U} \psi_2\}] \quad next := q.next] \end{array}$$

- ψ hat die Form $\psi = \psi_1 \mathcal{R} \psi_2$:

$$\begin{array}{ll} q_1 := [id := \text{NEW_ID}(), & q_2 := [id := \text{NEW_ID}() \\ & \quad incoming := q.incoming, \quad incoming := q.incoming, \\ & \quad old := q.old \cup \{\psi\}, \quad old := q.old \cup \{\psi\}, \\ & \quad new := q.new \cup \{\psi_1, \psi_2\}, \quad new := q.new \cup \{\psi_2\}, \\ & \quad next := q.next] \quad next := q.next \cup \{\psi_1 \mathcal{R} \psi_2\}] \end{array}$$

- ψ hat die Form $\psi = \circ \psi_1$:

$$q' := [id := \text{NEW_ID}(), \\ \quad incoming := q.incoming, \\ \quad old := q.old \cup \{\psi\}, \\ \quad new := q.new, \\ \quad next := q.next \cup \{\psi_1\}]$$

Beispiel 19.3. Als Beispiel wollen wir den Algorithmus mit der Formel $\varphi = (P \mathcal{U} (Q \wedge R))$ durchspielen:

Der Algorithmus beginnt mit dem initialen Knoten n_0 , der die Formel selbst repräsentiert: $[0, \{init\}, \{\}, \{P \mathcal{U} (Q \wedge R)\}, \{\}]$.

id incoming old new next

Der Ergebnisgraph ist leer: $nodes = \{\}$.

Schritt 1: Die zu bearbeitende Formel hat \mathcal{U} als Haupt-Junktor, d.h. es entstehen zwei neue Knoten:

$$n_1 = [1, \{init\}, \{P \mathcal{U} (Q \wedge R)\}, \{P\}, \{P \mathcal{U} (Q \wedge R)\}]$$

id incoming old new next

$$n_2 = [2, \{init\}, \{P \mathcal{U} (Q \wedge R)\}, \{Q \wedge R\}, \{\}]$$

id incoming old new next

Nun muss man erst n_1 rekursiv expandieren und dann mit dem dadurch entstehenden Ergebnisgraph mit der Expansion von n_2 in Schritt 13 fortfahren.

Schritt 2: Die zu bearbeitende Formel kommt nicht in $n_1.old$ vor. Sie ist ein Literal, somit muss ein neuer Knoten erzeugt werden:

$$n_3 = [3, \{init\}, \{P \mathcal{U} (Q \wedge R), P\}, \{\}, \{P \mathcal{U} (Q \wedge R)\}]$$

id incoming old new next

Schritt 3: Nun muss n_3 verarbeitet werden. $n_3.new$ ist leer und $nodes$ ist bisher auch leer, also muss man einen neuen Knoten einfügen:

$$n_4 = [4, \{n_3\}, \{\}, \{P \mathcal{U} (Q \wedge R)\}, \{\}] \text{ und } nodes = \{init, n_3\}$$

id incoming old new next

Schritt 4: Dieser neue Knoten muss jetzt wieder für die Abwicklung des \mathcal{U} geteilt werden: $n_5 = [5, \{n_3\}, \{P \mathcal{U} (Q \wedge R)\}, \{P\}, \{P \mathcal{U} (Q \wedge R)\}]$.

id incoming old new next

$$n_6 = [6, \{n_3\}, \{P \mathcal{U} (Q \wedge R)\}, \{Q \wedge R\}, \{\}]$$

id incoming old new next

Wieder muss man zunächst n_5 verarbeiten und mit n_6 in Schritt 7 fortfahren.

Schritt 5: Die zu bearbeitende Formel kommt nicht in $n_5.old$ vor. Sie ist ein Literal, somit muss ein neuer Knoten erzeugt werden:

$$n_7 = [7, \{n_3\}, \{P \mathcal{U} (Q \wedge R), P\}, \{\}, \{P \mathcal{U} (Q \wedge R)\}]$$

id incoming old new next

Schritt 6: $n_7.new$ ist leer. Jetzt gibt es aber einen Knoten in $nodes$, dessen Felder old und new mit denen von n_7 übereinstimmen, nämlich n_3 . n_3 bekommt also eine Schleife:

$$n_3 = [3, \{init, n_3\}, \{P \mathcal{U} (Q \wedge R), P\}, \{\}, \{P \mathcal{U} (Q \wedge R)\}] \text{ und die Rekurrenz bricht in diesem Zweig mit } nodes = \{init, n_3\} \text{ ab.}$$

id incoming old new next

Schritt 7: Es geht mit n_6 aus Schritt 4 weiter. Die zu bearbeitende Formel hat als Haupt-Junktor das \wedge , also muss eine neuer Knoten erzeugt werden:

$$n_8 = [8, \{n_3\}, \{PU(Q \wedge R), Q \wedge R\}, \{Q, R\}, \{\}], \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Schritt 8: In der Menge n_8 beginnen wir mit der ersten Formel, der Primformel Q :

$$n_9 = [9, \{n_3\}, \{PU(Q \wedge R), Q \wedge R, Q\}, \{R\}, \{\}], \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Schritt 9: Nun muss man R verarbeiten:

$$n_{10} = [10, \{n_3\}, \{PU(Q \wedge R), Q \wedge R, Q, R\}, \{\}, \{\}], \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Schritt 10: $n_{10}.new$ ist jetzt leer und es muss ein neuer Knoten erzeugt werden:

$$n_{11} = [11, \{n_{10}\}, \{\}, \{\}, \{\}] \text{ und } nodes = \{init, n_3, n_{10}\}, \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Schritt 11: Es geht mit der Expansion von n_{11} weiter. $n_{11}.new$ ist leer und es muss ein neuer Knoten erzeugt werden:

$$n_{12} = [12, \{n_{11}\}, \{\}, \{\}, \{\}] \text{ und } nodes = \{init, n_3, n_{10}, n_{11}\}, \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Schritt 12: Bei der Verarbeitung von n_{12} tritt der Fall ein, dass wir einen Knoten in $nodes$ haben, dessen Felder old und $next$ mit denen von n_{12} übereinstimmen, nämlich mit n_{11} , also

$$n_{11} = [11, \{n_{10}, n_{11}\}, \{\}, \{\}, \{\}] \text{ und } nodes = \{init, n_3, n_{10}, n_{11}\}, \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Schritt 13: Mit Schritt 12 ist dieser Zweig beendet und mit dem bisher entstandenen Graph $nodes = \{init, n_3, n_{10}\}$ geht es mit der Expansion von n_2 aus Schritt 1 weiter:

$$n_2 = [2, \{init\}, \{PU(Q \wedge R)\}, \{Q \wedge R\}, \{\}], \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Man muss also $Q \wedge R$ verarbeiten. Jetzt wiederholt sich das Vorgehen:

$$n_{13} = [13, \{init\}, \{PU(Q \wedge R), Q \wedge R\}, \{Q, R\}, \{\}], \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Schritt 14:

$$n_{14} = [14, \{init\}, \{PU(Q \wedge R), Q \wedge R, Q\}, \{R\}, \{\}], \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Schritt 15:

$$n_{15} = [15, \{init\}, \{PU(Q \wedge R), Q \wedge R, Q, R\}, \{\}, \{\}], \begin{array}{cccc} id & incoming & old & new \\ & & & next \end{array}$$

Schritt 16: Jetzt ist $n_{15}.new$ leer und wir müssen prüfen, ob wir bereits einen Knoten haben, dessen Felder old und $next$ mit denen von n_{15} übereinstimmen. Dies ist tatsächlich der Fall, nämlich n_{10} . Also

$$n_{10} = [10, \{n_3, init\}, \{P \cup (Q \wedge R), Q \wedge R, Q, R\}, \{\}, \{\}]$$

$\begin{matrix} id & incoming & old & new & next \end{matrix}$

und $nodes = \{init, n_3, n_{10}, n_{11}\}$ wird zurückgegeben.

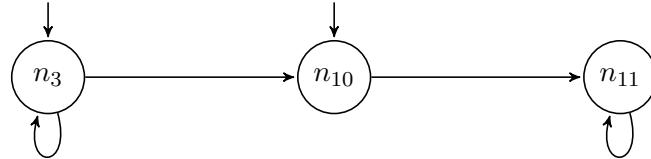


Abbildung 19.6: Graph, konstruiert zu $P \cup (Q \wedge R)$

$$n_3 = [3, \{init, n_3\}, \{P \cup (Q \wedge R), P\}, \{\}, \{P \cup (Q \wedge R)\}]$$

$$n_{10} = [10, \{n_3, init\}, \{P \cup (Q \wedge R), Q \wedge R, Q, R\}, \{\}, \{\}]$$

$\begin{matrix} id & incoming & old & new & next \end{matrix}$

$$n_{11} = [11, \{n_{10}, n_{11}\}, \{\}, \{\}, \{\}]$$

$\begin{matrix} id & incoming & old & new & next \end{matrix}$

Der konstruierte Graph wird in Abbildung 19.6 dargestellt.

Aus dem Graphen, den wir in der Liste der Knoten konstruiert haben, wird durch folgende Festlegungen ein verallgemeinerter Büchi-Automat:

- Das Alphabet Σ besteht aus Mengen von Aussagensymbolen, die in φ vorkommen.
- Die Menge Q der Zustände sind die Knoten in $nodes$ zusammen mit dem speziellen Zustand $init$.
- Ein Tupel $(q, a, q') \in \Delta$ genau dann wenn $q \in q'.incoming$ und a erfüllt alle Literale in $q'.old$.
- Der Initialzustand ist $init$.
- Für jede Teilformel der Form $\psi_1 \cup \psi_2$ gibt es eine Akzeptanzmenge F_i mit $F_i = \{q \mid \psi_2 \in q.old \text{ oder } \psi_1 \cup \psi_2 \notin r.old\}$.

Beispiel 19.4. In unserem Beispiel für die Formel $P \cup (Q \wedge R)$ ergibt sich somit:

- $\Sigma = \{P, Q, R\}$,
- $Q = \{init, n_3, n_{10}, n_{11}\}$,
- $\Delta = \{[init, \{P\}, n_3], [n_3, \{P\}, n_3], [init, \{Q, R\}, n_{10}], [n_{10}, \{Q, R\}, n_{10}], [n_{10}, \{\top\}, n_{11}], [n_{11}, \{\top\}, n_{11}]\}$,

- $init$ ist der Initialzustand und
- $F_1 = \{n_{10}, n_{11}\}$ ist die Akzeptanzmenge.

Abbildung 19.7 zeigt den Büchi-Automaten zur Formel $P \mathcal{U} (Q \wedge R)$.

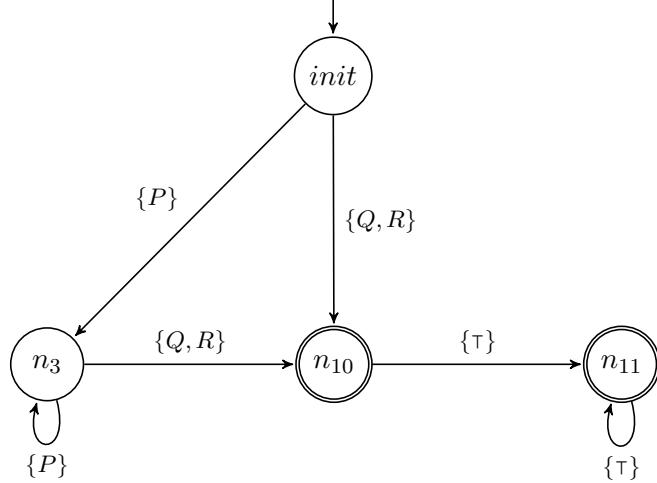


Abbildung 19.7: Büchi-Automat zu $P \mathcal{U} (Q \wedge R)$

An dem Büchi-Automaten zu einer Formel der LTL kann man unmittelbar eine Kripke-Struktur ablesen, in der die Formel wahr ist. Man nimmt einen Pfad zu einer Akzeptanzmenge, die einen Zyklus enthält. Beginnend bei $init$ interpretiert man die Markierung der Kanten des Büchi-Graphen als Zuweisung der Menge der Literale zum nächsten Knoten.

Beispiel 19.5. In unserem Beispiel der Formel $P \mathcal{U} (Q \wedge R)$ kann man so verschiedene Kripke-Strukturen ablesen, etwa die in Abbildung 19.8 dargestellte.

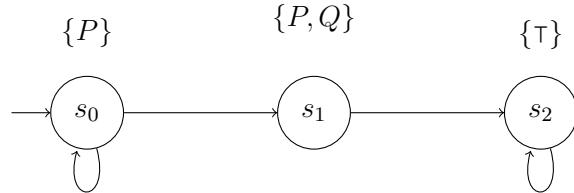


Abbildung 19.8: Kripke-Struktur, in der $P \mathcal{U} (Q \wedge R)$ gilt

Bemerkung. Mit dem dargestellten Verfahren die Erfüllbarkeit einer Formel der LTL zu entscheiden, hat man natürlich auch ein Entscheidungsverfahren für die Erfüllbarkeit der Aussagenlogik. Es handelt sich

die Methode der *semantischen Tableaux*, im Detail beschrieben z.B. in [BA12, Abschnitt 2.6].

*

In der Logic Workbench (lwb) wird für die Transformation einer Formel der LTL in einen Büchi-Automaten die Java-Bibliothek `ltl2buchi` eingesetzt. Die Bibliothek wurde implementiert von Dimitra Giannakopoulou und Flavio Lerda am [NASA Ames Research Center](#) als Komponente des Model Checkers [Java Pathfinder](#). Giannakopoulou und Lerda verwenden einen Algorithmus, der auf den oben erläuterten Algorithmus aufbaut, jedoch deutlich kleinere Büchi-Automaten generiert [GL02].

Die Funktion in der Logic Workbench ergeben in unserem Beispiel die in Abbildung 19.9 dargestellten Ergebnisse:

```
(def lex '(until P (and Q R)))

(lwb.ltl.buechi/ba lex)
;=> {:nodes [{:id 1, :accepting true} {:id 0, :init true}],
;      :edges [{:from 1, :to 1, :guard #{}, :label ""},
;               {:from 0, :to 0, :guard #{P}, :label ""},
;               {:from 0, :to 1, :guard #{R Q}, :label ""}]

(sat lex)
;=> {:atoms #{R Q P},
;      :nodes {[:s_1 #{R Q}]},
;      :initial :s_1,
;      :edges #{{[:s_1 :s_1]}}}
```

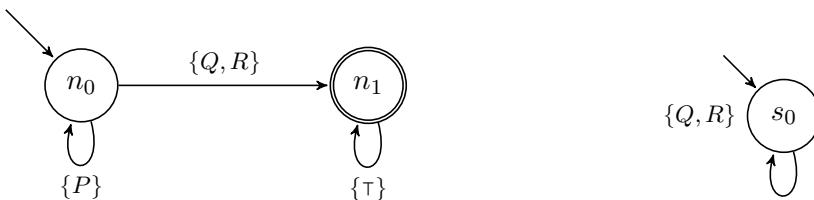


Abbildung 19.9: Mit lwb generiert: Büchi-Automat und Kripke-Struktur

19.6 Auswertung von Formeln in Kripke-Strukturen

Die Auswertung einer Formel φ der LTL in einer gegebenen Kripke-Struktur \mathcal{K} wird als *Model Checking* bezeichnet. Das Vorgehen zur Beantwortung der Frage $\mathcal{K} \models \varphi$? besteht aus folgenden Schritten (siehe auch Abbildung 19.10):

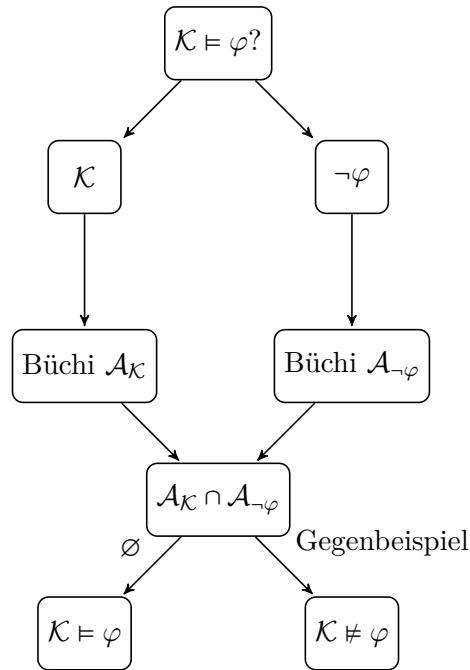


Abbildung 19.10: Evaluation von Formeln der LTL in Kripke-Strukturen

1. Zu der gegebenen Kripke-Struktur, dem Modell, konstruiert man den korrespondierenden Büchi-Automaten, siehe 19.4.1.
2. Die Negation der Formel, also zu $\neg\varphi$ wird in einen Büchi-Automaten übersetzt.
3. Nun bildet man den Durchschnitt der beiden Automaten. Wenn es einen erfolgreichen Lauf in diesem Automaten gibt, dann entspricht dies Berechnungspfaden in \mathcal{K} , die $\neg\varphi$ erfüllen.

Also: ist der Durchschnitt der beiden Automaten leer, dann ist φ in \mathcal{K} wahr. Andernfalls kann man dem Durchschnitt einen Berechnungspfad ablesen, der ein Gegenbeispiel darstellt.

Beispiel 19.6 (Wechselseitiger Ausschluss). Gegeben seien zwei Prozesse, die auf eine gemeinsame Ressource zu greifen möchten. Ihre Programm-schleife besteht darin, dass sie nicht-kritische Aktionen ausführen, dann versuchen den kritischen Bereich zu betreten und ihn schließlich wieder verlassen.

Die beiden Prozesse verwenden eine gemeinsame binäre Semaphore s . Wenn $s = 1$ gilt, dann ist die Semaphore frei, wenn $s = 0$ ist, dann ist die Semaphore gerade durch einen der beiden Prozesse belegt.

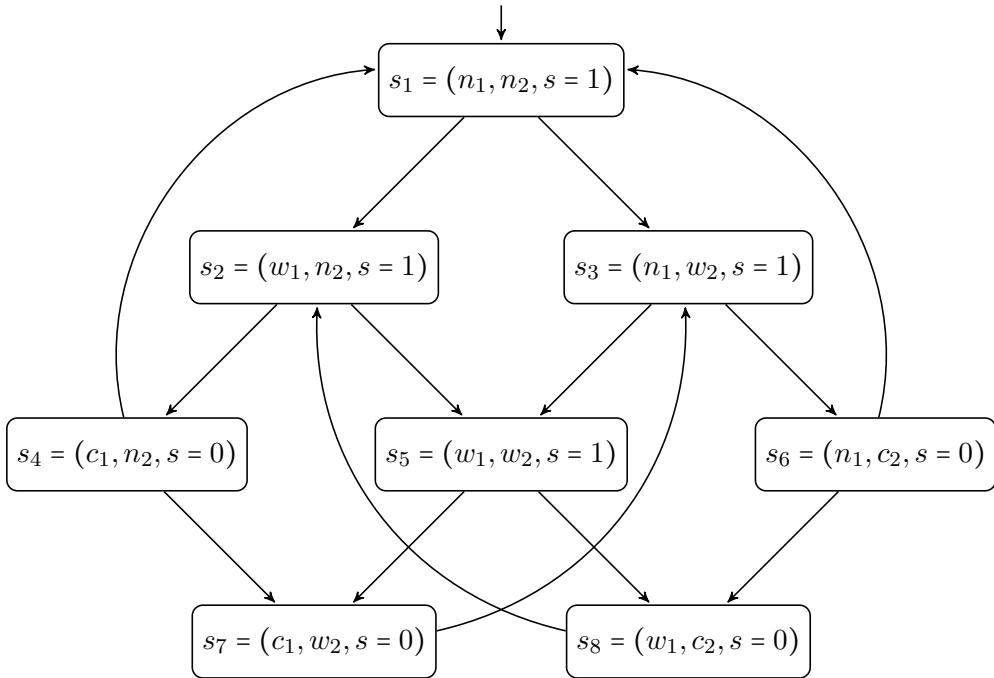


Abbildung 19.11: Kripke-Struktur zu einem Protokoll wechselseitigen Ausschlusses

In Abbildung 19.11 wird das Zusammenspiel der beiden Prozesse in einem Transitionssystem dargestellt. Dabei bezeichnet n_i , dass Prozess i nicht-kritische Aktionen ausführt, w_i , dass er auf den Zugang zu seinem kritischen Abschnitt wartet und c_i , dass er in seinem kritischen Abschnitt ist [BK08, Abschnitt 2.2].

In diesem Beispiel wollen wir nun folgende Formeln überprüfen:

- ① $\square (\neg c_1 \vee \neg c_2)$
- ② $\square \Diamond c_1 \vee \square \Diamond c_2$
- ③ $\square \Diamond c_1 \wedge \square \Diamond c_2$

Formel ① ist eine Sicherheitseigenschaft, die ausdrückt, dass niemals beide Prozesse gleichzeitig den kritischen Abschnitt verwenden dürfen. Die zweite Formel besagt, dass mindestens einer der Prozesse immer wieder den kritischen Sektor betreten kann. Und Formel ③ drückt Fairness aus: beide Prozesse können immer wieder den kritischen Abschnitt betreten.

Die Situation ist so einfach, dass man die Formeln durch genaues Hinsehen evaluieren kann. Wir wollen jedoch die Logic Workbench verwenden.

Zuerst wird die dem Transitionssystem entsprechende Kripke-Struktur in lwb definiert:

```
; Examples of algorithm for mutual exclusion based on a binary semaphore
; n = non-critical actions
; w = wait
; c = critical section
; s = semaphore s=1 true, s=0 false

(def ksx {:atoms  '#{n1 n2 w1 w2 c1 c2 s}
           :nodes   #{:s1 '#{n1 n2 s}
                      :s2 '#{w1 n2 s}
                      :s3 '#{n1 w2 s}
                      :s4 '#{c1 n2}
                      :s5 '#{w1 w2 s}
                      :s6 '#{n1 c2}
                      :s7 '#{c1 w2}
                      :s8 '#{w1 c2}}
           :initial :s1
           :edges   #{{[:s1 :s2]
                      [:s1 :s3]
                      [:s2 :s4]
                      [:s2 :s5]
                      [:s3 :s5]
                      [:s3 :s6]
                      [:s4 :s1]
                      [:s4 :s7]
                      [:s5 :s7]
                      [:s5 :s8]
                      [:s6 :s1]
                      [:s6 :s8]
                      [:s7 :s3]
                      [:s8 :s2]}}})
```

Dazu werden die verwendeten Atome definiert, die Zustände mit den Atomen, die in ihnen T sind, die anderen sind F. Ferner wird der Initialzustand festgelegt und die Zustandsübergänge.

Nun können wir Formeln in dieser Kripke-Struktur auswerten:

```
(def phi1 '(always (or (not c1) (not c2)))

(eval-phi phi1 ksx)
;=> true

(def phi2 '(or (always (finally c1)) (always (finally c2)))

(eval-phi phi2 ksx)
;=> true

(def phi3 '(and (always (finally c1)) (always (finally c2))))
```

```
(eval-phi phi3 ksx)
;=> false
(eval-phi phi3 ksx :counterexample)
;=> [:s1 :s3 :s5 :s7 :s3 :s5 :s8 :s2 :s5]
```

Die Formeln ① und ② werten zu T aus, wohingegen Formel ③ nicht zutrifft, also Fairness nicht gegeben ist. Man kann sich dann ein Gegenbeispiel anzeigen lassen. Es zeigt, dass es möglich ist, dass immer wieder der Zyklus s_5, s_8, s_2, s_5 durchlaufen werden kann und in diesem Fall wird Prozess 1 niemals mehr den kritischen Abschnitt betreten kann, er „verhungert“ in Zustand s_5 .

Kapitel 20

Natürliche Schließen in der LTL

Auch für die lineare temporale Logik ist es möglich, ein Beweissystem des natürlichen Schließens zu definieren. Für dieses Beweissystem wird sich herausstellen, dass es *korrekt* und *vollständig* ist bezogen auf die *Pfad-Semantik* (siehe 19.3).

20.1 Markierte Formeln und relationale Aussagen

Um Herleitungen in der linearen temporalen Logik machen zu können, müssen wir Aussagen über temporale Junktoren wie \circ oder \square durch Regeln begründen können. Dies beinhaltet aber immer auch einen Bezug auf den jeweiligen Zustand, so müssen wir im Beispiel von \circ von einem nächsten Zustand sprechen können.

Deshalb verwenden wir im Beweissystem für die LTL *markierte* Formeln. In diesem Formeln wird der Zustand im Pfad, auf die sie sich bezieht *explizit* gemacht.

Sei $L = \{i, j, k, \dots\}$ eine Menge von Marken sowie g eine Abbildung $L \rightarrow \mathbb{N}$.

g induziert auf L auf natürliche Weise binäre Relationen $=, <, \leq, succ$ sowie eine Funktion $',$ für die $succ(i, i')$ für alle $i \in L$ gilt. $succ$ ist die Nachfolger-Relation und die Funktion $'$ ordnet einer Marke i ihre Nachfolgerin i' zu.

Definition 20.1 (Markierte Formeln). Ist φ eine Formel der LTL und $i \in L$ eine Marke, dann schreiben wir die markierte Formel als $i : \varphi$. Sie bedeutet, dass die Formel φ im Zustand i gegeben ist.

Ferner werden wir in Herleitungen auch die Eigenschaften von L verwenden müssen. Also müssen wir solche Ausdrücke auch im Beweissystem verwenden können.

Definition 20.2 (Relationale Ausdrücke). Aussagen über Elemente von L wie etwa $i = j$, $s(i, i')$, $i \leq j$ etc. bezeichnen wir als *relationale* Ausdrücke.

Definition 20.3 (Syntax in Herleitungen in der LTL). Die nummerierten Zeilen in Herleitungen im natürlichen Schließen in der LTL enthalten

- markierte Formeln oder
- relationale Ausdrücke.

Für Herleitungen verwenden wir zur Kennzeichnung dieser speziellen Variante des natürlichen Schließens das Symbol \vdash_π .

Beispiel 20.1. Herleitung $\Diamond \neg P \vdash_\pi \neg \Box P$:

1.	$i : \Diamond \neg P$	gegeben
2.	$i : \Box P$	angenommen
3.	$i \leq j$	angenommen
4.	$j : \neg P$	abgenommen
5.	$j : P$	$\Box e$ 2, 3
6.	$j : \perp$	$\neg e$ 4, 5
7.	$j : \perp$	$\Diamond e$ 1, 3-6
8.	$i : \neg \Box P$	$\neg i$ 2-7

Zum Beweis nehmen wir in Zeile 2 an, dass $\Box P$ gilt. Da in Zeile 1 $\Diamond \neg P$ gegeben ist, gibt es ein $j \geq i$ mit $\neg P$ zum Zeitpunkt j . Andererseits folgt aus Zeile 2, dass im Zustand j P gelten muss. Also haben wir einen Widerspruch, d.h. aber dass die Annahme, dass immer P gilt, nicht zutreffen kann.

Im folgenden Abschnitt werden die Regeln für das natürliche Schließen in der LTL im Detail beschrieben.

20.2 Regeln für das natürliche Schließen in der LTL

Es finden sich in der Literatur verschiedene Sätze von Regeln für das natürliche Schließen in der LTL, u.a. in [Sim94, Mar02, BBGS06, BGS07b, BGS07a]. Die im Folgenden beschriebenen Regeln entsprechen (weitgehend) den Regeln in der Dissertation von Davide Marchignoli [Mar02].

20.2.1 Regeln für Boolesche Junktoren

Die Regeln für die *Konjunktion*:

	<i>Einführung</i>	<i>Elimination</i>
\wedge	$\frac{i : \varphi \quad i : \psi}{i : \varphi \wedge \psi} \text{ ai}$	$\frac{i : \varphi \wedge \psi}{i : \varphi} \text{ } \wedge e_1 \quad \frac{i : \varphi \wedge \psi}{i : \psi} \text{ } \wedge e_2$

Die Regeln für die *Disjunktion*:

	<i>Einführung</i>	<i>Elimination</i>
\vee	$\frac{i : \varphi}{i : \varphi \vee \psi} \text{ } \vee i_1 \quad \frac{i : \psi}{i : \varphi \vee \psi} \text{ } \vee i_2$	$\frac{i : \varphi \vee \psi \quad \begin{array}{ c c } \hline i : \varphi & i : \psi \\ \vdots & \vdots \\ j : \chi & j : \chi \\ \hline \end{array}}{j : \chi} \text{ } \vee e$

Die Regeln für die *Implikation*:

	<i>Einführung</i>	<i>Elimination</i>
\rightarrow	$\frac{\begin{array}{ c } \hline i : \varphi \\ \vdots \\ i : \psi \\ \hline \end{array}}{i : \varphi \rightarrow \psi} \rightarrow i$	$\frac{i : \varphi \quad i : \varphi \rightarrow \psi}{i : \psi} \rightarrow e, MP$

Die Regeln für die *Negation*:

	<i>Einführung</i>	<i>Elimination</i>
\neg	$\frac{\begin{array}{ c } \hline i : \varphi \\ \vdots \\ j : \perp \\ \hline \end{array}}{i : \neg \varphi} \neg i$	$\frac{i : \varphi \quad i : \neg \varphi}{j : \psi} \neg e$

Schließlich die Regeln für den *Widerspruchsbeweis* und EFQ:

	<i>Einführung</i>	<i>Elimination</i>
RAA, \perp	$\boxed{\begin{array}{c} i : \neg\varphi \\ \vdots \\ j : \perp \end{array}}$ RAA	$\frac{j : \perp}{i : \varphi}$ $\perp e$, EFQ

Die Regeln ergeben sich auf natürliche Weise aus den Regeln des natürlichen Schließens für die Aussagenlogik und ein Blick auf die Semantik der Operatoren zeigt, dass die Regel wahrheitserhaltend sind.

Zu bemerken ist höchstens, dass in den Regeln, in denen die Negation vorkommt, nicht nur der Zustand i eine Rolle spielt. Am Beispiel der Einführung der Negation etwa besagt die Regel: Wenn es gelingt aus $i : \varphi$ in irgendeinem Zustand j den Widerspruch herzuleiten, dann muss $i : \neg\varphi$ gelten.

20.2.2 Regeln für relationale Aussagen

Temporale Aussagen können sich auf unterschiedliche Zeitpunkte, also unterschiedliche Marken der Formeln beziehen. Also brauchen wir Regeln, mit denen wir Aussagen über die Beziehung von Zeitpunkten, *relationale Aussagen* herleiten können.

Regeln für die *Gleichheit*:

$=$	$\frac{}{i = i}$ =refl	$\frac{i = j}{j = i}$ =sym
	$\frac{i = j \quad j = k}{i = k}$ =trans	$\frac{i = j \quad i : \varphi}{j : \varphi}$ =fml
	$\frac{i = j}{i \leq j}$ =/<=	

Regeln für die *Nachfolger-Relation*:

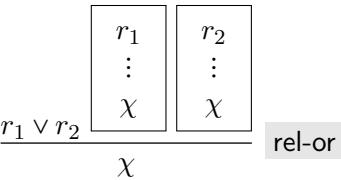
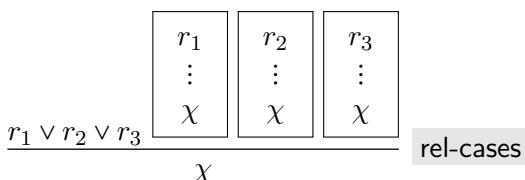
succ	$\frac{}{succ(i, i')}$ succ	$\frac{succ(i, i') \quad succ(i, i'')}{i' = i''}$ succ-fn
	$\frac{succ(i, i')}{i \leq i'}$ succ/ \leq	$\frac{succ(i, i') \quad i \leq j}{i = j \vee i' \leq j}$ succ/ \leq -linear
	$\frac{i \leq j \quad succ(i, i') \quad succ(j, j')}{i' \leq j'}$ \leq =succ ² / \leq	
	$\frac{succ(i, i') \quad succ(j, j') \quad i' \leq j'}{i \leq j}$ succ ² / \leq / \leq	
	$\frac{succ(i, i') \quad succ(j, j') \quad i = j}{i' = j'}$ =succ ² / $=$	

Regeln für die Relation \leq :

≤	$\frac{}{i \leq i}$ \leq =refl	$\frac{}{i \leq j}$ \leq =serial
	$\frac{i \leq j \quad j \leq k}{i \leq k}$ \leq =trans	$\frac{i \leq j \quad i \leq k}{j \leq k \vee j = k \vee k \leq j}$ \leq =linear

Die Regel \leq =serial bedeutet, dass es immer möglich ist, zu einem Zeitpunkt i einen späteren Zeitpunkt j zu finden.

Regeln für die Auflösung des \vee in relationalen Ausdrücken:

∨	$\frac{r_1 \vee r_2}{\chi}$ 	rel-or
	$\frac{r_1 \vee r_2 \vee r_3}{\chi}$ 	rel-cases

In diesen Regeln bezeichnen die r_i relationale Ausdrücke und χ eine

markierte Formel.

Dass die Regeln für relationale Ausdrücke wahrheitserhaltend sind, ergibt sich aus den Eigenschaften der korrespondierenden Relationen auf \mathbb{N} . Die Regeln für die Auflösung des \vee sind offensichtlich korrekt.

20.2.3 Regeln für temporale Junktoren

Die Regeln für \square :

	<i>Einführung</i>	<i>Elimination</i>
\square	$\frac{i : \varphi \quad \boxed{\begin{array}{c} i \leq j \\ \vdots \\ j : \varphi \end{array}}}{i : \square \varphi} \square i$	$\frac{i : \square \varphi \quad i \leq j}{j : \varphi} \square e$
ind	$\frac{i : \varphi \quad \boxed{\begin{array}{c} i \leq j \\ succ(j, j') \\ j : \varphi \\ \vdots \\ j' : \varphi \end{array}}}{i : \square \varphi} \text{ind}$	

Bei der Einführung von \square muss j ein beliebiger Zeitpunkt sein, er darf also nicht bereits an anderer Stelle vorher in der Herleitung verwendet werden. Diese Mimik entspricht der Einführung des Allquantors in der Prädikatenlogik.

Die zweite Möglichkeit der Einführung von \square ist die *Induktion*: Wenn man – vorausgesetzt, dass φ im Zustand i gilt – für ein beliebiges $j \geq i$ zeigen kann, dass wenn φ für j gilt, dies auch für den Nachfolger j' der Fall ist, dann hat man gezeigt, dass $i : \square \varphi$ gilt.

Die Regeln für \diamond :

	<i>Einführung</i>	<i>Elimination</i>
\diamond	$\frac{j : \varphi \quad i \leq j}{i : \diamond \varphi} \diamond i$	$\frac{i : \diamond \varphi \quad \boxed{\begin{array}{c} i \leq j \\ j : \varphi \\ \vdots \\ \chi \end{array}}}{\chi} \diamond e$

Bei der Einführung von \Diamond muss man ein j gegeben haben, an dem die Formel gilt, dann darf man schlussfolgern, dass im früheren Zeitpunkt i $\Diamond \varphi$ zutrifft.

Bei der Auflösung von \Diamond können wir von der Existenz eines Zeitpunkts j ausgehen, an dem φ gilt. Wenn es möglich ist unter dieser Annahme die markierte Formel χ herzuleiten, dann darf man sie als hergeleitet verwenden.

Die Regeln für \circ :

	<i>Einführung</i>	<i>Elimination</i>
\circ	$\frac{i' : \varphi \quad succ(i, i')}{i : \circ \varphi} \circ i$	$\frac{\begin{array}{c} succ(i, i') \\ i' : \varphi \\ \vdots \\ \chi \end{array}}{i : \circ \varphi} \circ e$
	$\boxed{\begin{array}{c} succ(i, i') \\ \vdots \\ i' : \varphi \end{array}} \circ i'$	$\frac{i : \circ \varphi \quad succ(i, i')}{i' : \varphi} \circ e'$

Für die Einführung von \circ haben wir zwei Varianten der Regel: $\circ i$ kann man verwenden, wenn die herzuleitende Aussage im Nachfolgezustand von i bereits hergeleitet ist. Die zweite Variante gibt uns die Beweisverpflichtung, im Zustand i' die Aussage φ herzuleiten.

Analog gibt es für die Auflösung von \circ zwei Varianten an Regeln. Bei $\circ e$ bezeichnet χ eine markierte Formel, die wir nach dem Erbringen der Beweisverpflichtung an die Stelle von $i : \circ \varphi$ schreiben können.

Die Regeln für \mathcal{U} :

	<i>Einführung</i>	<i>Elimination</i>
\mathcal{U}	$\frac{i : \psi}{i : \varphi \mathcal{U} \psi} \quad \mathcal{U} i_1$	$\frac{\begin{array}{c} i \leq j \\ j : \psi \\ \vdots \\ j : \chi \end{array}}{i : \chi} \quad \mathcal{U} e$
	$\frac{i : \varphi \quad i : \circ (\varphi \mathcal{U} \psi)}{i : \varphi \mathcal{U} \psi} \quad \mathcal{U} i_2$	$\frac{\begin{array}{c} i \leq j \\ j : \varphi \wedge \circ \chi \\ \vdots \\ j : \chi \end{array}}{i : \chi} \quad \mathcal{U} e$

Die beiden Regeln für die Einführung von \mathcal{U} ergeben sich aus der Expansion von \mathcal{U} :

$$\varphi \mathcal{U} \psi \equiv \psi \vee (\varphi \wedge \circ (\varphi \mathcal{U} \psi))$$

Und die Korrektheit der Regel $\mathcal{U} e$ ergibt sich aus folgender Überlegung:

Fall 1: Es gilt $i : \psi$. Wenn man nun für einen beliebigen Zeitpunkt $j \geq i$ zeigen kann, dass χ unter der Voraussetzung ψ gilt, dann folgt in diesem Fall $i : \chi$ aus $i : \varphi \mathcal{U} \psi$.

Fall 2: Es sei nun für alle Zeitpunkte zwischen i und $k - 1$ φ zutreffend und im Zustand $j = k$ gelte ψ . Da wir im Fall 1 schon für beliebiges j gezeigt haben, dass dann auch χ gilt, gilt im Zustand k χ . Also haben wir zum Zeitpunkt $k - 1$, dass φ gilt und dass $\circ \chi$ gilt. Können wir nun zeigen, dass immer in dieser Situation χ gilt, dann gilt diese Argumentation auch für alle weiteren Zeitpunkte $k - 2, k - 3, \dots$. Also folgt auch in diesem Fall $i : \chi$ aus $i : \varphi \mathcal{U} \psi$ (siehe [BGS07b]).

20.3 Beispiele für das natürliche Schließen in der LTL

Die ersten fünf Beispiele zeigen, dass man Herleitung aus der Aussagenlogik einfach in die LTL übernehmen kann.

Beispiel 20.2 (Tertium non datur TND).

1.	$i : \neg(\varphi \vee \neg\varphi)$	angenommen
2.	$i : \varphi$	angenommen
3.	$i : \varphi \vee \neg\varphi$	$\vee i_1 2$
4.	$i : \perp$	$\neg e 1, 3$
5.	$i : \neg\varphi$	$\neg i 2-4$
6.	$i : \varphi \vee \neg\varphi$	$\vee i_2 5$
7.	$i : \perp$	$\neg e 1, 6$
8.	$i : \varphi \vee \neg\varphi$	raa 1-7

Beispiel 20.3 (Modus tollens MT).

1.	$i : \varphi \rightarrow \psi$	gegeben
2.	$i : \neg\psi$	gegeben
3.	$i : \varphi$	angenommen
4.	$i : \psi)$	$\rightarrow e 1, 3$
5.	$i : \perp$	$\neg e 2, 4$
6.	$i : \neg\varphi$	$\neg i 3-5$

Beispiel 20.4 (Axiom 1 des Hilbert-Kalküls).

$$\vdash_{\pi} \varphi \rightarrow (\psi \rightarrow \varphi)$$

1.	$i : \varphi$	angenommen
2.	$i : \psi$	angenommen
3.	$i : \varphi$	übernommen 1
4.	$i : \psi \rightarrow \varphi$	$\rightarrow i 2-3$
5.	$i : \varphi \rightarrow (\psi \rightarrow \varphi))$	$\rightarrow i 1-4$

Beispiel 20.5 (Axiom 2 des Hilbert-Kalküls).

$$\vdash_{\pi} (\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$$

1.	$i : (\varphi \rightarrow (\psi \rightarrow \chi))$	angenommen
2.	$i : \varphi \rightarrow \psi$	angenommen
3.	$i : \varphi$	angenommen
4.	$i : \psi \rightarrow \chi$	$\rightarrow e 1, 3$
5.	$i : \psi \vee \neg\psi$	TND
6.	$i : \psi$	angenommen
7.	$i : \chi$	$\rightarrow e 4, 6$
8.	$i : \neg\psi$	angenommen
9.	$i : \neg\varphi$	MT 2, 8
10.	$i : \perp$	$\neg e 9, 3$
11.	$i : \chi$	EFQ 10
12.	$i : \chi$	$\vee e 5, 6-7, 8-11$
13.	$i : \varphi \rightarrow \chi$	$\rightarrow i 3-12$
14.	$i : (\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)$	$\rightarrow i 2-13$
15.	$i : (\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$	$\rightarrow i 1-14$

Beispiel 20.6 (Axiom 3 des Hilbert-Kalküls).

$$\vdash_{\pi} (\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi)$$

1.	$i : \neg\varphi \rightarrow \neg\psi$	angenommen
2.	$i : \psi$	angenommen
3.	$i : \varphi \vee \neg\varphi$	TND
4.	$i : \varphi$	angenommen
5.	$i : \varphi$	übernommen 4
6.	$i : \neg\varphi$	angenommen
7.	$i : \neg\psi$	$\rightarrow e 1, 6$
8.	$i : \perp$	$\neg e 7, 2$
9.	$i : \varphi$	EFQ 8
10.	$i : \varphi$	$\vee e 3, 4-5, 6-9$
11.	$i : \psi \rightarrow \varphi$	$\rightarrow i 2-10$
12.	$i : (\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi)$	$\rightarrow i 1-11$

Beispiel 20.7 ($\square \rightarrow \text{current}$).

$$\square \varphi \vdash_{\pi} \varphi$$

1. $i : \square \varphi$ gegeben
2. $i : i \leq i$ $\leq=\text{refl}$
3. $i : \varphi$ $\square e 1, 2$

Beispiel 20.8 ($\square \rightarrow \circ$).

$$\square \varphi \vdash_{\pi} \circ \varphi$$

1. $i : \square \varphi$ gegeben
2. $i : succ(i, i')$ succ
3. $i : i \leq i'$ $\text{succ}/\leq = 2$
4. $i' : \varphi$ $\square e 1, 3$
5. $i : \circ \varphi$ $\circ i 4, 2$

Beispiel 20.9 ($\circ \neg \rightarrow \neg \circ$).

$$\circ \neg \varphi \vdash_{\pi} \neg \circ \varphi$$

1. $i : \circ \neg \varphi$ gegeben
2. $i : \circ \varphi$ angenommen
3. $succ(i, i')$ angenommen
4. $i' : \varphi$ angenommen
5. $i' : \neg \varphi$ $\circ e' 1, 3$
6. $i' : \perp$ $\neg e 5, 4$
7. $i' : \perp$ $\circ e 2, 3-6$
8. $i : \neg \circ \varphi$ $\neg i 2-7$

Beispiel 20.10 ($\neg \circ \rightarrow \circ \neg$).

$$\neg \circ \varphi \vdash_{\pi} \circ \neg \varphi$$

1. $i : \neg \circ \varphi$ gegeben
2. $succ(i, i')$ succ
3. $i' : \varphi$ angenommen
4. $i : \circ \varphi$ $\circ i 3, 2$
5. $i' : \perp$ $\neg e 1, 4$
6. $i' : \neg \varphi$ $\neg i 3-5$
7. $i : \circ \neg \varphi$ $\circ i 6, 2$

20.4 Die Vollständigkeit des natürlichen Schließens in der LTL bezüglich der Pfad-Semantik

Die Korrektheit der natürlichen Schließens in der LTL, also dass $\Gamma \vdash_{\pi} \varphi \Rightarrow \Gamma \vDash_{\pi} \varphi$ gilt, ergibt sich aus der Korrektheit der Regeln.

Was die Vollständigkeit des natürlichen Schließens in der LTL angeht, also die Frage ob $\Gamma \vDash_{\pi} \varphi \Rightarrow \Gamma \vdash_{\pi} \varphi$ gilt, werden wir dies in diesem Abschnitt dadurch zeigen, dass wir die Frage zurückführen auf das Beweissystem \mathcal{L} in [BA12, Chap. 14], das vollständig ist.

Dabei ist zu beachten, dass sich die Semantik auf die Pfad-Semantik in 19.3 bezieht, für die wir \vDash_{π} als Symbol verwenden. Also bedeutet $\Gamma \vDash_{\pi} \varphi$, dass φ auf allen Pfaden gilt, auf denen auch die Formeln in Γ gelten.

20.4.1 Das Beweissystem \mathcal{L}

Die Operatoren im Beweissystem \mathcal{L} für die lineare temporale Logik sind die der Aussagenlogik sowie die temporalen Operatoren \Box, \circ, \Diamond sowie \mathcal{U} .

\mathcal{L} hat die folgenden Axiome:

Definition 20.4. Die Axiome von \mathcal{L} sind:

Axiom 0.1	Prop 1	$(\varphi \rightarrow (\psi \rightarrow \varphi))$
Axiom 0.2	Prop 2	$(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$
Axiom 0.3	Prop 3	$(\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi)$
Axiom 1	Distributivgesetz für \Box	$\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$
Axiom 2	Distributivgesetz für \circ	$\circ(\varphi \rightarrow \psi) \rightarrow (\circ\varphi \rightarrow \circ\psi)$
Axiom 3	Expansion von \Box	$\Box\varphi \rightarrow (\varphi \wedge \circ\varphi \wedge \circ \Box \varphi)$
Axiom 4	Induktion	$\Box(\varphi \rightarrow \circ\varphi) \rightarrow (\varphi \rightarrow \Box\varphi)$
Axiom 5	Linearität	$\circ\varphi \leftrightarrow \neg\circ\neg\varphi$
Axiom 6	Expansion von \mathcal{U}	$\varphi \mathcal{U} \psi \leftrightarrow (\psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi)))$
Axiom 7	Vorkommnis	$\varphi \mathcal{U} \psi \rightarrow \Diamond \psi$

Die Schlussregeln von \mathcal{L} sind:

$$\begin{array}{ll} \text{Modus Ponens} & \frac{\vdash \varphi \quad \vdash \varphi \rightarrow \psi}{\vdash \psi} \\ \text{Verallgemeinerung} & \frac{\vdash \varphi}{\vdash \Box \varphi} \end{array}$$

Ben-Ari zeigt die Vollständigkeit von \mathcal{L} mit den Axiomen 0 - 5 in [BA12, Chap. 14] durch die Konstruktion eines Beweises für eine allgemeingültige

Formel der LTL. Der ursprüngliche Beweis, der auch die Axiome mit \mathcal{U} einschließt ist in [GPSS80] zu finden.

20.4.2 Vollständigkeit des natürlichen Schließens in der LTL

Zunächst beobachtet man, dass die Schlussregeln von \mathcal{L} für das natürliche Schließen übertragbar sind:

Lemma 20.1. $\vdash_\pi \varphi \Rightarrow \vdash_\pi \Box \varphi$

Beweis. Um zu zeigen, dass $\Box \varphi$ gilt, verwenden wir die Regel für die Einführung von \Box , d.h. wir müssen einen Beweis für φ zu einem beliebigen Zeitpunkt j finden. Da $\vdash_\pi \varphi$ gilt, gibt es einen solchen Beweis, den wir einfach einsetzen können. \square

Lemma 20.2. Gilt $\vdash_\pi \varphi$ sowie $\vdash_\pi \varphi \rightarrow \psi$, dann gibt es auch eine Herleitung $\vdash_\pi \psi$.

Beweis. Um die Herleitung für ψ zu konstruieren, beginnen wir mit der Herleitung von φ . Die letzte Zeile dieser Herleitung ist dann $i : \varphi$. Nach Voraussetzung gibt es eine Herleitung für $\varphi \rightarrow \psi$, für die wir durch Umbenennung der Marken sicherstellen können, dass keine Marke aus der Herleitung von φ vorkommt. Dies tun wir um garantieren zu können, dass nicht Marken verwendet werden, die im Beweis von φ einen fixen Zeitpunkt markieren. Die letzte Marke im Beweis von $\varphi \rightarrow \psi$ kann aber keine solche sein, weil ja $\varphi \rightarrow \psi$ keine Voraussetzung hat. Also können wir an dieser Stelle und entsprechenden Stellen die Marke i verwenden. Wir ergänzen unsere Herleitung also durch den Beweis von $\varphi \rightarrow \psi$. Die vorherige Zeile war $i : \varphi$ und die letzte Zeile ist dann $i : \varphi \rightarrow \psi$, was uns die Elimination der Implikation erlaubt und $i : \psi$ ergibt. \square

Um die Vollständigkeit zu zeigen, müssen wir also jetzt Herleitungen für alle Axiome finden:

Lemma 20.3 (Axiome 0.x).

$$\begin{aligned} &\vdash_\pi (\varphi \rightarrow (\psi \rightarrow \varphi)) \\ &\vdash_\pi (\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)) \\ &\vdash_\pi (\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi) \end{aligned}$$

Beweis. Diese drei Axiome sind die Axiome des Hilbert-Kalküls für die Aussagenlogik. Sie zusammen mit dem Modus Ponens sind ein vollständiges Beweissystem für die Aussagenlogik [BA12, Chap. 3]. In den Beispielen 20.4 bis 20.6 wurden Herleitungen im Beweissystems des natürlichen

Schließens für diese Axiome konstruiert. Dass es für sie Herleitungen gibt, folgt natürlich auch aus der Vollständigkeit des natürlichen Schließens für die Aussagenlogik. \square

In den folgenden Herleitungen lassen wir den einleitenden Schritt der Einführung der Implikation weg und nehmen der Einfachheit halber die Antezedenz als gegeben.

Lemma 20.4 (Axiom 1).

$$\vdash_{\pi} \Box(\varphi \rightarrow \psi) \rightarrow (\Box \varphi \rightarrow \Box \psi)$$

Beweis.

1.	$i : \Box(\varphi \rightarrow \psi)$	gegeben
2.	$i : \Box \varphi$	angenommen
3.	$i \leq j$	angenommen
4.	$j : \varphi \rightarrow \psi$	$\Box e 1, 3$
5.	$j : \varphi$	$\Box e 2, 3$
6.	$j : \psi$	$\rightarrow e 4, 5$
7.	$i : \Box \psi$	$\Box i 3-6$
8.	$i : \Box \varphi \rightarrow \Box \psi$	$\rightarrow i 2-7$

\square

Lemma 20.5 (Axiom 2).

$$\vdash_{\pi} \circ(\varphi \rightarrow \psi) \rightarrow (\circ \varphi \rightarrow \circ \psi)$$

Beweis.

1.	$i : \circ(\varphi \rightarrow \psi)$	gegeben
2.	$i : \circ \varphi$	angenommen
3.	$succ(i, i')$	succ
4.	$i' : \varphi \rightarrow \psi$	$\circ e' 1, 3$
5.	$i' : \varphi$	$\circ e' 2, 3$
6.	$i' : \psi$	$\rightarrow e 4, 5$
7.	$i : \circ \psi$	$\circ i 6, 3$
8.	$i : \circ \varphi \rightarrow \circ \psi$	$\rightarrow i 2-8$

\square

Lemma 20.6 (Axiom 3).

$$\vdash_{\pi} \square \varphi \rightarrow (\varphi \wedge \circ \varphi \wedge \circ \square \varphi)$$

Beweis.

1.	$i : \square \varphi$	gegeben
2.	$i : \varphi$	$\square \rightarrow \text{current}$ (s. 20.7) 1
3.	$i : \circ \varphi$	$\square \rightarrow \circ$ (s. 20.8) 1
4.	$\text{succ}(i, i')$	succ
5.	$i' \leq j$	angenommen
6.	$i \leq i'$	$\text{succ}/\leq = 4$
7.	$i \leq j$	$\leq=\text{trans}$ 6, 5
8.	$j : \varphi$	$\square \in 1, 7$
9.	$i' : \square \varphi$	$\square i$ 5-8
10.	$i : \circ \square \varphi$	$\circ i$ 9, 4
11.	$i : \circ \varphi \wedge \circ \square \varphi$	$\wedge i$ 3, 10
12.	$i : \varphi \wedge \circ \varphi \wedge \circ \square \varphi$	$\wedge i$ 2, 11

□

Lemma 20.7 (Axiom 4).

$$\vdash_{\pi} \square(\varphi \rightarrow \circ \varphi) \rightarrow (\varphi \rightarrow \square \varphi)$$

Beweis.

1.	$i : \square(\varphi \rightarrow \psi)$	gegeben
2.	$i : \varphi$	angenommen
3.	$i : \varphi \rightarrow \psi$	$\square \rightarrow \text{current}$ 1
4.	$i : \circ \varphi$	$\rightarrow e$ 3, 2
5.	$i \leq j$	angenommen
6.	$\text{succ}(j, j')$	angenommen
7.	$j : \varphi$	angenommen
8.	$j : \varphi \rightarrow \circ \varphi$	$\square e$ 1, 5
9.	$j : \circ \varphi$	$\rightarrow e$ 8, 7
10.	$j' \varphi$	atnext - e' 9, 6
11.	$i : \square \varphi$	induction 2, 5-10
12.	$i : \varphi \rightarrow \square \varphi$	$\rightarrow i$ 2-11

□

Lemma 20.8 (Axiom 5).

$$\vdash_{\pi} \circ \varphi \rightarrow \neg \circ \neg \varphi$$

$$\vdash_{\pi} \neg \circ \neg \varphi \rightarrow \circ \varphi$$

Beweis.

1. $i : \circ \varphi$ gegeben
2. $i : \circ \neg \varphi$ angenommen
3. $i : \neg \circ \varphi$ $\circ \neg \rightarrow \neg \circ$ (s. 20.9) 2
4. $i : \perp$ $\neg e$ 3, 1
5. $i : \neg \circ \neg \varphi$ $\neg i$ 2-4

und mit der Verwendung der Elimination von $\neg\neg$, die man genau wie in der Aussagenlogik als abgeleitete Regel verwenden kann:

1. $i : \neg \circ \neg \varphi$ gegeben
2. $i : \circ \neg\neg \varphi$ $\neg \circ \rightarrow \circ \neg$ (s. 20.10) 1
3. $succ(i, i')$ succ
4. $i' : \neg\neg \varphi$ $\circ e'$ 2, 3
5. $i' : \varphi$ $\neg\neg e$ 4
6. $i : \circ \varphi$ $\circ i$ 5 3

□

Lemma 20.9 (Axiom 6).

$$\vdash_{\pi} (\psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi))) \rightarrow \varphi \mathcal{U} \psi$$

$$\vdash_{\pi} \varphi \mathcal{U} \psi \rightarrow (\psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi)))$$

Beweis.

1. $i : \psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi))$ gegeben
2. $i : \varphi$ angenommen
3. $i : \varphi \mathcal{U} \psi$ $\mathcal{U} i_1$ 2
4. $i : \varphi \wedge \circ(\varphi \mathcal{U} \psi)$ angenommen
5. $i : \varphi$ $\wedge e_1$ 4
6. $i : \circ(\varphi \mathcal{U} \psi)$ $\wedge e_2$ 4
7. $i : \varphi \mathcal{U} \psi$ $\mathcal{U} i_2$ 5, 6
8. $i : \varphi \mathcal{U} \psi$ $\vee e$ 1, 2-3, 4-7

Interessanterweise verwenden wir diese Richtung des Axioms, um die andere Richtung zu zeigen:

1.	$i : \varphi \mathcal{U} \psi$	gegeben
2.	$i \leq j$	angenommen
3.	$j : \psi$	angenommen
4.	$j : \psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi))$	$\vee i_1 3$
5.	$i \leq j$	angenommen
6.	$j : \varphi \wedge \circ(\psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi)))$	angenommen
7.	$j : \circ(\psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi)))$	$\wedge e_2 6$
8.	$j : \varphi$	$\wedge e_1 6$
9.	$\text{succ}(j, j')$	succ
10.	$j' : \psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi))$	$\circ e' 7, 9$
11.	$j' : \varphi \mathcal{U} \psi$	$\mathcal{U} \text{ expansion}_1 10$
12.	$j : \circ(\varphi \mathcal{U} \psi)$	$\circ i 11, 9$
13.	$j : \varphi \wedge \circ(\varphi \mathcal{U} \psi)$	$\wedge i 8, 12$
14.	$j : \psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi))$	$\vee i_2 13$
15.	$i : \psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi))$	$\mathcal{U} e 1, 2-4, 5-14$

□

Lemma 20.10 (Axiom 7).

$$\vdash_{\pi} \varphi \mathcal{U} \psi \rightarrow \diamond \psi$$

Beweis. Siehe folgende Seite. □

Satz 20.1. Das Beweissystem des natürlichen Schließens in der linearen temporalen Logik ist vollständig, d.h.

$$\Gamma \vDash_{\pi} \varphi \Rightarrow \Gamma \vdash_{\pi} \varphi.$$

Beweis. Die Lemmata 20.3 - 20.10 zeigen, dass alle Axiome der LTL mit den Regeln des natürlichen Schließens hergeleitet werden können. Darüberhinaus können nach 20.1 und 20.2 aus den Schlussregeln von \mathcal{L} leicht Herleitungen des natürlichen Schließens konstruiert werden. Damit ergibt sich die Vollständigkeit des natürlichen Schließens aus der von \mathcal{L} . □

1.	$i : \varphi \mathcal{U} \psi$	gegeben
2.	$i \leq j$	angenommen
3.	$j : \psi$	angenommen
4.	$j \leq j$	$\leq \text{refl}$
5.	$j : \Diamond \psi$	$\Diamond i 3, 4$
6.	$i \leq j$	angenommen
7.	$j : \varphi \wedge \Diamond \psi$	angenommen
8.	$j : \Diamond \psi$	$\wedge e_2 7$
9.	$\text{succ}(j, j')$	succ
10.	$j' : \Diamond \psi$	$\Diamond e' 8, 9$
11.	$j' \leq k$	angenommen
12.	$k : \psi$	angenommen
13.	$j \leq j'$	≤ 9
14.	$j \leq k$	$\leq \text{trans } 13, 11$
15.	$j : \Diamond B$	$\Diamond i 12, 14$
16.	$j : \Diamond \psi$	$\Diamond e 10, 11-15$
17.	$i : \Diamond \psi$	$\mathcal{U} e 1, 2-5, 6-16$

Bemerkung. Bolotov et al. beschreiben in [BGS07a] einen Algorithmus, mit dem man eine Herleitung $\vdash_\pi \varphi$ für eine allgemeingültige Formel der LTL konstruieren kann.

Kapitel 21

Anwendungen der LTL in der Softwaretechnik

In diesem Kapitel wollen wir zwei Anwendungen der linearen temporalen Logik in der Softwaretechnik betrachten: das *Model Checking*, eine Technik zur Verifikation von Software-Artefakten, sowie das *Zielmodell* in der Softwareanforderungsanalyse, im *Requirements Engineering*.

21.1 Model Checking

21.1.1 Konzept des Model Checkings mit SPIN

Das grundlegende Konzept des Model Checkings haben wir bereits beim Thema der Evaluierung von Formeln der linearen temporalen Logik in einer Kripke-Struktur in Abschnitt 19.6 kennengelernt.

Diesem Konzept folgt der Model Checker SPIN, der von Gerard J. Holzmann¹ in den 1980er Jahren bei den Bell Labs entwickelt wurde. 2001 erhielt Holzmann für seine Entwicklung den ACM Software System Award.

SPIN wird in folgender Weise eingesetzt, siehe [Hol04]. Die Nummerierung der Schritte bezieht sich auf Abbildung 21.1.

- ① Das zu verifizierende Software-Artefakt kann ein Protokoll, ein Algorithmus, ein Konzept für einen verteilten Algorithmus oder Ähnliches sein. Um Model Checking einsetzen zu können, muss es in der Sprache PROMELA für SPIN formuliert werden. PROMELA steht für *Process* oder auch *Protocol Meta Language*. In dieser Sprache werden Modelle für SPIN geschrieben, was auch in den ursprüng-

¹ Gerard Holzmann, niederländisch-amerikanischer Informatiker.

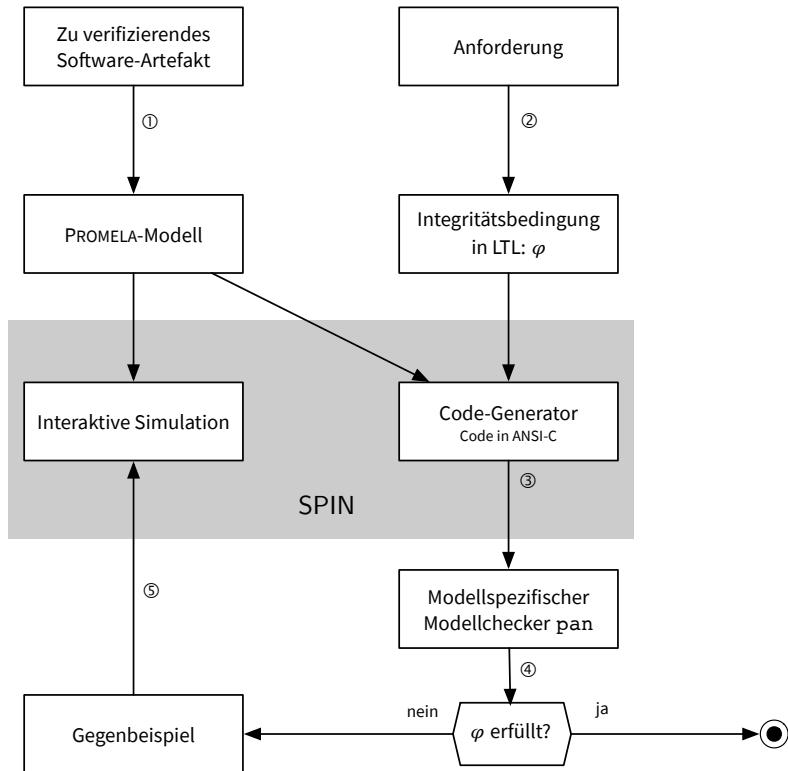


Abbildung 21.1: Model Checking mit SPIN

lichen Namen von SPIN, nämlich *Simple PROMELA Interpreter*, eingegangen ist.

Die Spezifikation des Modells in PROMELA muss natürlich exakt dem Artefakt entsprechen, welches man verifizieren möchte.

Es ist möglich, in SPIN interaktive Simulationen des PROMELA-Modells durchzuführen. Diese Funktionalität von SPIN kann helfen, das Modell korrekt, d.h. dem zu verifizierenden Artefakt entsprechend zu formulieren.

- ② Die Anforderung, die beschreibt, was das Software-Artefakt erfüllen soll, muss als Integritätsbedingung in LTL ausgedrückt werden. Innerhalb einer Spezifikation können mehrere Formeln angegeben werden. In der Regel gibt man Korrektheitsbedingungen als LTL-Formeln an, die dann im weiteren Verlauf auf dem gegebenen Modell evaluiert werden wie in 19.6 dargestellt.
- ③ SPIN ist im Grunde ein Generator für Modellprüfer: Das Modell und die zu verifizierenden Formeln in LTL werden von SPINS Code-

Generator zu Quellcode in ANSI-C für den modellspezifischen Model Checker übersetzt.

Dieser Quellcode muss dann von einem C-Compiler übersetzt werden. Das Ergebnis ist ein C-Programm namens `pan` (für *Protocol Specification Analyzer*), das dann die eigentliche Überprüfung durchführt.

Wenn die zu prüfende Integritätsbedingung nicht erfüllt ist, erzeugt `pan` ein Gegenbeispiel, d.h. einen Ablauf im Modell, der zu einer Situation führt, in der die Integritätsbedingung verletzt ist.

- ④ Diesen *trail*, der das Zustandekommen der Verletzung aufzeigt, kann man im Simulator von SPIN nachvollziehen, um die Ursache des Fehlers verstehen zu können.

In den folgenden beiden Abschnitt wollen wir zwei Beispiele für den Einsatz von SPIN betrachten und dabei gleichzeitig dafür nötigen Konstrukte von PROMELA und das Vorgehen mit SPIN erläutern.

21.1.2 Eine Überraschung für Mordechai Ben-Ari

In seinem Buch „Principles of the Spin Model Checker“ [BA08] beschreibt Mordechai Ben-Ari ein simples Szenario zweier nebenläufiger Programme, die eine gemeinsame Ressource verwenden. Es scheint so zu sein, dass man leicht versteht, was bei der Ausführung alles passieren kann. Oder doch nicht?

21.1.2.1 Einfaches Beispiel von Wechselwirkung zwischen Prozessen

Das Szenario muss für SPIN in PROMELA definiert werden. Das folgende Listing `111.pml` zeigt ein einfaches solches Programm:

```

1 #include "for.h"
2 byte n = 0;
3
4 proctype P() {
5     byte temp;
6     for (i, 1, 10)
7         temp = n + 1;
8         n = temp
9     rof(i)
10 }
11
12 init {
13     atomic {
14         run P();
15         run P();
16     }

```

```

17     (_nr_pr == 1) ->
18         printf("The value is %d\n", n);
19 }
```

PROMELA hat numerische Datentypen, von denen `byte` einer ist. In Zeile 2 wird ein Byte `n` mit 0 initialisiert.

Programme in PROMELA bestehen aus *Prozessen*, die als `proctype` definiert werden. In den Zeilen 4 - 10 wird ein Prozess `P` definiert, der eine lokale Variable `temp` verwendet. In der Include-Datei `for.h` hat Ben-Ari ein Makro definiert, das eine `for`-Schleife simuliert. PROMELA selbst hat nur `do ... od` um Schleifen zu spezifizieren, `for.h` verwendet `do`. In der `for`-Schleife liest `P` die globale Variable `n` und schreibt ihr Inkrement in die lokale Variable `temp`. Danach wird der globalen Variablen `n` der inkrementierte Wert in `temp` zugewiesen.²

Eine spezielle Rolle spielt der Prozess `init`, der in den Zeilen 12 - 19 definiert wird. Hat ein Programm einen solchen Prozess, dann wird er stets zuerst aktiviert und bekommt die Prozess-Id `_pid 0`.

In unserem Beispiel startet `init` zwei Instanzen des Prozesses `P`. `atomic` sorgt dafür, dass beide Prozesse zunächst erzeugt werden und nicht der erste schon startet, ehe der zweite erzeugt ist.

In Zeile 17 wird die eingebaute Variable `_nr_pr` überprüft. Sie enthält stets die aktuelle Zahl der Prozesse. Ein Ausdruck wie `(_nr_pr == 1)`, der zu einem Wahrheitswert auswertet, ist in PROMELA genau dann *ausführbar*, wenn er `true` ist. Das bedeutet, dass in Zeile 17 der Prozess `init` solange blockiert, bis die Bedingung wahr wird, d.h. die Zahl der Prozesse 1 wird, die beiden gestarteten Prozesse `P` also beendet sind. Tritt das ein, wird der Wert der globalen Variablen `n` ausgegeben.

Es ist recht einfach, einen Ablauf des Programms zu machen:

```
spin ill.pml
```

Mit diesem Kommando erzeugt SPIN *einen* möglichen Ablauf, das Programm wird *simuliert*.

Die Ausgabe eines solchen Ablaufs:

² Es gibt in PROMELA auch den Inkrement-Operator `++`. In unserem Beispiel wollen wir ihn jedoch nicht verwenden, sondern davon ausgehen, dass wir eine Maschine modellieren, in der das Inkrement dadurch gebildet wird, dass eine Wert aus einer Speicherzelle nur dadurch verändert werden kann, dass er in ein Register geladen wird und dann nach der Änderung in die Speicherzelle zurückgeschrieben wird. `n` wäre also die Speicherzelle und `temp` das Register.

```
The value is 17
3 processes created
```

Wiederholt man die Simulation, wird SPIN einen anderen Ablauf erzeugen und ein anderes Ergebnis zeigen.

Macht man das öfters, wird man feststellen, dass die Werte zwischen 10 und 20 liegen. Schaut man sich die denkbaren Abläufe genauer an, sieht man, dass dies kein Zufall ist.

21.1.2.2 Denkbare Abläufe

Die beiden Prozesse, nennen wir sie P_1 und P_2 , werden nebenläufig durchgeführt. Dadurch kann es vorkommen, dass die Aktionen der beiden Prozesse verschränkt ausgeführt werden.

Man kann sich zwei gewissermaßen extreme Ausprägungen dieser Verschränkung vorstellen, nämlich gar keine oder die „maximale“, bei der die Teilschritte der beiden Prozesse immer abwechselnd durchgeführt werden.

Ablauf 1: Die beiden Prozesse werden *sequenziell* ausgeführt, sage zuerst P_1 , dann P_2 .

Zuerst wird also P_1 die globale Variable schrittweise auf 10 erhöhen und danach macht P_2 weiter und inkrementiert auch 10 mal.

Das Ergebnis: The value is 20

Ablauf 2: Die beiden Prozesse sind verschränkt und zwar so, dass ihre Teilschritte immer *abwechselnd* aufeinander folgen.

Zuerst setzt also P_1 seine lokale Variable auf 1, dann P_2 die seinige ebenfalls auf 1. Dann weist P_1 der globalen Variablen n die 1 zu und schließlich macht auch P_2 diese Anweisung $n = 1$.

Jedes Paar von Schleifendurchläufen der beiden Prozesse erhöht also die globale Variable n um 1 und folglich hat sie am Ende den Wert 10.

Das Ergebnis: The value is 10

Also liegt die Schlussfolgerung nahe, dass unsere Überlegungen belegen, dass für jeden denkbaren verschränkten Ablauf der beiden Prozesse am Ende gilt: $10 \leq n \leq 20$. Wirklich?

21.1.2.3 Die Überraschung

Ein Student von Ben-Ari hat immer wieder die Simulation in SPIN aufgerufen. Er war offenbar ziemlich ausdauernd – und plötzlich verblüfft: Das Ergebnis war bei einem Durchlauf 9! (Ausrufezeichen, nicht Fakultät – versteht sich) [BA08, S.42]

Und es stellt sich sogar heraus, dass die Variable `n` am Ende sogar den Wert 2 haben kann.

“Intuitively, it seems as if ‘perfect’ interleaving represents the maximum ‘amount of interference’ possible, and for many years I taught that the final value of `n` must be between 10 and 20. It came somewhat of a shock when I discovered that the final value can be as low as 2!”³ [BA08, S. 39f.]

Besser also, man überprüft die Vermutung. Und das kann man mit SPIN tun.

21.1.2.4 Model Checking an diesem Beispiel

Eine einfache Art, SPIN zu verwenden, besteht darin, dass man bestimmte Bedingungen annimmt und SPIN überprüfen lässt, ob Gegenbeispiele existieren.

In unserem Beispiel wollen wir prüfen, ob als Ergebnis auch eine Wert herauskommen kann, der kleiner als 10 ist, ja sogar, ob als Ergebnis der Wert 2 auftreten kann.

Dazu hat SPIN das Konstrukt `assert`. Es bedeutet, dass SPIN ein Gegenbeispiel erzeugt, falls es *irgendeinen* Durchlauf gibt, bei dem die Bedingung des `assert` *nicht* erfüllt ist.

Wollen wir also prüfen, ob es einen Durchlauf mit dem Ergebnis 2 gibt, nehmen wir an, dass das nicht der Fall ist und lassen uns von SPIN ein Gegenbeispiel produzieren, also eines das 2 als Ergebnis hat. Tun wir das in der Datei `i12.pml`:

```

1 #include "for.h"
2 byte n = 0;
3
4 proctype P() {
5     byte temp;
6     for (i, 1, 10)

```

³ Das gleiche Szenario wird diskutiert in Michael Jacksons Buch über *Problem Frames* [Jac01, S. 320f.]. Jackson nennt dort als Quelle des Beispiels eine Aufgabe aus alten ACM-Tests, auf das ihn Anthony Hall aufmerksam gemacht hat. Ben-Ari hat diesen Test wohl nicht gekannt.

```

7      temp = n + 1;
8      n = temp
9      rof(i)
10 }
11
12 init {
13     atomic {
14         run P();
15         run P();
16     }
17     (_nr_pr == 1) ->
18     printf("The value is %d\n", n);
19     assert(n > 2)
20 }
```

Die einzige Veränderung ist die neue Zeile 19, bei der nach Ende der beiden Prozesse fordern, dass ($n > 2$) gilt. Wir lassen SPIN dann prüfen, ob das wirklich der Fall ist.

Dazu geht man so vor:

1. Mit dem Kommando `spin -a il2.pml` erzeugt SPIN C-Code für den *Verifier*, der dann die eigentliche Überprüfung durchführt. Ergebnis der Ausführung des Kommandos sind einige C-Quellen, insbesondere `pan.c` und `pan.h`.
2. Wir übersetzen diese Quellen mit `gcc -DSAFETY -o pan pan.c` (oder `-o pan.exe` unter Windows) Dabei verwenden wir das *define SAFETY*, weil wir nur eine Sicherheitsbedingung prüfen wollen.⁴
3. Nun kann der *Verifier* `pan` (*process analyzer*) ausgeführt werden und wir erhalten tatsächlich eine Fehlermeldung.

```
pan:1: assertion violated (n>2) (at depth 90)
pan: wrote il2.pml.trail
```

Die dabei erzeugte Datei `il2.pml.trail` enthält den „Zeugen“, einen Durchlauf, bei dem die gewünschte Bedingung verletzt wurde.

4. Die geführte Simulation `spin -t -p -g il2.pml` verwendet diesen Zeugen und gibt das Szenario aus, das zur Verletzung der Bedingung geführt hat – in unserem Beispiel einen Ablauf, der als Ergebnis den Wert 2 hat.

Die folgende Datei zeigt diesen Ablauf und es wird jetzt auch ersichtlich, wie dieses überraschende Ergebnis entstanden ist.

⁴ SPIN kann in vielfacher Hinsicht parametrisiert werden, um möglichst effizienten C-Code zu produzieren. Siehe <http://spinroot.com/spin/Man/Pan.html>

```

2: proc 0 (:init::1) il2.pml:22 (state 2) [(run P())]
3: proc 2 (P:1) il2.pml:13 (state 1) [i = 1]
4: proc 2 (P:1) il2.pml:13 (state 4) [else]
5: proc 1 (P:1) il2.pml:13 (state 1) [i = 1]
6: proc 1 (P:1) il2.pml:13 (state 4) [else]
7: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
8: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
9: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 1
10: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]
11: proc 2 (P:1) il2.pml:13 (state 4) [else]
12: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
13: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 2
14: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]
15: proc 2 (P:1) il2.pml:13 (state 4) [else]
16: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
17: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 3
18: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]
19: proc 2 (P:1) il2.pml:13 (state 4) [else]
20: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
21: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 4
22: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]
23: proc 2 (P:1) il2.pml:13 (state 4) [else]
24: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
25: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 5
26: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]
27: proc 2 (P:1) il2.pml:13 (state 4) [else]
28: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
29: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 6
30: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]
31: proc 2 (P:1) il2.pml:13 (state 4) [else]
32: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
33: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 7
34: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]
35: proc 2 (P:1) il2.pml:13 (state 4) [else]
36: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
37: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 8
38: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]
39: proc 2 (P:1) il2.pml:13 (state 4) [else]
40: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
41: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 9
42: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]

```

```

43: proc 2 (P:1) il2.pml:13 (state 4) [else]
44: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 1
45: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]
46: proc 1 (P:1) il2.pml:13 (state 4) [else]
47: proc 2 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
48: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
49: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 2
50: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]
51: proc 1 (P:1) il2.pml:13 (state 4) [else]
52: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
53: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 3
54: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]
55: proc 1 (P:1) il2.pml:13 (state 4) [else]
56: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
57: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 4
58: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]
59: proc 1 (P:1) il2.pml:13 (state 4) [else]
60: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
61: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 5
62: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]
63: proc 1 (P:1) il2.pml:13 (state 4) [else]
64: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
65: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 6
66: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]
67: proc 1 (P:1) il2.pml:13 (state 4) [else]
68: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
69: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 7
70: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]
71: proc 1 (P:1) il2.pml:13 (state 4) [else]
72: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
73: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 8
74: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]
75: proc 1 (P:1) il2.pml:13 (state 4) [else]
76: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
77: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 9
78: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]
79: proc 1 (P:1) il2.pml:13 (state 4) [else]
80: proc 1 (P:1) il2.pml:14 (state 5) [temp = (n+1)]
81: proc 1 (P:1) il2.pml:15 (state 6) [n = temp]
    n = 10
82: proc 1 (P:1) il2.pml:16 (state 7) [i = (i+1)]

```

```

83: proc 1 (P:1) il2.pml:13 (state 2) [((i>10))]
84: proc 2 (P:1) il2.pml:15 (state 6) [n = temp]
  n = 2
85: proc 2 (P:1) il2.pml:16 (state 7) [i = (i+1)]
86: proc 2 (P:1) il2.pml:13 (state 2) [((i>10))]
87: proc 2 terminates
88: proc 1 terminates
89: proc 0 (:init::1) il2.pml:24 (state 4) [(_nr_pr==1)]
  The value is 2
90: proc 0 (:init::1) il2.pml:25 (state 5) [printf('The value is %d\n',n)]
spin: il2.pml:26, Error: assertion violated
spin: text of failed assertion: assert((n>2))
91: proc 0 (:init::1) il2.pml:26 (state 6) [assert((n>2))]
spin: trail ends after 91 steps
#processes: 1
  n = 2
91: proc 0 (:init::1) il2.pml:27 (state 7) <valid end state>
3 processes created

```

In Schritt 8 speichert P_1 1 in seiner lokalen Variablen `temp`. Ab dann läuft erst mal nur P_2 und erhöht `n` schließlich auf 9 in Schritt 41.

In Schritt 44 schreibt nun P_1 seinen lokalen Wert von 1 in die Variable `n`.

In Schritt 47 liest P_2 die globale Variable und merkt sich 2 in seiner lokalen Variablen `temp`.

Ab jetzt ist nur P_1 dran, verwendet seinen lokalen Wert von 1 und erhöht sukzessive bis 10 in Schritt 81. Jetzt ist P_1 fertig, aber P_2 muss noch den 10. Schleifendurchgang beenden und schreibt den in Schritt 47 gemerkten Wert von 2 in die globale Variable `n`.

So ist es passiert!

21.1.3 Überprüfung kryptographischer Protokolle mit SPIN

Das Needham-Schroeder-Protokoll (1978) dient der wechselseitigen Authentifizierung zweier Partner. Das Protokoll ist im Grunde eine Kombination zweier Abläufe: (1) dem Beziehen von öffentlichen Schlüsseln von einem Authentifizierungsserver und (2) der eigentlichen wechselseitigen Authentifizierung der Partner. Es ist schon lange bekannt, dass das Protokoll unsicher ist, weil es nicht garantiert, dass die bezogenen öffentlichen Schlüssel aktuell und nicht kompromittiert sind. Dieses Problem lässt sich jedoch leicht durch Zeitstempel verhindern. Das Protokoll wird dargestellt in [Sch96, S. 69ff].

Erst 1995 hat Gavin Lowe [Low95] gezeigt, dass auch der Teil zur wechselseitigen Authentifizierung fehlerhaft ist. Diesen Nachweis kann man durch Model Checking führen.

21.1.3.1 Das Needham-Schroeder-Protokoll

Wir betrachten nur den Teil des Needham-Schroeder-Protokolls zur wechselseitigen Authentifizierung und unterstellen, dass die beiden Partner, Alice und Bob⁵ die öffentlichen Schlüssel des jeweils anderen Partners bereits haben.

Sei im Folgenden K_A der öffentliche Schlüssel von Alice und K_B der von Bob. Mit $enc_K(M)$ bezeichnen wir die Verschlüsselung der Nachricht M mit dem Schlüssel K .

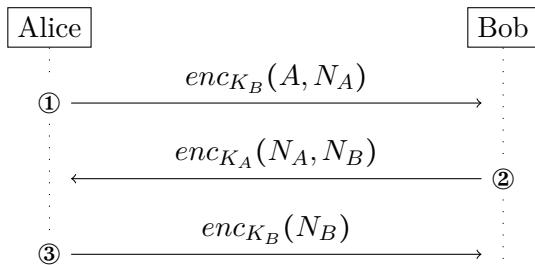


Abbildung 21.2: Authentifizierung nach Needham-Schroeder

Dieser Teil des Needham-Schroeder-Protokolls besteht dann aus drei Schritten (siehe 21.2):

- ① Alice erzeugt eine Zufallszahl (*nonce*) N_A und sendet $enc_{K_B}(A, N_A)$, wobei A eine Nachricht ist, die die Identität von Alice enthält. Bob entschlüsselt diese Nachricht und kennt nun das Geheimnis N_A von Alice.
- ② Bob erzeugt auch eine Zufallszahl N_B und sendet $enc_{K_A}(N_A, N_B)$ zurück. Alice entschlüsselt die Nachricht und kennt nun das Geheimnis N_B von Bob. Da Alice ihr eigenes Geheimnis von Bob zurückhält, ist sie überzeugt, mit Bob zu kommunizieren.
- ③ Alice sendet nun die von Bob erhaltene Zufallszahl N_B in $enc_{K_B}(N_B)$ zurück. Bob entschlüsselt die Nachricht. Er erhält seine eigene Zufallszahl zurück und ist überzeugt, mit Alice zu kommunizieren.

Ein Angreifer (*intruder*) kann wie jeder andere Teilnehmer innerhalb eines Netzwerkes als Partner in einer Kommunikation auftreten. Er

⁵ In der Beschreibung kryptographischer Protokolle wird üblicherweise die einleitende Beteiligte *Alice* und der andere Partner *Bob* genannt.

kann aber nur solche Nachrichten entschlüsseln, die mit seinem eigenen Schlüssel verschlüsselt wurden. Er kann natürlich Tricks versuchen, z.B. sich als eine andere Person ausgeben, oder Nachrichten abfangen und später verschicken u.ä.

21.1.3.2 Modellierung in Promela

Stephan Merz⁶ Protokoll sowie die Aktionen eines potentiellen Angreifers in PROMELA, der Sprache des Model Checkers SPIN modelliert [Mer01]. Man kann dann mit SPIN zeigen, dass das Protokoll unterlaufen werden kann.

Das Modell des Protokolls macht einige vereinfachende Annahmen:

- Es gibt drei Akteure: Bob, Alice und I, den Intruder.
- Alice initiiert den Ablauf mit Bob oder I.
- Bob tritt nur als Antwortender auf.
- Es gibt nur einen Lauf für Alice und Bob.
- Der Intruder kann Nachrichten auf dem Netzwerk lesen.

Merz definiert zunächst einige Typen, die die Lesbarkeit der Spezifikation erleichtern:

```
mtype = { ok, err, msg1, msg2, msg3, keyA, keyB, keyI,
          agentA, agentB, agentI, nonceA, nonceB, nonceI };
```

Nachrichten bestehen aus einem Schlüssel (dem öffentlichen Schlüssel des intendierten Empfängers) und zwei Datenteilen:

```
typedef Crypt { mtype key, data1, data2 };
```

Das Netzwerk, auf dem die Kommunikation abgewickelt wird, wird modelliert als Kanal (*channel*), den alle Akteure verwenden. In PROMELA wird synchrone Kommunikation durch einen Kanal der Puffergröße 0 dargestellt. Das bedeutet, dass der Sender warten muss, wenn der Empfänger nicht bereit ist und umgekehrt.

```
chan network = [0] of { mtype, /* msg# */
                         mtype, /* receiver */
                         Crypt /* encrypted msg */};
```

Alice wird nun als Prozess modelliert. In PROMELA wird in einer **if**-Anweisung eine aus eventuell mehreren ausführbaren Anweisungen (nicht-deterministisch) gewählt und ausgeführt. Im Prozess für Alice wählt

⁶ Webseite von Stephan Merz bei INRIA Nancy.

Alice somit in der `if`-Anweisung Bob oder den Intruder als Partner der Kommunikation.

Alice verschickt dann ihre erste Nachricht und wartet auf die Antwort. Hat die Antwort ihren Schlüssel und enthält sie als ersten Teil ihre Zufallszahl, liest Alice das Geheimnis ihres Partners. Dann verschickt sie ihre Antwort. Kommt der Prozess zu dieser Stelle, ist der Status für Alice in Ordnung.

```
mtype partnerA;
mtype statusA = err;
active proctype Alice() {
    mtype pkey, pnonce;
    Crypt data;

    if /* choose a partner for this run */
        :: partnerA = agentB; pkey = keyB;
        :: partnerA = agentI; pkey = keyI;
    fi;
    network ! (msg1, partnerA, Crypt{pkey, agentA, nonceA});

    network ? (msg2, agentA, data);
    (data.key == keyA) && (data.info1 == nonceA);
    pnonce = data.info2;

    network ! (msg3, partnerA, Crypt{pkey, pnonce, 0});
    statusA = ok;
}
```

Bob wird ganz analog modelliert.

Der Intruder versucht nun Nachrichten, die er empfängt zu entschlüsseln und weiterzuleiten oder, wenn er sie nicht entschlüsseln kann, wie empfangen weiterzuleiten.

```
bool knows_nonceA, knows_nonceB;

active proctype Intruder() {
    mtype msg, receipt;
    Crypt data, intercepted;

    do
        :: network ? (msg, _, data) ->
            if /* perhaps store the message */
                :: intercepted = data;
                :: skip;
            fi;
            if /* record newly learnt nonces */
                :: (data.key == keyI) ->
```

```

if
:: (data.info1 == nonceA) || (data.info2 == nonceA)
-> knows_nonceA = true;
:: else -> skip;
fi;
/* similar for knows_nonceB */
:: else -> skip;
fi;
:: /* Replay or send a message */
if /* choose message type */
:: msg = msg1;
:: msg = msg2;
:: msg = msg3;
fi;
if /* choose recipient */
:: recpt = agentA;
:: recpt = agentB;
fi;
if /* replay intercepted message or assemble it */
:: data = intercepted;
:: if
:: data.info1 = agentA;
:: data.info1 = agentB;
:: data.info1 = agentI;
:: knows_nonceA -> data.info1 = nonceA;
:: knows_nonceB -> data.info1 = nonceB;
:: data.info1 = nonceI;
fi;
/* similar for data.info2 and data.key */
fi;
network ! (msg, recpt, data);
od;
}

```

21.1.3.3 Verifikation mit Spin

Zur Verifikation müssen wir nun die zu überprüfende Eigenschaft als LTL-Formel ausdrücken:

Seien $statusA, statusB$ die Flags, die anzeigen, ob der Lauf aus Sicht von Alice bzw. Bob erfolgreich durchlaufen wurde. $partnerA, partnerB$ seien die Partner, die ihre Geheimnisse ausgetauscht haben und $knows_nonceA, knows_nonceB$ gibt an, ob der Intruder das Geheimnis von Alice bzw. Bob kennt.

Dann kann man die gewünschte Eigenschaft des Protokoll durch folgende vier Formeln der LTL ausdrücken:

- $\square (statusA = ok \wedge statusB = ok \rightarrow (partnerA = agentB \rightarrow partnerB = agentA))$
- $\square (statusA = ok \wedge statusB = ok \rightarrow (partnerB = agentA \rightarrow partnerA = agentB))$
 - $\square (statusA = ok \wedge partnerA = agentB \rightarrow \neg knows_nonceA)$
 - $\square (statusB = ok \wedge partnerB = agentA \rightarrow \neg knows_nonceB)$

Diese Aussagen übernehmen wir in die PROMELA-Datei:

```
#define success statusA == ok && statusB == ok
#define aliceBob  partnerA == bob
#define bobAlice  partnerB == alice
#define nonceASecret !knowNA
#define nonceBSecret !knowNB

/* Verification formulae in LTL */
ltl fml1 { [] (success -> (aliceBob -> bobAlice)) }
ltl fml2 { [] (success -> (bobAlice -> aliceBob)) }
ltl fml3 { [] ((aliceBob && success) -> nonceASecret) }
ltl fml4 { [] ((bobAlice && success) -> nonceBSecret) }
```

Wir können nun die einzelnen Aussagen checken:

```
spin -a NeedhamSchroeder.pml
gcc -o pan pan.c
pan -N fml2 NeedhamSchroeder.pml
```

Mit der zweiten Aussage `fml2` zeigt der von SPIN erzeugte *Verifier pan* (*process analyzer*) einen Fehler an.

21.1.3.4 Analyse des Ergebnisses

Wenn bei der Verifikation einer Spezifikation gegenüber einer LTL-Formel ein Fehler auftritt, fertigt SPIN eine sogenannte *Trail*-Datei, die einen Pfad zeigt, auf dem die Formel verletzt wird. Diese Datei kann man für die Analyse des Ergebnisses verwenden. In unserem Fall ist es am einfachsten mit

```
spin -t -c NeedhamSchroeder.pml // Ausgabe auf Konsole
spin -t -M NeedhamSchroeder.pml // Ausgabe als Tcl/Tk-Datei
// und Anzeige mit wish
```

eine Ausgabe als MSC (*Message Sequence Chart*) zu erzeugen und diese zu analysieren. Unsere Analyse hat Abb. 21.3 zum Resultat.

Wenn man dieses Diagramm genau durchsieht, erhält man das in Abb. 21.4 dargestellte Szenario.

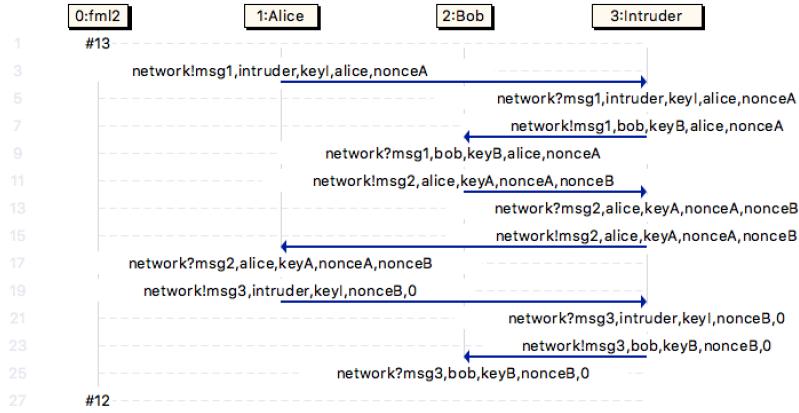


Abbildung 21.3: Ablauf des Angriffs als Message Sequence Chart

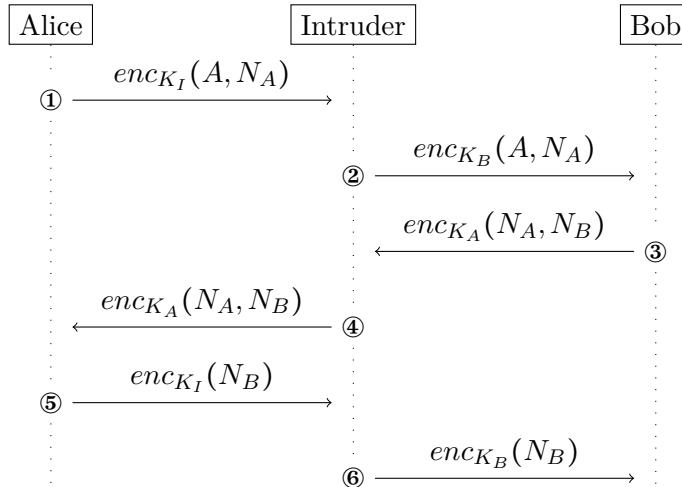


Abbildung 21.4: Angriffsszenario zum Needham-Schroeder-Protokoll

1. Alice möchte eine Sitzung mit I eröffnen und sendet deshalb ihre Identität und ihr Geheimnis. I kann diese Nachricht entschlüsseln und kennt nun das Geheimnis von Alice.
 2. I sendet die Nachricht von Alice an Bob weiter, verschlüsselt mit dessen Schlüssel. I spiegelt vor, Alice zu sein.
 3. Bob denkt, mit Alice zu kommunizieren und sendet nun sein Geheimnis, das I auf dem Netzwerk abfängt.
 4. Diese Nachricht von Bob kann I zwar nicht entschlüsseln, er leitet sie aber an Alice weiter.
- Alice sieht dies als die Antwort auf ihre eigene Nachricht an und

entschlüsselt sie. Sie kennt nun das Geheimnis von Bob, hält dieses jedoch für das Geheimnis von I.

5. Alice sendet nun das vermeintliche Geheimnis von I (tatsächlich das von Bob) an I zurück. I kann nun die Nachricht entschlüsseln und kennt nun auch das Geheimnis von Bob.
6. I sendet die Nachricht von Alice verschlüsselt mit Bobs Schlüssel an Bob weiter. Bob denkt, mit Alice zu sprechen!

Das Ergebnis ist somit:

- I, der *Intruder*, kennt die Geheimnisse von Alice und Bob.
- Alice denkt (korrekterweise) eine Sitzung mit I initiiert zu haben.
- Bob denkt mit Alice eine Sitzung initiiert zu haben, spricht jedoch tatsächlich mit I. Der Intruder hat die Identität von Alice übernommen.

21.1.4 Anwendungen in der Industrie

In diesem Abschnitt werden zunächst einige Beispiele für die Nutzung von SPIN in der Industrie genannt.

Neben SPIN gibt es noch andere Model Checker, wie zum Beispiel TLA⁺, entwickelt von Leslie Lamport. Zum Abschluss des Abschnitts über Anwendungen von Model Checking in der Industrie lernen wir noch einige Beispiele der Nutzung von TLA⁺ kennen.

21.1.4.1 Beispiele der Nutzung von SPIN

Ursprünglich wurde SPIN für die Verifikation von Protokollen eingesetzt, weshalb das von SPIN erzeugte Verifikationsprogramm auch `pan` für *Protocol Specification Analyzer* heißt. SPIN kann jedoch für viele andere Fragestellungen verwendet werden. In diesem Abschnitt werden einige Beispiele für Model Checking mit SPIN in der Industrie vorgestellt.

Sturmflutschutzsystem Rotterdam (1995) Im Projekt BOS „Beslis- en Oudersteunend Systeem“ (Entscheidungs- und Unterstützungs- system) wurde das System entwickelt, das entscheidet, wann und wie die Sturmflutsperren auf dem Kanal, der Rotterdam mit der Nordsee verbindet, bewegt werden. Das System steuert vollautomatisch die Bewegung des Sturmflutschutzsystems und verwendet dabei eine Vielzahl von Sensoren, die Wassertiefe, Wetterdaten, Flutdaten usw. messen. Das System wurde von der Firma Computer Management Group (CMG) Den Haag in Zusammenarbeit mit der Formal Methods Group der Universität Twente entwickelt. Für die Spezifikation des Systems wurden zwei

Sprachen verwendet: PROMELA für die Spezifikation der Funktionen und Z⁷ für die der Daten. [Kar96, BV11]

NASA Deep Space One Mission (1998) Die Raumsonde *Deep Space One*, kurz DS1, wurde von der NASA zwecks Erprobung neuer Technologien entwickelt. Sie wurde am 24. Oktober 1998 gestartet und war bis Dezember 2001 im Einsatz.

Bei der Entwicklung der Flugsoftware wurde die Sprache *Executive Support Language* (ESL) eingesetzt, die speziell für die Unterstützung der Entwicklung reaktiver Systeme für autonome Roboter und Raumschiffe ausgelegt ist. SPIN wurde verwendet, um eine Komponente des *Remote Agent* (RA) der Flugsoftware von DS1 zu verifizieren. Der Code dieser Komponente wurde manuell in eine PROMELA-Spezifikation übersetzt und dann mit SPIN überprüft. 5 Fehler konnten auf diese Weise gefunden, 4 davon waren wichtig. “According to the RA programming team the effort has had a major impact, locating errors that would probably not have been located otherwise and identifying a major design flaw...” [HLP01]

Nachträglich wurden von Peter R. Glück (NASA) und Gerard J. Holzmann, dem Entwickler von SPIN, weitere Analysen durchgeführt. Es handelte sich dabei um eine in C programmierte Komponente der Flugsoftware von DS1. Die Besonderheit dieser Analyse besteht darin, dass aus dem Code die PROMELA-Spezifikation automatisch extrahiert wurde. Dazu wurde ein Werkzeug namens *FeaVer* eingesetzt, das im Prinzip wie ein C-Compiler arbeitet, jedoch keinen Binärcode, sondern für die zu untersuchenden Funktionen PROMELA-Code erzeugt. Bei dieser Analyse konnten bereits bekannte Fehler gefunden werden, aber auch eine vorher unbekannte *race condition*. [GH02]

Telekommunikationssoftware von Lucent (1998) Die eben erwähnte Technik der Extraktion von PROMELA-Spezifikationen mittels *FeaVer* wurde auch bei der formalen Verifikation von Softwarekomponenten der *PathStar Access Server* Telekommunikationssoftware von Lucent Technologies. Aus dieser Anwendung kommt auch die Bezeichnung *FeaVer*, die für *Feature Verification* steht. Das Ergebnis dieses Einsatzes von SPIN: “We intercepted approximately 75 errors in the implementation

⁷ Z ist eine Spezifikationssprache basierend auf der Mengenlehre und der Prädikatenlogik erster Stufe. (Also eine formale Methode, die gut in den zweiten Teil dieser Vorlesung gepasst hätte. Ich habe mich jedoch für Alloy als Beispiel entschieden, weil man mit dem Alloy Analyzer so schön interaktiv arbeiten kann. Und weil Alloy mit seiner Technik des Model Finding so gut zur Behandlung der SAT Solver im ersten Teil passt.) Z wurde von Jean-Raymond Abrial entwickelt und von der Programming Research Group der Universität Oxford weiterentwickelt. 2002 wurde Z durch die ISO (International Organization for Standardization) standardisiert.

of the feature code by repeated verifications. Many of these errors were considered critically important by the programmers, especially in the early phases of the design. About 5 of the errors caught were also found independently by the normal system test group, especially in the later phases of the project.”[\[HS00\]](#)

Spezifikation der Enterprise JavaBeans (2001) Die bisherigen Beispiele könnten den Eindruck erwecken, dass sich Model Checking ausschließlich für reaktive Systeme eignet. Aber auch für Geschäftsanwendungen ist Model Checking eine relevante Methode der Verifikation. Enterprise JavaBeans (EJB) definieren eine Spezifikation für eine Komponentenarchitektur in verteilten Geschäftsanwendungen. Es ist möglich die Spezifikation von *Entity Beans* und *Session Beans* in PROMELA zu formalisieren und das verlangte Verhalten durch Formeln der LTL auszudrücken. Dadurch wurden potenzielle Fehler in der Spezifikation von Version 1.0 der EJB aufgedeckt. Insbesondere aber zwingt die formale Spezifikation des Verhaltens der EJBs zu einer Präzision, die das in englischer Sprache geschriebene Spezifikationsdokument vermissen lässt. [\[NT01\]](#)

Toyota – Untersuchung amerikanischer Behörden (2010) In den USA gab es mehrere Autounfälle mit Fahrzeugen von Toyota, bei denen Fahrer behaupteten, dass die Bremse nicht richtig funktioniert habe bzw. das Auto plötzlich autonom beschleunigt habe. Schließlich gingen etwa 3000 Beschwerden bei amerikanischen Behörden ein. Das Verkehrsministerium der USA hat daraufhin eine Untersuchung angeordnet, die im Februar 2011 abgeschlossen wurde. Das Ergebnis: “NASA engineers found no electronic flaws in Toyota vehicles capable of producing the large throttle openings required to create dangerous high-speed unintended acceleration incidents.”⁸

Als Teil dieser Untersuchung wurde auch die im Toyota Camry eingesetzte Software analysiert, wobei auch SPIN zum Einsatz kam. Hierbei konnten Probleme identifiziert werden, jedoch keine, die zur behaupteten plötzlichen autonomen Beschleunigung des Fahrzeugs hätten führen können. Im Untersuchungsbericht⁹ heißt es: “The logic model verifications identified a number of potential issues. All of these issues involved unrealistic timing delays in the multiprocessing, asynchronous software control flow. Even if they were to occur, none of these potential issues could be

⁸ U.S. Department of Transportation Releases Results from NHTSA-NASA Study of Unintended Acceleration in Toyota Vehicles vom 8. Februar 2001.

⁹ NASA Engineering and Safety Center National Highway Traffic Safety Administration Toyota Unintended Acceleration Investigation Appendix A. Software Seite 11.

tied to unintended acceleration. No cause for unintended acceleration was found from the logic model verification effort.”

Industrieroboter (2011) In diesem Beispiel wurde ein Compiler entwickelt, der die Steuerungssoftware von Industrierobotern in PROMELA-Spezifikationen übersetzt. Dadurch wurde die systematische Analyse von Eigenschaften ermöglicht. Die Ergebnisse der Analyse mit SPIN wurden im Falle der Verletzung von Integritätsbedingungen in den Originalkontext des Codes der Roboter zurückgespiegelt, so dass die Programmierer der Roboter die problematischen Stellen in ihrem Code analysieren konnten.

Als Fallstudie wurde die Roboter-Software auf Verklemmungen und Vermeidung von Kollisionen hin untersucht. Dabei wurde ein Deadlock in einer Karosserie-Schweißstation mit 9 Robotern entdeckt. Der Fehler konnte dadurch korrigiert werden und das verbesserte Programm konnte mit SPIN verifiziert werden. Dieser Einsatz von SPIN wurde bei AUDI mit Unterstützung der Fakultät für Informatik der TU München durchgeführt. [WBBK11]

Erfahrungen mit SPIN bei Telekommunikationssystemen In diesem Beispiel wurde Model Checking mit SPIN für die Analyse des Failover-Algorithmus eines kommerziellen Systems namens WebArrow für webbasierte Konferenzen eingesetzt. Das Failover dieses Systems ist durch komplexe verteilte Algorithmen realisiert, die in einer Studie mittels SPIN analysiert wurden. Die Autoren der Studie schildern ihre Erfahrungen eher als ernüchternd:

- Um ein handhabbares Modell in PROMELA zu erhalten, mussten die Autoren der Studie viele Details des Systems weglassen, so dass sie am Ende Zweifel an der Aussagekraft der Analyse hatten.
- Den zeitlichen Aufwand für die Erstellung des Modells scheint den Autoren als zu hoch für die heutigen Erfordernisse des „Time-to-Market“.
- Die Dokumentation der Entwickler des Systems im Code reichten für die Spezifikation in PROMELA nicht aus, was die Erstellung des Modells sehr erschwerte.

Das Fazit der Autoren: “Our conclusion is that model checking technology is not yet sufficiently mature for use in the development of telecommunications systems such as WebArrow.” [LDG08]

Dieses Beispiel zeigt, dass es nicht einfach ist, Model Checking in der Softwareentwicklung zu verwenden. Das mag daran liegen – wie die Autoren der Studie darstellen –, dass die Werkzeuge für akademische

Beispiele geeignet sein mögen, jedoch für den Einsatz in der Industrie (noch) nicht wirklich. Andererseits stellt sich aber schon die Frage wie die Entwickler des Systems eine für die Implementierung ausreichende Dokumentation haben sollten, die sich jedoch für die Formalisierung als unzureichend erwiesen hat. Denn schließlich ist die Codierung der Algorithmen ja nichts anderes als eine Formalisierung. Wie können die Entwickler dann Sicherheit darüber haben, dass ihre Implementierung die geforderten Eigenschaften erfüllt?

21.1.4.2 Beispiele der Nutzung von TLA⁺

Leslie Lamport nennt auf seiner Webseite [Industrial Use of TLA⁺](#) einige Beispiele für den Einsatz von TLA⁺ in der Industrie, unter Anderem die folgenden:

Amazon Amazon Web Services AWS wie zum Beispiel S3 (*Simple Storage Service*) sind für die Anwender einfach zu verwenden, intern aber komplexe verteilte Systeme. Um Performance und Datensicherheit zu gewährleisten, verwenden diese Dienste fehlertolerante verteilte Algorithmen für Replikation, Konsistenz, Synchronisationskontrolle, automatische Skalierung, Lastausgleich und andere Koordinierungsaufgaben. Bei der Entwicklung und Implementierung dieser Algorithmen hat sich herausgestellt, dass die Standardverfahren der Qualitätssicherung subtile Fehler nicht finden konnten. Techniken wie Design-Reviews, Code-Reviews, statische Code-Analyse sowie diverse Testverfahren konnten Fehler nicht entdecken, weil „die menschliche Intuition die wahre Wahrscheinlichkeit von vermeintlich ‚äußerst seltenen‘ Kombinationen von Ereignissen in Systemen, die in einer Größenordnung von Millionen von Anforderungen pro Sekunde arbeiten, nur unzureichend abschätzen kann.“ [NRZ⁺¹⁵, Eigene Übersetzung]

Amazon verwendet TLA⁺ in 10 großen, komplexen Systemen (Angabe von 2015) in 7 Entwicklerteams. Die oft geäußerte Befürchtung, dass formale Methoden schwierig einzuführen seien, hat sich bei Amazon nicht bestätigt. Eines der ersten Projekte, in denen bei Amazon TLA⁺ eingesetzt wurde, war DynamoDB, eine NoSQL-Datenbank. Die Spezifikation der Algorithmen für die Replikation war 939 Zeilen in TLA⁺ lang. Sie wurde in der Amazon Cloud auf einem Cluster von 10 EC2-Instanzen (*Elastic Compute Cloud*) mit 8 Kernen und 23 GB RAM gecheckt. Dabei wurden drei subtile Fehler entdeckt, die durch Testen wahrscheinlich niemals hätten entdeckt werden können. Nach den guten Erfahrungen in diesem Projekt, wurde der Einsatz von TLA⁺ bei Amazon auf weitere Projekte ausgedehnt.

Die Autoren fassen ihre Erfahrungen mit TLA⁺ in [NRZ⁺¹⁵] so zusam-

men:

- Formal methods find bugs in system designs that cannot be found through any other technique we know of.
- Formal methods are surprisingly feasible for mainstream software development and give good return on investment.
- At Amazon, formal methods are routinely applied to the design of complex real-world software, including public cloud services.

Chris Newcombe, der Hauptautor des eben zitierten Artikels, sagt über seine Erfahrungen mit TLA⁺: “TLA⁺ is the most valuable thing that I’ve learned in my professional career. It has changed how I work, by giving me an immensely powerful tool to find subtle flaws in system designs. It has changed how I think, by giving me a framework for constructing new kinds of mental-models, by revealing the precise relationship between correctness properties and system designs, and by allowing me to move from ‘plausible prose’ to precise statements much earlier in the software development process. At Amazon we’ve used TLA⁺ with great success on several projects, and I’m hoping to increase that momentum.” [In einem Beitrag in einer [Google Group](#).]

Microsoft Bei Microsoft wurde TLA⁺ seit 2004 in verschiedenen Projekten eingesetzt. Bei der Entwicklung der Xbox 360 wurde mit TLA⁺ ein subtler Fehler im *Memory Coherence Protocol* des Chips der XBox gefunden. Der Chip wurde von IBM hergestellt. Die Ingenieure von IBM wollten zunächst nicht glauben, dass es ein Problem gibt. Erst ein paar Wochen später erkannte auch IBM, dass es sich um einen echten Fehler handelte, der durch ihre Regressionstests nicht erkannt worden war. Diese Anekdote erzählt Leslie Lamport in seinem Vortrag [Thinking for Programmers](#) am 30. März 2014.

Bei der Entwicklung der Webservices von Microsoft (Azure) wurde TLA⁺ an vielen Stellen eingesetzt, wie David Langworthy in seiner Key Note [TLA+ at Microsoft: 16 Years in Production](#) auf der TLA+ Conf 2019 berichtet:

- Azure Cosmos DB – Präzise Definition der Konsistenzlevel im verteilten Datenbankmanagementsystem Cosmos DB, nachdem in ersten Versionen mit dem Model Checker von TLA⁺ Fehler gefunden worden waren.
- Azure Service Fabric, Plattform für verteilte Systeme, die das Packen, Bereitstellen und Verwalten skalierbarer und zuverlässiger Microservices und Container vereinfacht – Verifikation mittels TLA⁺.

- Azure Batch Poolserver – TLA⁺ hilft, die Sicherheits- und Lebendigkeitseigenschaften besser zu verstehen und präzise zu beschreiben.
- Azure Storage Paxos Ring Management – in diesem Projekt konnte durch den Model Checker von TLA⁺ Verletzungen der Sicherheitsbedingungen entdeckt werden.
- Azure Networking – hier hat der Model Checker zwei Verletzungen von Invarianten gefunden.

OpenComRTOS OpenComRTOS ist ein verteiltes Echtzeit-Betriebssystem. Es ist Nachfolger von Virtuoso, welches bei der Rosetta-Mission der ESA 2004 - 2016 eingesetzt wurde.

Bei der Entwicklung von OpenComRTOS wurde TLA⁺ von Beginn an eingesetzt. Die Entwickler beschreiben die Vorgehensweise so:

“Applied on the development of OpenComRTOS, the [development] process was started by elaborating a first set of requirements and specifications. Next an initial architecture was defined. Starting from this point on, two groups started to work in parallel. The first group worked out an architectural model while a second group developed an initial formal model using TLA+/TLC. This model was incrementally refined.

At each review meeting between the software engineers and the formal modeling engineer, more details were added to the model, the model was checked for correctness and a new iteration started. This process was stopped when the formal model was deemed close enough to the implementation architecture. Next, a simulation model was developed on a PC (using Windows NT as a virtual target). This code was then ported to a real 16bit microcontroller. On this target a few target specific optimizations were performed on the implementation, while fully maintaining the design and architecture. The software was written in ANSI C and verified with a MISRA¹⁰ rule checker. Finally the reverse process was undertaken. For each service class a formal model was built matching the implementation and essential properties were verified.”

[VdJM08, VBF⁺¹¹]

21.2 Zielemodell in der Anforderungsanalyse

Die lineare temporale Logik kann nicht nur eingesetzt werden, um das *Wie* eines Softwaresystems präzise zu beschreiben, sondern auch wenn es um das *Was* geht, in der Softwareanforderungsanalyse. In diesem

¹⁰ MISRA-C (entwickelt von der *Motor Industry Software Reliability Association*) ist ein Programmierstandard für die Sprache C. Er definiert eine Untermenge des Sprachumfangs von C und Regeln für die Verwendung von C.

Abschnitt wollen wir einen Ansatz betrachten, bei dem systematisch ein *Zielemodell* für ein Softwaresystem entwickelt wird. Es gibt verschiedene Formen des Zielemodells; hier befassen wir uns mit dem Zielemodell in *KAOS* (*Knowledge Acquisition in autOmated Specification* oder auch *Keep All Objectives Satisfied*), einer Methode, die Axel van Lamsweerde¹¹ entwickelt hat [vL08, van09].

21.2.1 Das Systemmodell in KAOS

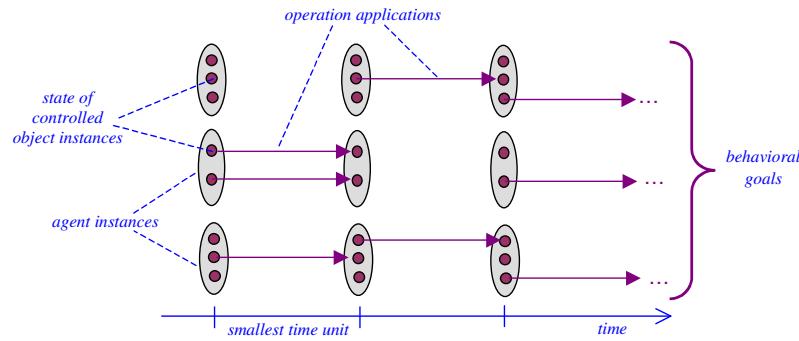


Abbildung 21.5: Das Systemmodell in KAOS [vL08]

In Abbildung 21.5 wird das Konzept eines Softwaresystems dargestellt, wie es Axel van Lamsweerde für die KAOS-Methode unterstellt. Parallel Akteure (*agents*) steuern den Zustand von Objektinstanzen. In einem diskreten Modell der Zeit wenden die Akteure Operationen an, die von Zeitpunkt zu Zeitpunkt die Zustände der Objektinstanzen verändern. Im Verlaufe der Zeit werden dadurch die funktionalen sowie die Qualitätsziele des Systems erreicht.

Diese Vorstellung eines Softwaresystems setzt eine klare Definition der Ziele voraus, die durch das Zielemodell erreicht wird. Die Akteure führen ihre Operationen so durch, dass die Integritätsbedingungen erhalten oder erreicht werden, die für die Zielerreichung erforderlich sind.

21.2.2 Das Zielemodell

Das Zielemodell in KAOS umfasst auch sogenannte *soft goals*, die nicht exakt nachgeprüft werden können, wie zum Beispiel das Ziel der Zufriedenheit von Kunden. Solche Ziele werden häufig formuliert und bewertet, wenn es um alternative Lösungen geht. Solche Ziele wollen wir im Folgenden nicht betrachten, sondern funktionale und Qualitätsziele, die in KAOS als *behavioral goals* bezeichnet werden.

¹¹ Axel van Lamsweerde, belgischer Informatiker.

Ziele (in diesem Sinne) sind präskriptive Aussagen über die Effekte, die ein System durch die Kooperation seiner Akteure im Anwendungsgebiet erreichen soll.

Im Zielemodell werden Ziele systematisch verfeinert. Dabei gibt es zwei Möglichkeiten:

- Ein Ziel wird erreicht dadurch, dass alle seine Unterziele erreicht werden. In diesem Fall spricht man von einer AND-Verfeinerung des Oberziels.
- Ein Ziel kann durch unterschiedliche Unterziele erreicht werden. In diesem Fall handelt es sich um eine OR-Verfeinerung des Oberziels. Eine OR-Verfeinerung modelliert verschiedene alternative Lösungsmöglichkeiten zur Erreichung des Ziels oder Variationspunkts, wenn das Zielemodell eine Softwareproduktlinie betrifft.

Graphisch wird das Zielemodell in KAOS durch einen Und-Oder-Baum dargestellt. Ein Beispiel für einen Teil eines Zielemodells für ein System der Steuerung eines Zugs wird in Abbildung 21.6 dargestellt. An diesem Beispiel lassen sich die Konzepte des Zielemodells in KAOS illustrieren.

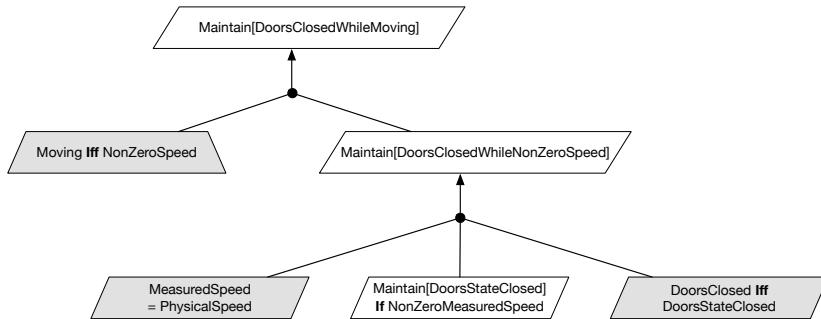


Abbildung 21.6: Ausschnitt eines Zielemodells für eine Zugsteuerung, nach [van09]

Im Zielediagramm können Ziele auftreten, die durch Parallelogramme visualisiert werden. Außerdem können für die Erreichung eines Ziels Gegebenheiten eines Anwendungsgebiets eine Rolle spielen. Diese Domäneneigenschaften werden als Trapez dargestellt. Ferner kann es Erwartungen geben, die sich an Akteure richten, die nicht durch das Softwaresystem selbst gesteuert werden. Solche Erwartung werden als grau hinterlegtes Parallelogramm dargestellt.

Im Beispiel in Abbildung 21.6 wird das Ziel `Maintain[DoorsClosedWhile-`

Moving] verfeinert. Diese Ziel ist eine Sicherheitseigenschaft, die in allen Zuständen des Zugsystems erreicht werden muss. Dies wird durch das **Maintain** festgelegt. Tatsächlich handelt es sich dabei um einen temporalen Operator. So geht an dieser Stelle die lineare temporale Logik in das Zielemodell ein. In KAOS sind folgende Schlüsselworte für die Charakterisierung von Zielen vorgesehen:

Maintain[Ziel]	\Box	Ziel
Avoid[Ziel]	$\Box \neg$	Ziel
Achieve[Ziel]	\Diamond	Ziel

Das Ziel **Maintain[DoorsClosedWhileMoving]** wird erreicht durch die Domänenegenschaft **Moving Iff NonZeroSpeed** zusammen mit dem Unterziel **Maintain[DoorsClosedWhileNonZeroSpeed]**.

Blätter des Zielebaums sind Anforderungen oder Erwartungen. Anforderungen sind Ziele, für die ein bestimmter Akteur des Softwaresystems verantwortlich ist. In unserem kleinen Beispiel ist es eine Komponente der Zugsteuerung, die verantwortlich ist für das Ziel **Maintain[DoorsState-Closed If NonZeroMeasuredSpeed]**. Dabei müssen gleichzeitig zwei Erwartungen an externe Akteure erfüllt sein: Von einem Geschwindigkeitssensor als externem Akteur wird erwartet, dass die von ihm gemessene Geschwindigkeit der tatsächlichen Geschwindigkeit des Zugs entspricht. Und von einem weiteren externen Akteur, dem Aktor, der die Türen bewegt, wird erwartet, dass der von ihm gemeldete Zustand der Türen auch dem tatsächlichen Zustand der Türen entspricht.

Man kann das Zielemodell somit als eine Formel der linearen temporalen Logik auffassen und Techniken der Analyse solcher Formeln, wie zum Beispiel Model Checking oder auch Prüfung auf Erfüllbarkeit, anwenden. Dadurch wird das Zielemodell formal analysierbar.

Dies war der kurze Blick auf das Zielemodell in der Anforderungsanalyse und wie temporale Logik ins Spiel kommt. In Wirklichkeit geht die Verwendung der temporalen Logik in der KAOS-Methode viel weiter: man kann Ziele formal definieren, in dem man präzise ausdrückt, welche Zustände der Objektinstanzen des Systems die Akteure erreichen oder stets garantieren müssen. Mehr dazu im Buch von Axel van Lamsweerde [[van09](#)].

Literaturverzeichnis

- [ABKS13] SVEN APEL, DON BATÓRY, CHRISTIAN KÄSTNER, ET AL. *Feature-Oriented Software Product Lines*. Berlin Heidelberg: Springer, 2013.
- [BA08] MORDECHAI BEN-ARI. *Principles of the Spin Model Checker*. London: Springer, 2008.
- [BA12] MORDECHAI BEN-ARI. *Mathematical Logic for Computer Science: Third Edition*. London: Springer, 2012.
- [Bat05] DON S. BATÓRY. Feature Models, Grammars, and Propositional Formulas. In: *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings, Lecture Notes in Computer Science*, Band 3714, S. 7–20. Springer, 2005.
- [BBGS06] ALEXANDER BOLOTOV, ARTIE BASUKOSKI, OLEG M. GRIGORIEV, ET AL. Natural Deduction Calculus for Linear-Time Temporal Logic. In: MICHAEL FISHER, WIEBE VAN DER HOEK, BORIS KONEV, ET AL., Hg., *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Liverpool, UK, September 13-15, 2006, Proceedings, Lecture Notes in Computer Science*, Band 4160, S. 56–68. Springer, 2006.
- [BC04] YVES BERTOT, PIERRE CASTÉRAN. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Berlin: Springer, 2004.
- [BGS07a] ALEXANDER BOLOTOV, OLEG M. GRIGORIEV, VASILYI SHANGIN. Automated Natural Deduction for Propositional Linear-Time Temporal Logic. In: *14th International Symposium on Temporal Representation and Reasoning (TIME 2007), 28-30 June 2007, Alicante, Spain*, S. 47–58. IEEE Computer Society, 2007.
- [BGS07b] ALEXANDER BOLOTOV, OLEG M. GRIGORIEV, VASILYI SHANGIN. A Simpler Formulation of Natural Deduction Calculus for Linear-Time Temporal Logic. In: BHANU PRASAD,

- Hg., *Proceedings of the 3rd Indian International Conference on Artificial Intelligence, Pune, India, December 17-19, 2007*, S. 1253–1266. IICAI, 2007.
- [BK08] CHRISTEL BAIER, JOOST-PIETER KATOEN. *Principles of Model Checking*. Cambridge, MA: MIT Press, 2008.
- [Bor05] RICHARD BORNAT. *Proof and Disproof in Formal Logic: An introduction for programmers*. Oxford: Oxford University Press, 2005.
- [BR18] DANIEL LE BERRE, PASCAL RAPICAULT. Boolean-Based Dependency Management for the Eclipse Ecosystem. *International Journal on Artificial Intelligence Tools*, 27(1):1–23, 2018.
- [BV11] MARCO BOZZANO, ADOLFO VILLAFIORITA. *Design and Safety Assessment of Critical Systems*. Boca Raton, FL: Auerbach Publications, 2011.
- [CGP99] EDMUND M. CLARKE, JR., ORNA GRUMBERG, DORON A. PELED. *Model Checking*. Cambridge, MA: MIT Press, 1999.
- [DLL62] MARTIN DAVIS, GEORGE LOGEMANN, DONALD LOVELAND. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [dMB10] LEONARDO MENDONÇA DE MOURA, NIKOLAJ BJØRNER. Applications and Challenges in Satisfiability Modulo Theories. In: *Second International Workshop on Invariant Generation, WING 2009, York, UK, March 29, 2009 and Third International Workshop on Invariant Generation, WING 2010, Edinburgh, UK, July 21, 2010, EPiC Series in Computing*, Band 1, S. 1–11. 2010.
- [DP60] MARTIN DAVIS, HILARY PUTNAM. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.
- [GBE⁺14] JÜRGEN GIESL, MARC BROCKSCHMIDT, FABIAN EMMES, ET AL. Proving Termination of Programs Automatically with AProVE. In: *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings, Lecture Notes in Computer Science*, Band 8562, S. 184–191. Springer, 2014.
- [Gen35] GERHARD GENTZEN. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift*, 39:176–210, 1935. URL <http://gdz.sub.uni-goettingen.de/dms/resolveppn/?PPN=GDZPPN002375508>.

- [GH02] P. R. GLÜCK, G. J. HOLZMANN. Using SPIN model checking for flight software verification. In: *Proceedings, IEEE Aerospace Conference*, Band 1, S. 1–1. 2002.
- [GL02] DIMITRA GIANNAKOPOULOU, FLAVIO LERDA. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In: DORON A. PELED, MOSHE Y. VARDI, Hg., *Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11-14, 2002, Proceedings, Lecture Notes in Computer Science*, Band 2529, S. 308–326. Springer, 2002.
- [GPSS80] DOV M. GABBAY, AMIR PNUEL, SAHARON SHELAH, ET AL. On the Temporal Analysis of Fairness. In: PAUL W. ABRAHAMS, RICHARD J. LIPTON, STEPHEN R. BOURNE, Hg., *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, S. 163–173. ACM Press, 1980.
- [GPVW95] ROB GERTH, DORON A. PELED, MOSHE Y. VARDI, ET AL. Simple on-the-fly automatic verification of linear temporal logic. In: PIOTR DEMBINSKI, MAREK SREDNIAWA, Hg., *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995, IFIP Conference Proceedings*, Band 38, S. 3–18. Chapman & Hall, 1995.
- [Gö29] KURT GÖDEL. *Über die Vollständigkeit des Logikkalküls*. Dissertation, 1929.
- [Hal11] ANTHONY HALL. $E = mc^2$ Explained. In: BASHAR NUSEIBEH, PAMELA ZAVE, Hg., *Software Requirements and Design: the Work of Michael Jackson*. lulu.com, 2011.
- [Hal15] VOLKER HALBACH. *The Logic Manual*. Oxford: Oxford University Press, 2015.
- [Hed04] SHAWN HEDMAN. *A First Course in Logic: An Introduction to Model Theory, Proof Theory, Computability, and Complexity*. Oxford: Oxford University Press, 2004.
- [HL11] MARTIN HOFMANN, MARTIN LANGE. *Automatentheorie und Logik*. Heidelberg: Springer, 2011.
- [HLP01] KLAUS HAVELUND, MICHAEL R. LOWRY, JOHN PENIX. Formal Analysis of a Space-Craft Controller Using SPIN. *IEEE Trans. Software Eng.*, 27(8):749–765, 2001.

- [Hof11] DIRK W. HOFFMANN. *Grenzen der Mathematik: Eine Reise durch die Kerngebiete der mathematischen Logik*. Heidelberg: Spektrum Akademischer Verlag, 2011.
- [Hol04] GERARD J. HOLZMANN. *The Spin Model Checker: Primer and Reference Manual*. Boston, MA: Addison-Wesley, 2004.
- [HR04] MICHAEL HUTH, MARK RYAN. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge, UK: Cambridge University Press, 2. Auflage, 2004.
- [HS00] GERARD J. HOLZMANN, MARGARET H. SMITH. Automating software feature verification. *Bell Labs Tech. J.*, 5(2):72–87, 2000.
- [Jac01] MICHAEL JACKSON. *Problem Frames: Analysing and Structuring Software Development Problems*. Harlow, England: Addison-Wesley, 2001.
- [Jac02] MICHAEL JACKSON. Where, Exactly, Is Software Development? In: BERNHARD K. AICHERNIG, T. S. E. MAIBAUM, Hg., *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers, Lecture Notes in Computer Science*, Band 2757, S. 115–131. Springer, 2002.
- [Jac06] MICHAEL JACKSON. The role of structure: a software engineering perspective. In: DENIS BESNARD, CRISTINA GACEK, CLIFFORD B. JONES, Hg., *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, Kapitel 2, S. 16–45. Springer, 2006.
- [Jac12] DANIEL JACKSON. *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA: MIT Press, revised Auflage, 2012.
- [Kar96] PIM KARS. The application of Promela and Spin in the BOS project. In: JEAN-CHARLES GRÉGOIRE, GERARD J. HOLZMANN, DORON A. PELED, Hg., *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Band 32, S. 51–63. DIMACS/AMS, 1996.
- [KCH⁺90] KYO C. KANG, SHOLOM G. COHEN, JAMES A. HESS, ET AL. *Technical Report: Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Pittsburgh, PA: Software Engineering Institute (SEI), 1990.

- [KGN⁺09] ROOPE KAIVOLA, RAJNISH GHUGHAL, NAREN NARASIMHAN, ET AL. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, Lecture Notes in Computer Science*, Band 5643, S. 414–429. Springer, 2009.
- [Knu15] DONALD E. KNUTH. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Boston, MA: Addison-Wesley, 2015.
- [KS06] DANIEL KROENING, OFER STRICHMAN. *Decision Procedures: An Algorithmic Point of View*. Springer, 2006.
- [Lam02] LESLIE LAMPORT. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA: Addison-Wesley, 2002.
- [LDG08] BARRY LONG, JÜRGEN DINGEL, T. C. NICHOLAS GRAHAM. Experience applying the SPIN model checker to an industrial telecommunications system. In: WILHELM SCHÄFER, MATTHEW B. DWYER, VOLKER GRUHN, Hg., *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, S. 693–702. ACM, 2008.
- [Low95] GAVIN LOWE. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 55(3):131–133, 1995.
- [Mar02] DAVIDE MARCHIGNOLI. *Natural Deduction Systems for Temporal Logics*. Dissertation, Dipartimento di Informatica, Università di Pisa, 2002.
- [Mer01] S. MERZ. Model Checking: A Tutorial Overview. In: F. CASSEZ, C. JARD, B. ROZOY, ET AL., Hg., *Modeling and Verification of Parallel Processes*, Nummer 2067 in Lecture Notes in Computer Science, S. 3–38. Springer-Verlag, 2001.
- [MH07] ANDREAS METZGER, PATRICK HEYMANS. *Technischer Bericht: Comparing Feature Diagram Examples Found in the Research Literature*. Software Systems Engineering, Universität Duisburg-Essen, 2007.
- [MTS⁺17] JENS MEINICKE, THOMAS THÜM, REIMAR SCHRÖTER, ET AL. *Mastering Software Variability with FeatureIDE*. Cham: Springer, 2017.
- [NRZ⁺15] CHRIS NEWCOMBE, TIM RATH, FAN ZHANG, ET AL. How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.

- [NT01] SHIN NAKAJIMA, TETSUO TAMAI. Behavioural Analysis of the Enterprise JavaBeans™ Component Architecture. In: MATTHEW B. DWYER, Hg., *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings, Lecture Notes in Computer Science*, Band 2057, S. 163–182. Springer, 2001.
- [Rau08] WOLFGANG RAUTENBERG. *Einführung in die Mathematische Logik, 3., überarbeitete Auflage*. Wiesbaden: Vieweg + Teubner, 2008.
- [Sch96] BRUCE SCHNEIER. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*. Bonn: Addison-Wesley, 1996.
- [Sch00] UWE SCHÖNING. *Logik für Informatiker*. Heidelberg: Spektrum Akademischer Verlag, 5. Auflage, 2000.
- [Sim94] ALEX K. SIMPSON. *The proof theory and semantics of intuitionistic modal logic*. Dissertation, University of Edinburgh, UK, 1994.
- [Sip13] MICHAEL SIPSER. *Introduction to the Theory of Computation, Third Edition*. Boston, MA: Cengage Learning, 2013.
- [SKK03] CARSTEN SINZ, ANDREAS KAISER, WOLFGANG KÜCHLIN. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.
- [Tse83] G. S. TSEITIN. On the Complexity of Derivation in Propositional Calculus. In: J. SIEKMANN, G. WRIGHTSON, Hg., *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, S. 466–483. Berlin, Heidelberg: Springer, 1983.
- [van09] AXEL VAN LAMSWEERDE. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Chichester: John Wiley & Sons, 2009.
- [VBF⁺11] ERIC VERHULST, RAYMOND T. BOUTE, JOSÉ MIGUEL SAMPAIO FARIA, ET AL. *Formal development of a network-centric RTOS: software engineering for reliable embedded systems*. New York: Springer, 2011.
- [vD13] DIRK VAN DALEN. *Logic and Structure*. Berlin: Springer, 5. Auflage, 2013.
- [VdJM08] ERIC VERHULST, GJALT G. DE JONG, VITALIY MEZHUYEV. An Industrial Case: Pitfalls and Benefits of Applying Formal Methods to the Development of a Network-Centric

- RTOS. In: JORGE CUÉLLAR, T. S. E. MAIBAUM, KAISA SERE, Hg., *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings, Lecture Notes in Computer Science*, Band 5014, S. 411–418. Springer, 2008.
- [vL08] AXEL VAN LAMSWEERDE. Requirements engineering: from craft to discipline. In: MARY JEAN HARROLD, GAIL C. MURPHY, Hg., *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, S. 238–249. ACM, 2008.
- [WBBK11] MARKUS WEISSMANN, STEFAN BEDENK, CHRISTIAN BUCKL, ET AL. Model Checking Industrial Robot Systems. In: ALEX GROCE, MADANLAL MUSUVATHI, Hg., *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings, Lecture Notes in Computer Science*, Band 6823, S. 161–176. Springer, 2011.
- [Zav12] PAMELA ZAVE. Using lightweight modeling to understand chord. *ACM SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, 2012.