TECHNISCHE UNIVERSITÄT DRESDEN

Verifikation von DNA-basierten Programmen

Diplomarbeit

Technische Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik

vorgelegt von:
Matthias Hintzmann

Betreuerin:
Dr.-Ing. Monika Sturm

Verantwortlicher Hochschullehrer:
Prof. Dr.-Ing. Franz Baader

Dresden, im April 2006

Technische Universität Dresden
Institut für Theoretische Informatik
Lehrstuhl für Automatentheorie

17.10.2005

Aufgabenstellung für die Diplomarbeit

Bearbeiter: Matthias Hintzmann
Studiengang: Informatik
Matrikelnr.: 2804933

Thema: Verifikation von DNA-basierten Programmen

Zielstellung. In der Diplomarbeit sollen zwei Fachgebiete zusammengeführt werden: DNA-Computing als ein unkonventionelles Modell zur Berechenbarkeit sowie die Verifikation von Modellen und Programmen, die auf der Basis von DNA-Molekülen Berechnungen vollziehen. Dabei sollen folgende Punkte bei der Bearbeitung berücksichtigt werden:

- Vorstellung beider Fachgebiete und ihrer aktuellen Schwerpunkte
- DNA-basierte Implementierung eines parallelen Theorembeweisers
- DNA-basierte Implementierung des SAT-Problems
- Spezifikation des restriktiven DNA-Computing-Modells nach Adleman
- Verifikation des Lipton-Experimentes (zum Lösen des SAT-Problems) mithilfe von PVS
- Verifikation eines Inferenzalgorithmus mithilfe von PVS
- Bewertung der Ergebnisse

Eine selbständige Einarbeitung in das System PVS wird erwartet. Folgende Literatur wird empfohlen:

1. Th. Hinze, M. Sturm. *Rechnen mit DNA - Eine Einführung in Theorie und Praxis.* Oldenbourg Verlag München, 2004
2. C. Graciani Díaz, F.J. Martín Mateos, and Mario J. Pérez Jiménez. *Specification of Adleman's Restricted Model using an Automated Reasoning System: Verification of Lipton's Experiment.* In Proceedings of the Third International Conference, Unconventional Models of Computation, Japan, 2002
3. I.-H. Lee, J.-Y. Park, H.-M. Jang, Y.-G. Chai, B.-T. Zhang. *DNA Implementation of Theorem Proving with Resolution Refutation in Propositional Logic.* In Proceedings of the 8th International Workshop on DNA Based Computers, DNA8, Japan, 2002

4. C. Graciani Díaz, Mario J. Pérez Jiménez. *Using Automated Reasoning Systems on Molecular Computing.* In Proceedings of the 10th International Workshop on DNA Based Computers, DNA 10, Italy, 2004

5. G. Păun, G. Rozenberg, A. Salomaa. *DNA-Computing - New Paradigms.* Springer Verlag Berlin, Heidelberg, New York, 1998

Betreuerin:	Dr.-Ing. M. Sturm
Betreuender Hochschullehrer:	Prof. Dr.-Ing. F. Baader
Bearbeitungszeitraum:	17.10.2005 - 18.04.2006
Abschlussleistung:	Diplomarbeit, Verteidigung mit Vortrag

Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Dresden, den 17. April 2006

Danksagung

An dieser Stelle möchte ich mich ganz herzlich bei Frau Dr. Sturm für die interessante Themenstellung, die außerordentlich gute Betreuung und die unkomplizierte Zusammenarbeit bedanken. Die vielen spannenden Diskussionen und Gespräche haben mir sehr geholfen. Vielen Dank auch an Dr. Tews für die Einführung in PVS und die geduldige Beantwortung meiner Fragen.

Ganz besonders möchte ich mich bei meiner Familie und meinen Freunden für die stets geleistete moralische Unterstützung bedanken.

Inhaltsverzeichnis

1 Einführung

Kann man die Stoffe und Materialien, die in der Natur seit Jahrmillionen äußerst erfolgreich zur Speicherung von Informationen eingesetzt werden, auch für die künstliche Datenverarbeitung sinnvoll nutzen? Diese Frage umschreibt treffend die Grundidee des *DNA-Computing*, eines stark interdisziplinär geprägten Gebietes, das neben der Informatik und Molekularbiologie auch die Chemie, Mathematik und Physik mit einbezieht.

Doch wie ist es überhaupt möglich, mit Molekülen wie der DNA zu rechnen? Die DNA-Moleküle werden im DNA-Computing als Speichermedium verwendet. Über diverse molekularbiologische und biochemische Operationen werden die Moleküle dann geeignet verändert, um damit Rechenvorgänge zu realisieren. Eine Analyse der Moleküle nach Anwendung der Operationen dient abschließend als Ergebnisausgabe. Das Potenzial liegt hierbei in der extrem hohen Speicherdichte und -kapazität sowie in der Parallelität, da die Operationen idealerweise parallel auf alle DNA-Moleküle gleichzeitig einwirken. Probleme bei der experimentellen Umsetzung bestehen insbesondere durch schwer zu kontrollierende Seiteneffekte der DNA-Operationen.

Der Startschuss zur Entwicklung dieses noch jungen Forschungsgebietes fiel im Jahre 1994, als Leonard Adleman in [Adl94] erstmals experimentell DNA für die Lösung eines mathematischen Problems einsetzte. Es handelte sich hierbei um eine kleine Instanz des NP-vollständigen Hamiltonkreis-Problems, die er *in vitro*, d. h. außerhalb eines lebenden Organismus, sozusagen im Reagenzglas, lösen konnte. In Folge des berühmt gewordenen Adleman-Experiments wurden in der Theorie viele adäquate Berechnungsmodelle für das DNA-Computing entwickelt. Ein solches Modell verfügt über eine Menge Basisoperationen, die auf DNA-kodierte Daten einwirken. Die Operationen sind im Vergleich zu den biochemischen Prozessen stark abstrahiert und idealisiert definiert. Dies erlaubt die Konstruktion Algorithmen in diesen Modellen, auch als molekulare Programme bezeichnet.

Während in vielen Bereichen der Informatik die Verifikation, also die formale Überprüfung von spezifizierten Eigenschaften z.B. einer Implementation, erfolgreich eingesetzt wird, ist dies im Bereich des DNA-Computing bis auf [Día03, DMJ02] noch nicht der Fall. Jedoch ist gerade hier eine Verifikation der molekularen Programme besonders im Vorfeld ihrer kostenintensiven laborexperimentellen Umsetzung sinnvoll und nützlich. Die vorliegende Diplomarbeit soll einen Beitrag zur Etablierung von Verifikationstechniken im DNA-Computing leisten und die Nützlichkeit solcher Methoden aufzeigen. Exemplarisch wird eine DNA-basierte Implementation der aussagenlogischen Resolution auf die Eigenschaften Korrektheit und Vollständigkeit hin untersucht. Hierzu kommt das interaktive Verifikationssystem PVS zum Einsatz.

Die Arbeit gliedert sich wie folgt. Im Anschluss an dieses einführende Kapitel befindet sich im Kapitel 2 eine Betrachtung der biologischen und theoretischen Grundlagen des

DNA-Computing. Das Kapitel 3 präsentiert eine Beschreibung des Verifikationssystems PVS und seiner integrierten Spezifikationssprache. Darauf folgt im Kapitel 4 die Erläuterung eines DNA-basierten Lösungsansatzes für das Erfüllbarkeitsproblem und seiner Verifikation. In Kapitel 5 wird eine mögliche DNA-basierte Implementation für das aussagenlogische Resolutionsverfahren vorgestellt und als molekulares Programm formalisiert. Sodann wird im Kapitel 6 auf die Spezifikation der Operationen und des Programms in PVS sowie auf den geführten Nachweis der Korrektheit und Vollständigkeit eingegangen. Die Diplomarbeit schließt im Kapitel 7 mit einer Schlussbetrachtung und einem Ausblick. Im Anhang A befindet sich der Quelltext der entwickelten PVS Spezifikation.

2 DNA-Computing

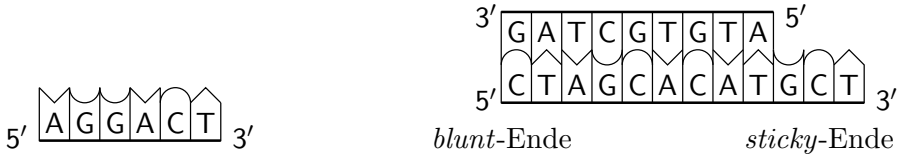
Das Grundprinzip einer Berechnung im Sinne des DNA-Computing lässt sich in drei Schritte unterteilen. Zuerst werden die für den Algorithmus zur Problemlösung benötigten Eingabedaten in geeigneter Weise als DNA-Stränge kodiert. Die benötigten DNA-Moleküle werden zumeist künstlich hergestellt. Im Normalfall befinden sich von jedem Strang mehrere Millionen Kopien im Reagenzglas, um eine ausreichende Redundanz zu gewährleisten. Dann wird eine Serie von molekularbiologischen Operationen auf die DNA-Moleküle angewendet. Dieser Schritt verkörpert den eigentlichen Berechnungsvorgang. Am Schluss steht als Ausgabeoperation die Analyse der durch die Operationen veränderten DNA-Stränge.

2.1 Aufbau und Struktur der DNA

Die Desoxyribonukleinsäure, meist nach der englischen Bezeichnung *deoxyribonucleic acid* mit DNA abgekürzt, ist ein Makromolekül und dient in der Vererbung bei zellulären Organismen als Träger der genetischen Information. Die Struktur der DNA wurde 1953 von James Watson und Francis Crick aufgeklärt, die dafür im Jahre 1962 gemeinsam mit Maurice Wilkins den Nobelpreis für Medizin erhielten.

Die DNA ist ein langes Polymer, das heißt, ein Kettenmolekül aus vielen Einzelbausteinen, die als Desoxyribonukleotide bezeichnet werden. Es gibt vier verschiedene Bausteine dieser Art. Jedes Nukleotid setzt sich aus dem Zucker Desoxyribose, einem Phosphorsäure-Molekül und einer heterozyklischen Nukleobase zusammen. Während die Desoxyribose- und Phosphorsäure-Untereinheiten bei jedem Nukleotid gleich sind, unterscheidet man vier verschiedene Basen: *Adenin*, *Thymin*, *Guanin* und *Cytosin*, üblicherweise entsprechend abgekürzt durch die Buchstaben A, T, G sowie C. Während Adenin und Guanin zu den Purinbasen gehören, zählt man Cytosin und Thymin zu den Pyrimidinbasen.

Die einzelnen Nukleotide des Kettenmoleküls sind über Phosphordiesterbindungen zu einem Strang verknüpft. Diese feste Bindung ist *kovalent*, also eine Atombindung, und bildet sich zwischen den Desoxyribosen zweier benachbarter Nukleotide aus. Die fünf Kohlenstoffatome einer Desoxyribose sind ringförmig angeordnet und mit $1'$ bis $5'$ durchnummeriert. Bei den in der DNA vorkommenden Nukleotiden befindet sich am $5'$-Ende der Desoxyribose ein Phosphatrest und am $3'$-Ende eine OH-Gruppe. Letztere reagiert bei der Verknüpfung zu einer Kette unter Wasserabspaltung mit der Phosphatgruppe des jeweils nächsten Nukleotids. Die Enden eines so gebildeten DNA-Einzelstrangs können in $5'$- und $3'$-Ende unterschieden werden. Ein DNA-Einzelstrang besitzt demnach eine natürliche Orientierung. Üblicherweise werden Nukleotidsequenzen in $5'$-$3'$-Richtung notiert.



(a) *Einzelstrang* (b) *Doppelstrang*

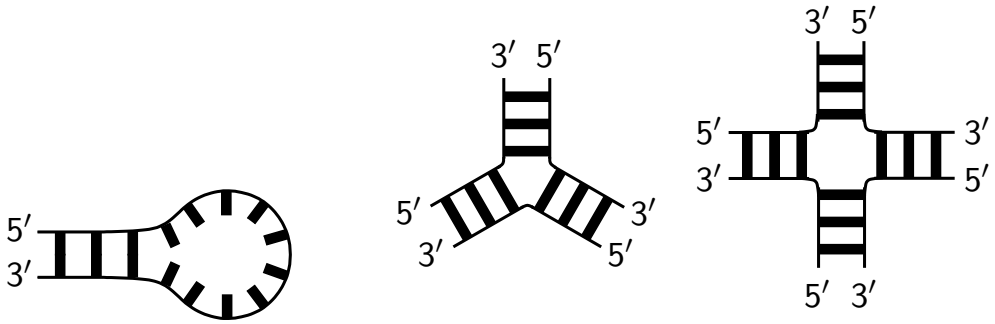
Abbildung 2.1: *Vereinfachte Darstellung von DNA-Strängen*

Zusätzlich zu den beschriebenen festen Bindungen mit ihren beiden Nachbarn kann jede Base auch noch über Wasserstoffbrücken unter bestimmten Voraussetzungen eine lockere Bindung mit einer weiteren Base ausbilden. Eine solche Verbindung kann von Adenin allerdings nur mit Thymin und von Guanin nur mit Cytosin eingegangen werden. Man spricht hier von spezifischer Basenpaarung. Während sich zwischen Adenin und Thymin zwei Wasserstoffbrücken ausbilden, sind es zwischen Guanin und Cytosin derer drei. Die Nukleotide A und T werden als komplementär zueinander bezeichnet, ebenso die Nukleotide C und G.

Über die Wasserstoffbrückenbindungen zwischen den einzelnen Basen können sich zwei DNA-Einzelstränge zu einem DNA-Doppelstrang zusammenlagern. Dem Prinzip der spezifischen Basenpaarung folgend, weisen beide Einzelstränge dabei eine entgegengesetzt komplementäre Basenabfolge auf. So verläuft der eine Strang in 5′-3′-Richtung, während der andere von 3′ nach 5′ ausgerichtet ist. Entgegengesetzt ausgerichtete Einzelstränge werden auch als *antiparallel* bezeichnet.

An den äußeren Enden eines DNA-Doppelstrangs können so genannte *Einzelstrangüberhänge*, das sind zumeist kurze, einzelsträngige Nukleotidsequenzen, auftreten. DNA-Doppelstrangenden mit Einzelstrangüberhang werden als *sticky* (klebrig) bezeichnet, da sie ein sehr effektives Zusammenfügen von DNA-Doppelsträngen erlauben. Ein DNA-Doppelstrangende ohne Einzelstrangüberhang heißt *blunt* (glatt), siehe Abbildung 2.1(b). Weiterhin kann ein Doppelstrang auch *Einzelstrangabschnitte* aufweisen. Die klassische Doppelhelixstruktur der DNA entsteht durch die Windung des DNA-Doppelstrangs um eine gemeinsame zentrale Achse.

Ein DNA-Doppelstrang muss nicht zwangsläufig genau zwei äußere Enden besitzen. So kann es vorkommen, dass sich ein Einzelstrang, dessen Enden zueinander kompatibel sind, schleifenförmig (man spricht hier auch von einer *hairpin-loop*-Struktur) zu einem Doppelstrang zusammensetzt, siehe Abbildung 2.2(a). Darüber hinaus ist, wie in der Abbildung 2.2(b) dargestellt, die Verbindung von mehr als zwei Einzelsträngen zu *verzweigten* Doppelstrangmolekülen möglich. DNA-Stränge, die nicht genau zwei Strangenden aufweisen, werden als *nichtlinear* bezeichnet und entstehen entweder unbeabsichtigt, wenn die Basenpaarung nicht optimal verläuft, oder sie werden, trotz der weitaus schwierigeren Handhabung im Labor, ganz gezielt zur Kodierung bestimmter Daten im DNA-Computing verwendet, siehe beispielsweise Kapitel 5 sowie [LPJ+03] und [UHK02].



(a) *Hairpin* (b) *Verzweigte Doppelstränge*

Abbildung 2.2: *Beispiele nichtlinearer DNA-Doppelstrangmoleküle*

2.2 Molekularbiologische Operationen auf DNA-Molekülen

In der Molekularbiologie wurden viele verschiedene Labortechniken entwickelt, die eine Manipulation von DNA-Molekülen ermöglichen. Einige dieser molekularbiologischen Prozesse, beziehungsweise Kombinationen derselben, stellen in Abstraktion die Basisoperationen in den unterschiedlichen Modellen des DNA-Computing dar. Die meisten der DNA-Operationen werden in vitro in einem Reagenzglas (auch als *Tube* bezeichnet) ausgeführt, das die DNA-Moleküle in wässriger Lösung enthält. In diesem Abschnitt sollen die relevanten Operationen kurz und übersichtlich beschrieben werden. Für eine detailliertere Darstellung der Labortechniken sowie ihrer Prozessparameter und Seiteneffektanfälligkeit sei auf [HS04] verwiesen.

Bei einigen Operationen handelt es sich um enzymatische Reaktionen. Enzyme sind Proteine, die chemische Reaktionen katalysieren. Sie wirken sehr spezifisch und beschleunigen den Reaktionsverlauf millionenfach, indem sie die Aktivierungsenergie herabsetzen, die für eine Stoffumsetzung überwunden werden muss. Ohne Enzyme wären viele chemische Reaktionen in Zellen schlichtweg zu langsam, um das Leben zu ermöglichen. Als Katalysator liegt das Enzym nach der Reaktion wieder in seiner Ausgangsform vor.

Für die im Folgenden angegebenen Operationen existieren bedingt durch ihre geschichtliche Entwicklung oft unterschiedliche Bezeichnungen. Während wir uns hier an die international üblichen Begriffe halten wollen, sind die wichtigsten synonymen Bezeichnungen ebenfalls angegeben.

Synthesis: Mit *Synthesis* (auch *Einzelstrangsynthese*, *Oligonukleotidsynthese*) wird die künstliche Erzeugung von DNA-Einzelsträngen frei wählbarer Nukleotidabfolge bezeichnet. Die resultierende Exemplaranzahl kann in ihrer Größenordnung vorgegeben werden. Die Synthese erfolgt in 3′-5′-Richtung, wobei nach dem Prinzip der wachsenden Kette vorgegangen wird. Fixiert an einen Glasträger wird jeder DNA-Einzelstrang zyklisch Nukleotid für Nukleotid aufgebaut.

Hybridisierung: Das Zusammenlagern von mindestens zwei antiparallelen und komplementären DNA-Einzelsträngen zu DNA-Doppelsträngen nennt man *Hybridisierung* (auch *Annealing*, *Erstarren*). Die Verbindung entsteht über Wasserstoffbrücken nach dem Prinzip der spezifischen Basenpaarung unter Ausbildung aller Anlagerungsmöglichkeiten. Die Bildung der Wasserstoffbrückenbindungen sind hauptsächlich abhängig von der Temperatur. Die Hybridisierung charakterisiert sich daher laborpraktisch als ein langsam verlaufender Abkühlungsprozess.

Denaturierung: Unter *Denaturierung* (auch *Melting*) wird die Aufspaltung von DNA-Doppelsträngen in die zugrunde liegenden DNA-Einzelstränge verstanden. Es ist somit die Umkehrung des molekularbiologischen Ablaufs bei der Hybridisierung.

Merging: Das Zusammenfügen zweier oder mehrerer Tubeinhalte in ein einzelnes Tube wird als *Merging* (auch *Union*) bezeichnet.

Ligation: Die Operation *Ligation* bewirkt eine fortgesetzte Verkettung von DNA-Doppelsträngen an deren Enden. Die DNA-Doppelstrangenden, die sowohl sticky als auch blunt sein können, müssen kompatibel also passgenau sein, und mindestens an einem $5'$-Ende eine Phosphatgruppe besitzen. Es werden dabei die kovalenten Bindungen zwischen den benachbarten Nukleotiden ausgebildet. Ligation ist eine biochemische Reaktion, die durch das Enzym DNA-Ligase katalysiert wird.

Digestion: Restriktionsenzyme ermöglichen das gezielte Zerschneiden von DNA-Doppelsträngen. Diese Reaktion wird mit dem Begriff *Digestion* (auch *Cut*, *Schnitt*, *Restriktionsspaltung*) bezeichnet. Die für das DNA-Computing besonders interessanten Vertreter der Restriktionsenzyme sind die Restriktionsendonukleasen. Sie lagern sich an einer durch das jeweilige Enzym festgelegten kurzen DNA-*Erkennungssequenz* an den DNA-Doppelstrang an und spalten ihn an definierten Spaltstellen. Hierbei können auch Einzelstrangüberhänge entstehen. Die Erkennungssequenzen bestehen zumeist aus vier bis acht Nukleotiden.

Die Gruppe der Exonukleasen ermöglicht das Verkürzen von DNA-Strängen durch Abspaltung von einem Nukleotid pro Reaktionszyklus am Ende des Strangs beginnend. Einige Exonukleasen arbeiten richtungsspezifisch, andere degradieren dagegen entweder Doppel- oder Einzelstränge.

Labeling: Mit *Labeling* wird eine Reaktion bezeichnet, bei der über Enzyme eine Strangenendenmarkierung, das heißt eine geeignete chemische Gruppe oder ein Molekül, an bestimmten Enden von DNA-Strängen entweder angelagert oder abgebaut wird. Über das entsprechende Enzym wird festgelegt, ob an $5'$-Enden oder an $3'$-Enden operiert wird und ob ein Anfügen oder Entfernen einer Strangendenmarkierung vorgenommen werden soll.

Polymerisation: Die sticky-Enden eines DNA-Doppelstrangs können mit der Operation *Polymerisation* zu blunt-Enden konvertiert werden. Als Enzym wirkt hier die DNA-Polymerase. Konkret werden vorkommende $5'$-Einzelstrangüberhänge zu

blunt-Enden aufgefüllt und 3′-Überhänge zu blunt-Enden abgebaut. DNA-Einzelstränge werden vollständig abgebaut.

Polymerase-Kettenreaktion: Die *Polymerase-Kettenreaktion* (*Polymerase Chain Reaction*, *PCR*, *Amplifizieren*) ermöglicht das exponentielle Vervielfältigen von DNA-Strangabschnitten. Das Anfangs- und Endstück des zu kopierenden Strangabschnitts muss bekannt sein und in Form von kurzen DNA-Einzelsträngen, so genannten *Primern*, vorliegen. Während typische Primer eine Länge von 20 Nukleotiden haben, kann die zu kopierende Sequenz mehrere tausend Nukleotide umfassen.

Die Polymerase-Kettenreaktion setzt sich aus der wiederholten zyklischen Anwendung der folgenden drei Operationen zusammen (siehe auch Abbildung 2.3):

- Denaturierung: Das Denaturieren der DNA-Doppelstränge in ihre komplementären Einzelstränge.
- Hybridisierung: Die Anlagerung der Primer an die entsprechend komplementären Stellen der zuvor erzeugten DNA-Einzelstränge.
- Polymerisation: Die Primerverlängerung und damit Auffüllung der Einzelstrangabschnitte zu kompletten DNA-Doppelsträngen.

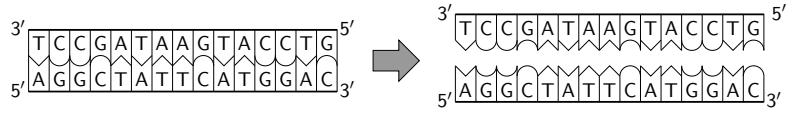
Die neu entstandenen DNA-Stränge bilden wiederum die Vorlage für den nächsten Zyklus, so dass sich die Menge an Duplikaten mit jedem Durchlauf verdoppelt.

Avidin-Biotin-Separation Mit der Operation *Avidin-Biotin-Separation* (auch *Affinity Purification*) ist es möglich, speziell mit einem Biotinmolekül markierte DNA-Stränge räumlich von nichtmarkierten DNA-Strängen zu trennen.

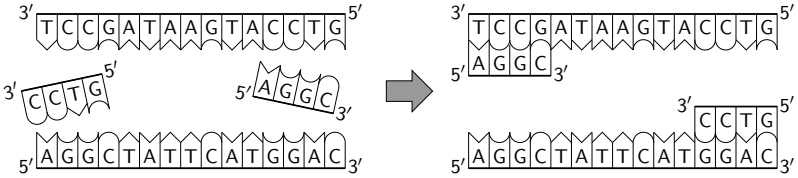
Die Avidin-Biotin-Separation ist darüber hinaus Hauptbestandteil einer Operationsfolge, die eine *Separation nach Subsequenz* gestattet, wobei jene DNA-Stränge selektiert und extrahiert werden, die eine vorgegebene beliebige DNA-Subsequenz enthalten.

Gel-Elektrophorese: Die *Gel-Elektrophorese* (auch *Gel Electrophoresis*) ermöglicht die Längenseperation und -bestimmung von DNA-Strängen und dient im DNA-Computing unter anderem als Ausgabeoperation. DNA-Moleküle sind negativ geladen und wandern im elektrischen Feld innerhalb eines geeigneten Gels als Trägermedium in Richtung der positiven Elektrode, wobei die Wanderungsgeschwindigkeit abhängig von der Stranglänge ist. Die DNA-Stränge können so entsprechend ihrer Nukleotidanzahl räumlich voneinander getrennt werden.

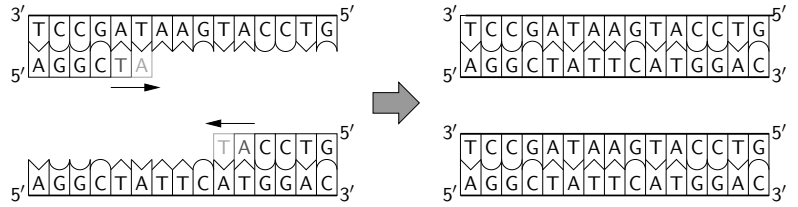
Sequenzieren: Die Bestimmung der exakten Nukleotidabfolge in einem DNA-Molekül wird als *Sequenzieren* bezeichnet. Die heute am häufigsten verwendete Methode nach Sanger ist eine Kombination aus Polymerisation und hochauflösender Polyacrylamidgel-Elektrophorese.



(a) *Denaturierung der Doppelstränge*



(b) *Primeranlagerung per Hybridisierung*



(c) *Verlängerung der Primer durch Polymerisation*

Abbildung 2.3: *Prinzip der Polymerase Kettenreaktion*

2.3 Modelle des DNA-Computing

Trotz der relativ kurzen Geschichte haben sich im Bereich des DNA-Computing eine Vielzahl diverser Modelle entwickelt. DNA-Computing-Modelle dienen der mathematischen und formalen Beschreibung der ablaufenden Berechnungsprozesse und ermöglichen so eine theoretische Betrachtung ihrer Eigenschaften sowie eine Klassifizierung nach bestimmten Kriterien wie beispielsweise ihrer Berechnungsstärke [HS04]. Sie definieren die Struktur der Daten und die Menge der auf sie einwirkenden Operationen. Die Modelloperationen sind im Vergleich zu den molekularbiologischen Prozessen auf DNA-Molekülen idealisiert und abstrakt dargestellt. Dies gestattet die Definition von Algorithmen in diesen Modellen, die unter der geeigneten Verwendung der Basisoperationen bestimmte Problemstellungen lösen.

Vergleicht man Rechenvorgänge im DNA-Computing mit denen im konventionellen Rechnen, so gibt es viele Unterschiede. So werden beispielsweise die Operanden einer biologischen Operation durch diese zumeist „verbraucht“, stehen also nach der Anwen-

dung der Operation nicht mehr in ihrer Ausgangsform zur Verfügung.

In den Modellen des DNA-Computing wird als Basisdatentyp zumeist das *Tube* verwendet, in Abstraktion zu den in den Laboren verwendeten Reaktionsgefäßen. Ein DNA-Strang wird oft als Wort einer formalen Sprache über einem Alphabet Σ mit $\Sigma = \{A, T, C, G\}$ angesehen. Berechnungsmodelle, die auf der Theorie der formalen Sprachen basieren, sind somit ein möglicher und naheliegender Ansatz, Berechnungsvorgänge im DNA-Computing geeignet zu modellieren.

Viele der entwickelten Berechnungsmodellen sind universell, sie besitzen also die gleiche Berechnungsstärke wie die Turingmaschine [HS04]. Forschungsziel ist die Entwicklung eines Modells, das neben der Universalität ausschließlich über Operationen verfügt, die sich möglichst gut auf vorhandene molekularbiologische Operationen abbilden lassen. In den folgenden beiden Abschnitten wollen wir zwei bekannte Modelle des DNA-Computing, Splicing-Systeme sowie Insertion-Deletion-Systeme, kurz vorstellen. Anschließend wird das für die späteren Betrachtungen wichtige Filtering Modell beschrieben. Für weitere Modelle sei auf die Literatur verwiesen [HS04].

2.3.1 Splicing-Systeme

Splicing-Systeme sind Berechnungsmodelle, die auf der Theorie der formalen Sprachen basieren. Grundbestandteil ist die *Splicing-Operation*, die in Abstraktion zur DNA-Rekombination definiert ist. DNA-Rekombination bezeichnet einen natürlichen Vorgang, bei dem DNA-Stränge an bestimmten Stellen mithilfe von Restriktionsenzymen zerschnitten und anschließend per Ligation in neuer Kombination wieder zusammengefügt werden. Die Splicing-Operation oder Splicing-Regel stellt diesen Vorgang abstrakt als gezielte Wortaufspaltung an vordefinierten Schnittstellen und die anschließende Verkettung der erzeugten Präfixe und Suffixe dar. Neben einer Menge von Splicing-Regeln ist in einem Splicing-System auch eine Menge von vorgegebenen Wörtern, den Axiomen, enthalten. Die iterative Anwendung der Splicing-Regeln auf die Axiome bzw. Wörter, die in vorhergehenden Splicing-Schritten entstanden sind, führt zur Ableitung neuer Wörter. Es gibt viele Ausprägungen und Varianten von Splicing-Systemen, z.B. H-, EH-Systeme, verteilte Splicing-Systeme. Ein Vertreter der verteilten Splicing-Systeme ist das an der TU Dresden entwickelte Mehrtubesystem TT6 [HS04].

2.3.2 Insertion-Deletion-Systeme

Ein in Zellen natürlich ablaufender Vorgang ist das Herausschneiden sowie Einsetzen von DNA-Segmenten aus bzw. in DNA-Stränge. Modelliert wird dies innerhalb von Insertion-Deletion-Systemen durch eine kontextsensitive Teilwortentfernung und -einsetzung in Wörter einer formalen Sprache. Insertion-Deletion-Systeme repräsentieren somit einen Mechanismus zur Generierung formaler Sprachen. Sie erreichen die Berechnungsstärke der Turingmaschine [HS04]. Ein wichtiger positiver Aspekt ist die Nähe zu existierenden und wohluntersuchten biologischen Operationen. Insertion-Deletion-Systeme werden auch zur Modellierung von natürlich ablaufenden Prozessen wie dem Gene-Assembly in Zellen angewendet.

2.3.3 Das Filtering Modell

Das *Filtering Modell* nach Adleman und Lipton in seinen verschiedenen Ausführungen entstand unmittelbar nach dem berühmten Adleman-Experiment [Adl94]. In diesem Experiment demonstrierte Leonard Adleman eine im Labor realisierte, DNA-basierte und effiziente Lösung für das NP-vollständige Hamiltonkreis-Problem auf einem gerichteten Graph mit sieben Knoten. Da es sich um die erste praktische Implementierung im Bereich des DNA-Computing handelte und darüber hinaus erfolgreich verlief, hatte sie eine herausragende Bedeutung für die Entwicklung der theoretischen sowie experimentellen Forschung in diesem Fachgebiet.

Die Filtering-Modelle sind im allgemeinen zur Lösung von Entscheidungsproblemen konzipiert. Als Grundprinzip der Lösungsstrategie liegt diesen Modellen eine vollständige Suche im DNA-kodierten Suchraum zugrunde. Das Lösungsverfahren lässt sich in zwei Phasen einteilen:

1. Generiere alle Lösungsmöglichkeiten für das Problem und kodiere sie entsprechend mithilfe von DNA-Strängen

2. Filtere, möglicherweise in mehreren Schritten, alle ungültigen Lösungen heraus

Im Anschluss an den Filterungsprozess ist es für die Entscheidungsfindung nur noch erforderlich zu prüfen, ob sich in der Ergebnismenge DNA-Stränge befinden oder nicht. Ein Beispiel, wie mithilfe des Filtering-Modells das NP-vollständige Erfüllbarkeitsproblem der Aussagenlogik zeiteffizient gelöst werden kann, wird im Kapitel 4 erläutert.

Das hier betrachtete *restriktive Filtering-Modell* nach Adleman [Adl96] verwendet als Elemente des Alphabets nicht explizit die Grundbausteine der DNA, also die Nukleotide A, T, G und C, sondern ganz allgemein beliebige Symbole. Dadurch ist das Modell nicht auf die Verwendung von DNA-Molekülen als Datenträger festgelegt. Als Konsequenz kann aber eine für DNA-Moleküle typische natürliche Orientierung nicht vorausgesetzt werden, so dass die Elemente eines Tubes nicht Sequenzen, sondern Multimengen von Symbolen des Alphabets darstellen. Die Definitionen erfolgen nach [Adl96].

Definition 2.1. *Ein* Aggregat *über einem Alphabet Σ ist eine endliche Multimenge von Symbolen aus Σ. Ein* Tube *ist eine Multimenge von Aggregaten über Σ.*

Definition 2.2. *Das Modell umfasst folgende Basisoperationen:*

- Separate. *Sei T ein Tube und $s \in \Sigma$ ein Symbol. Die Operation* `separate(T,s)` *erzeugt zwei Tubes, $+(T,s)$ sowie $-(T,s)$, wobei $+(T,s)$ diejenigen Aggregate aus T enthält, die das Symbol s beinhalten und $-(T,s)$ all jene Aggregate aus T enthält, die das Symbol s nicht beinhalten. Es gilt:*

$$T = +(T,s) \cup -(T,s) \qquad \textit{wobei}$$

$$+(T,s) = \{a \in T \mid s \in a\} \quad \textit{und} \quad -(T,s) = \{a \in T \mid s \notin a\}$$

- Merge. *Seien T_1 und T_2 Tubes. Die Operation* `merge(`T_1`,`T_2`)` *erzeugt ein Tube* $\cup(T_1, T_2)$*, welches die Vereinigung von T_1 und T_2 darstellt.*

$$\texttt{merge}(T_1, T_2) = T_1 \cup T_2$$

- Detect. *Sei T ein Tube. Die Operation* `detect(`T`)` *gibt* „ja“ *aus, wenn T mindestens ein Aggregat enthält, und sie* „nein“ *aus, falls T leer ist.*

$$\texttt{detect}(T) = \begin{cases} \text{nein} & \textit{falls } T = \emptyset \\ \text{ja} & \textit{sonst} \end{cases}$$

Das hier vorgestellte Modell ist *restriktiv*, das heißt die Eingangsoperanden stehen nach der Anwendung einer der Modelloperationen in ihrer Ausgangsform nicht mehr zur Verfügung. Ursprünglich enthielt das Modell noch eine zusätzliche Operation namens *Amplify*, die eine exakte Duplizierung von Tubeinhalten ermöglichte. Mit dieser zusätzlichen Operation würde es sich um ein nichtrestriktives Modell handeln, da man die Operanden vor Anwendung einer restriktiven Operation einfach kopieren könnte. Nichrestriktivität ist eine erstrebenswerte Eigenschaft für die Modelle des DNA-Computing. Da es jedoch keine echte laborpraktische Entsprechung für die Amplify-Operation gibt – die Polymerase-Kettenreaktion beispielsweise ist nur für die Kopierung einiger weniger DNA-Strangabschnitte geeignet – wurde die Operation aus dem Modell entfernt.

2.4 Hoffnungen, Probleme und Perspektiven

Die Hoffnungen, die in das DNA-Computing als ein Berechnungskonzept der Zukunft gesetzt werden, begründen sich auf einige vielversprechende Eigenschaften der DNA als Datenträger. So würde ein Liter wässriger Lösung mit ca. sechs Gramm DNA-Material eine theoretische Speicherkapazität von 10^{21} Basenpaaren aufweisen, was einer Speicherdichte von rund 1 bit/nm^3 entspricht. Die vorgestellten molekularbiologischen Operationen wirken idealerweise auf die Gesamtheit der DNA-Stränge gleichzeitig, wodurch eine massive Parallelität erreicht wird, die die augenscheinlich hohe Zeitintensität einiger labortechnischer Operationen von bis zu mehreren Stunden im Endeffekt aufwiegt. Weitere vorteilhafte Eigenschaften sind der geringe Energieverbrauch, die von der Natur bereitgestellte spezifische Basenpaarung und auch die vollständige Recyclingfähigkeit. Außerdem handelt es sich bei der DNA um ein sehr stabiles Molekül, das lange konservierbar ist und sich generell gut für eine in vitro Bearbeitung eignet [HS04].

Bedingt durch die hohe Parallelität in der Abarbeitung und den verfügbaren Speicherplatz, eignet sich das DNA-Computing besonders für solche Probleme, deren Lösung bei sequentieller Abarbeitung einen exponentiellen Zeitaufwand in Abhängigkeit von der Problemgröße notwendig macht. So konnte von einigen NP-vollständigen Problemen gezeigt werden, dass ihre DNA-basierte Problemlösung nur noch eine lineare Zeitkomplexität aufweist, mit entsprechendem exponentiellen Speicherplatzbedarf.

Bisher konnten experimentell jedoch zumeist nur kleine Problemgrößen bearbeitet werden. Dies liegt besonders an unerwünschten Wirkungen, den so genannten Seiteneffekten, der DNA-Operationen. Hier hofft man auf Fortschritte in der Molekularbiologie und eine Weiterentwicklung der Labortechniken zur Verringerung der Fehlerraten. Der exponentielle Speicherplatzbedarf setzt der Erhöhung der Problemgröße jedoch ebenfalls Grenzen: Berechnungen haben gezeigt, dass die von Adleman angewendete Lösungsmethode für das Hamiltonkreis-Problem bei einem Graphen mit 70 Knoten insgesamt 10^{25} kg DNA benötigen würde [Mal98]. Mit Änderungen im Lösungsverfahren könnten jedoch theoretisch auch noch größere Instanzen gelöst werden.

Oft wird die derzeitige Situation des DNA-Computing mit dem dem Entwicklungsbeginn der konventionellen Rechentechnik verglichen: die anfänglichen Prototypen waren damals alles andere als leistungsstark und fehlerresistent. So ist bis zu einer verbreiteten und wirklich nutzbringenden Anwendung des DNA-Computing noch ein weiter Weg zurückzulegen. Viele Forscher vertreten die Ansicht, dass ein möglicher DNA-Computer zukünftig eher eine Ergänzung denn ein Ersatz der derzeit verwendeten Rechentechnik darstellen wird [DK],[Mal98].

3 Das Verifikationssystem PVS

PVS (*Prototype Verification System*) ist ein interaktives Verifikationssystem zur Entwicklung und Analyse von formalen Spezifikationen. Es wurde am Stanford Research Institute (SRI) entwickelt und wird sowohl im akademischen als auch im industriellen Bereich eingesetzt. PVS besteht aus einer formalen Spezifikationssprache, einem Parser, einem Typüberprüfer, einem Beweisermodul und Dokumentationswerkzeugen. PVS ist momentan für die Betriebssysteme Solaris und Linux verfügbar und kann für den nichtkommerziellen Gebrauch kostenfrei installiert werden[1]. Der Benutzer interagiert mit dem System über eine Erweiterung des verbreiteten GNU Emacs bzw. XEmacs Editors. Die Arbeit mit PVS lässt sich grob in drei unterschiedliche Phasen unterteilen:

- Erstellen der Spezifikation
  - Modellierung
  - Aufstellen der Lemmata, Theoreme, etc.
- Typüberprüfung
- interaktives Beweisen der aufgestellten Lemmata, Theoreme, etc.

Es sind verschiedene Dokumentationen zu PVS vorhanden, sehr detaillierten Einblick bieten die offiziellen Dokumentationen [OSRSC01a], [OSRSC01b] und [OSRSC01c], als Einstieg sind besonders [COR+95] sowie [vHP06] zu empfehlen. Bei PVS handelt es sich um ein überaus komplexes System, so dass die folgenden Abschnitte nur einen kleinen Einblick geben können. Für weitere Informationen möchten wir den Leser deshalb auf die angegebene Literatur verweisen.

3.1 Die Spezifikationssprache

Die Spezifikationen in PVS [OSRSC01a] sind syntaktisch in so genannte Theorien untergliedert, die sich auf mehrere Dateien aufteilen können. Theorien enthalten zusammengehörige Definitionen oder Deklarationen von Typen, Konstanten, Variablen, Formeln, Theoremen usw.

Theorien können einander importieren und bezüglich Typen und Werten parametrisiert werden. In 1 ist die allgemeine Form einer Theorie angegeben.

[1] Projektadresse: `http://pvs.csl.sri.com`

1

```
<Name> [<Parameter>] : THEORY
BEGIN
  ASSUMING
    <Voraussetzungen>
  ENDASSUMING
  <Deklarationen, Definitionen, Formeln, ...>
END <Name>
```

Während für sämtliche Spezifikationen in PVS die Dateiendung `.pvs` verwendet wird, sind die mit den Spezifikationen verknüpften Beweise in Dateien mit der Endung `.prf` abgelegt. Das Verzeichnis, welches alle zu einer Spezifikation gehörenden Theorien und Beweise enthält, bildet den so genannten *Kontext*.

Basierend auf einer streng getypten Logik höherer Ordnung, stellt PVS neben den Basistypen wie `real`, `integer`, `nat`, `rationals`, `number`, `bool` ... auch die Möglichkeit der Deklaration von uninterpretierten und abstrakten Datentypen sowie mit `enum` Aufzählungstypen bereit. Die vom System vordeklarierten Typen mit vielen vordefinierten Eigenschaften befinden sich in den Theorien der Datei `prelude.pvs`, die immer automatisch importiert wird und damit dem Benutzer unmittelbar zur Verfügung steht. Es sind viele Operatoren und Relationen für die Basistypen definiert, die in der üblichen Infixnotation verwendet werden. Über Typkonstruktoren können

- Funktionstypen `[`$\mathtt{t}_1, \ldots, \mathtt{t}_n$ `-> t]`,
- Tupel- oder Produkttypen `[`$\mathtt{t}_1, \ldots, \mathtt{t}_n$`]` sowie
- Strukturtypen `[#` $\mathtt{a}_1$`:`$\mathtt{t}_1, \ldots, \mathtt{a}_n$`:`$\mathtt{t}_n$ `#]`

deklariert werden. Darüber hinaus besteht die Möglichkeit Teiltypen und abhängige Typen zu bilden. Ein prädikativer Teiltyp in PVS ist ein Typ, der diejenigen Elemente eines bereits existierenden Typs beinhaltet, die ein bestimmtes Prädikat erfüllen. Ein Beispiel ist der Typ der reellen Zahlen ohne Null, der dann für die Definition des Divisionsoperators genutzt werden kann (siehe 2).

2

```
  nonzero_real: NONEMPTY_TYPE = {r: real | r /= 0}

  /: [real, nonzero_real -> real]
```

Wird der Divisionsoperator nun in einer Spezifikation verwendet, erkennt die Typüberprüfung, dass der Nenner ungleich Null sein muss und generiert eine TCC (type correctness condition – Typkorrektheitsbedingung), in welcher der Benutzer mithilfe des Beweisers zeigen muss, dass der verwendete Nenner wirklich ungleich Null ist. Die generierten TCC's sind obligatorisch, das heißt bevor eine Theorie als vollständig verifiziert angesehen werden kann, müssen alle TCC's bewiesen werden. Die meisten der im Laufe einer Spezifikation auftretenden TCC's können jedoch vom System automatisch ohne Eingreifen des Benutzers bewiesen werden.

Variablen und Konstanten können in PVS, wie beispielhaft in 3 angegeben, deklariert werden.

3

```
x, y: VAR nat   % Variablendeklaration
c: nat = 5      % Konstantendeklaration
```

Bei den Variablen handelt es sich um *logische* Variablen, die nachfolgende Spezifikationskonstrukte vereinfachen können. Lokale Variablendeklarationen verdecken gleichnamige globale Variablendeklarationen. Die Variablendeklarationen einer Theorie können nicht exportiert werden.

Mengen eines Typs `T` werden – ebenso wie Prädikate – als Funktion `[T -> bool]` modelliert. Entsprechende Definitionen und Eigenschaften sind in der Theorie `sets` in `prelude.pvs` bereits vordefiniert.

Für boolesche Ausdrücke gibt es in PVS die beiden Konstanten `TRUE` und `FALSE`, sowie die üblichen logischen Operatoren: `NOT`, `AND`, `OR`, `IMPLIES`, die semantische Äquivalenz `IFF`, die Quantoren `FORALL` und `EXISTS`, sowie Gleichheit `=` und Ungleichheit `/=`.

Durch einen `LAMBDA`-Ausdruck lässt sich in PVS eine unbenannte, also anonyme Funktion ausdrücken. Die Funktion $f(x) = 3x - 4$ kann auch geschrieben werden als

```
(LAMBDA (x: int): 3 * x - 4)
```

Dieser Ausdruck hat den Typ `[int -> int]`. Eine Funktionsapplikation stellt sich beispielsweise wie folgt dar:

```
(LAMBDA (x: int): 3 * x - 4)(2)
```

Die Quantoren `FORALL` und `EXISTS` haben eine ähnliche Syntax wie der `LAMBDA`-Ausdruck: dem Schlüsselwort für den Operator folgt eine Liste von Variablenbindungen und ein Ausdruck. Es entsteht jeweils ein lokaler Gültigkeitsbereich für die gebundenen Variablen.

Da PVS, wie erwähnt, auf einer Logik höherer Ordnung basiert, ist eine Quantifizierung über alle Typen möglich genauso wie die Definition von Funktionen höherer Ordnung. Funktionen können überladen werden.

Bedingte Ausdrücke werden in PVS mit `IF-THEN-ELSE` gebildet:

```
IF cond
  THEN expr1
  ELSE expr2
ENDIF
```

Der Typ des Gesamtausdrucks ist gleich dem Typ von `expr1` und `expr2`, während `cond` vom Typ `bool` ist. Der `ELSE`-Zweig ist obligatorisch, da es sich um einen Ausdruck und nicht um eine Anweisung handelt. Über `ELSIF-THEN` können weitere bedingte Zweige eingefügt werden.

```
IF cond1
  THEN expr1
  ELSIF cond2 THEN expr2
  ELSE expr3
ENDIF
```

Mit `LET` werden lokal Werte an Variablen gebunden, die innerhalb des Ausdrucks dann referenziert werden können. Der folgende Ausdruck ergibt den Wert 6:

```
LET x: int = 2, y: int = x * x IN x + y
```

Die Typ-Deklarationen der Variablen im `LET`-Ausdruck sind optional. Intern wird ein `LET`-Ausdruck in einen `LAMBDA`-Ausdruck transformiert. Aus dem Beispiel entsteht so:

```
(LAMBDA (x: int): (LAMBDA (y: int): x + y)(x * x))(2)
```

Einzelne Werte von Funktionen, Strukturen und Tupel können mithilfe des `WITH`-Ausdrucks modifiziert werden. Das Ergebnis eines solchen Ausdrucks ist genau die gleiche Funktion, Struktur bzw. das gleiche Tupel bis auf die an den spezifizierten Argumenten veränderten Werte. Die sich aus folgendem Ausdruck

```
identity WITH [(0) := 1, (1) := 2]
```

ergebende Funktion ist also gleich der in `prelude.pvs` vordefinierten Identitätsfunktion, die ausschließlich an den Argumenten 0 und 1 veränderte Funktionswerte aufweist.

Projektionen einzelner Elemente aus einem Tupel oder einer Funktion können mithilfe der Projektionsoperatoren `'1`, `'2`, ... bewerkstelligt werden.

Sämtliche Funktionen müssen in PVS total sein, also an jeder Stelle ihres Definitionsbereiches definiert sein. Für Funktionen, die in sich partiell sind, bietet sich in PVS die Möglichkeit, den Definitionsbereich der Funktion über einen prädikativen Teiltyp geeignet einzuschränken und so die Funktion als total zu definieren. Wird eine Funktion hingegen rekursiv definiert, muss ihre Terminierung nachgewiesen werden. Syntaktisch geschieht dies durch die obligatorische Angabe einer Maßfunktion mittels `MEASURE`. Die Maßfunktion bildet das Rekursionsargument auf einen Typ mit einer wohlfundierten Ordnung ab. Bezüglich dieser Ordnung muss das Argument in jedem Rekursionsschritt kleiner werden. Das System erzeugt zu diesem Zweck eine Terminations-TCC, in der genau dies bewiesen werden muss. Als Beispiel betrachten wir die in [4] angegebene rekursive Definition der mathematischen Fakultätsfunktion.

4

```
fac(x: nat): RECURSIVE nat =
  IF x = 0
    THEN 1
    ELSE x * fac(x - 1)
  ENDIF
 MEASURE (LAMBDA (x: nat): x)
```

Die Maßfunktion ist hier die Identitätsfunktion auf dem Datentyp `nat`, als wohlfundierte Relation dient die `<`-Relation. Es wird folgende Terminations-TCC generiert, die aber vom System autonom bewiesen werden kann.

```
% Termination TCC generated (at line 16, column 29) for  fac(x - 1)
  % proved - complete
fac_TCC2: OBLIGATION FORALL (x: nat): NOT x = 0 IMPLIES x - 1 < x;
```

Die `LAMBDA`-Bindung für die Maßfunktion könnte in diesem Fall weggelassen werden, da während der Typüberprüfung automatisch die entprechende Bindung eingefügt werden würde. Ein weiteres Beispiel einer rekursiv definierten Funktion ist mit der `reverse`-Funktion für den Listen-Datentyp in 5 dargestellt.

Der `CASES`-Ausdruck wird in PVS vor allem zur Abarbeitung der einzelnen Alternativen bei induktiv definierten abstrakten Datentypen verwendet. Ein Beispiel für die entsprechende Verwendung für den Datentyp `list` ist in 5 abgebildet. Eine Liste ist entweder die leere Liste `null` oder sie setzt sich per `cons` aus einem Kopfelement und einer Restliste zusammen. `CASES` arbeitet hierbei nach dem Prinzip des Pattern-Matchings, um die passende Instanziierung zu finden.

5

```
reverse(l): RECURSIVE list[T] =
  CASES l OF
    null: l,
    cons(h, t): append(reverse(t), cons(h, null))
  ENDCASES
 MEASURE length(l)
```

Mithilfe von Theoremen werden innerhalb einer Theorie Aussagen spezifiziert, die bewiesen werden müssen. Syntaktisch wird ein Theorem in PVS über einen eindeutigen Namen gefolgt von einem Schlüsselwort und einem Ausdruck vom Typ `bool` definiert. Über den Namen kann das entsprechende Theorem möglicherweise später in einem Beweis referenziert werden. Als Schlüsselworte stehen `THEOREM`, `LEMMA`, `CHALLENGE`, `CLAIM`, `CONJECTURE`, `COROLLARY`, `FACT`, `FORMULA`, `LAW`, `LEMMA`, `PROPOSITION`, `SUBLEMMA` zur Verfügung. Die Wahl des Schlüsselwortes ist egal – sie dienen ausschließlich der Klassifikation der Formeln im Sinne des Anwenders. In 6 ist ein auf 5 Bezug nehmendes Lemma angegeben.

6

```
reverse_prop: LEMMA
  FORALL (l: list[T]): reverse(reverse(l)) = l
```

Axiome werden syntaktisch unter Verwendung des Schlüsselwortes `AXIOM` genauso wie Theoreme spezifiziert und verwendet, müssen jedoch nicht bewiesen werden.

3.2 Der Beweiser

Das Beweiser-Modul von PVS basiert auf dem *Sequenzenkalkül*, oft auch nach seinem Entwickler Gerhard Gentzen[2] als *Gentzenkalkül* bezeichnet [OSRSC01b, vHP06]. Das Grundelement des Kalküls – die Sequenz – ist ein Ausdruck der Form $\Gamma \vdash \Delta$, wobei Γ und Δ endliche Multimengen von Formeln sind. Man bezeichnet Γ als *Antezedent* und Δ als *Sukzedent* der Sequenz. Eine Sequenz ist dann gültig, wenn jedes Modell von Γ auch Modell für mindestens eine Formel aus Δ ist. Somit lässt sich eine Sequenz $A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_n$ als Implikation $A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow B_1 \vee B_2 \vee \ldots \vee B_n$ interpretieren. In der Beweiserkomponente von PVS hat eine solche Sequenz, die ein Beweisziel verkörpert, die folgende Gestalt.

7

```
[-1] A1
[-2] A2
  ...
[-n] An
  |-------
[1]  B1
[2]  B2
  ...
[n]  Bn
```

Unter der Annahme, dass die Formeln $A_1, A_2, \ldots, A_n$ *wahr* sind, wird nun versucht, eine der Formeln B_i als *wahr* nachzuweisen. Das einzige Axiom im Sequenzenkalkül ist $\Gamma, A \vdash A, \Delta$. Es gibt im Sequenzenkalkül sowohl strukturelle als auch logische Schlussregeln. Durch die Symmetrie zwischen Antezedent und Sukzedent kann jede Teilformel negiert auf der jeweils anderen Seite erscheinen. Die Nummerierung der einzelnen Teilformeln der Sequenzdarstellung in PVS dient der Referenzierung während des Beweises.

Ein Beweis in PVS wird „rückwärts“ konstruiert: ausgehend vom Beweisziel wird dieses fortlaufend in Unterziele zerlegt, bis diese nach entsprechender Anwendung der Beweiserregeln trivial zu beweisen sind. Ein Unterziel gilt dann als bewiesen, wenn es das Axiom des Sequenzenkalküls erfüllt, eine Formel im Antezedent *falsch* oder eine Formel im Sukzedent *wahr* ist. Wurde eines der Unterziele bewiesen, springt der Fokus des Beweisers auf das nächste, noch zu beweisende Unterziel. Die Abarbeitungsreihenfolge der einzelnen Unterziele ist hierbei beliebig. Sobald alle Unterziele bewiesen sind, gilt die Ausgangssequenz als bewiesen.

Dem Benutzer stehen zur interaktiven Beweissteuerung eine Vielzahl an Beweiserkommandos zur Verfügung. Sie implementieren zumeist Gruppen von strukturellen bzw. logischen Regeln des Sequenzenkalküls. Es folgt eine Auswahl einiger wichtiger Kommandos [OSRSC01b]:

`(flatten)` Elimination von äußeren Disjunktionen im Sukzedent und von Konjunktionen im Antezedent.

[2] Gerhard Karl Erich Gentzen (*1909 – †1945): Mathematiker und Logiker

`(split)` Aufspaltungen von äußeren Konjunktionen, bedingten Verzweigungen im Sukzedent sowie von äußeren Disjunktionen, bedingten Verzweigungen im Antezedent in entsprechende neue Unterziele im Beweis.

`(lift-if)` Bedingte Verzweigungen werden in den Teilformeln auf die äußerste Ebene gehoben.

`(skolem!)` Ersetzung der durch einen Allquantor in einer Formel im Sukzedent bzw. der durch einen Existenzquantor in einer Formel im Antezedent gebundenen Variablen durch Skolemkonstanten, deren Namen automatisch generiert werden.

`(skosimp*)` Wiederholte Anwendung von `(skolem!)` und `(flatten)`.

`(inst)` Instanziierung von Existenzquantoren im Sukzedent und Allquantoren im Antezedent. Mit `(inst?)` versucht das System automatisch einen geeigneten Term für die Instanziierung zu finden.

`(lemma` *`name`*`)` Einfügen eines Lemmas oder Theorems in den Antezedent.

`(use` *`name`*`)` Kombination von `(lemma)` und `(inst?)`

`(beta)` Anwendung der β-Reduktion.

`(expand` *`name`*`)` Expansion, also „Auspacken“, einer Funktionsdefinition.

`(replace` *`fnum`*`)` Hat eine Formel im Antezedent die Gestalt einer Gleichung, so werden alle Vorkommen der linken Seite in anderen Teilformeln durch die rechte Seite ersetzt. Um die Ersetzung umgekehrt erfolgen zu lassen, kann die Option `:dir rl` angegeben werden.

`(rewrite` *`fnum/lemma`*`)` Termersetzung mit Substitution von Variablen. Als Grundlage kann ein Lemma oder eine Formel im Antezedent dienen.

`(prop)` Durchführung bestimmter aussagenlogischer Vereinfachungen.

`(assert)` Simplifikation durch Entscheidungsprozeduren. Offensichtliche Beweisziele werden aufgelöst.

`(grind)` Leistungsfähige Strategie, die wiederholt u.a. `(rewrite)`, `(assert)`, `(skolem!)`, `(replace)`, `(lift-if)`, `(flatten)` ausführt.

`(induct var)` Induktionsbeweis für eine allquantifizierte Variable im Antezedent unter Zuhilfenahme eines passenden Induktionsaxioms. Es werden automatisch die Unterziele für den Induktionsanfang und -schritt gebildet.

Viele der aufgeführten Beweiserkommandos können zumeist über optionale Parameter noch weiter an die jeweilige Situation angepasst werden [OSRSC01b]. Eine Hilfe zu einer bestimmten Beweiserregel kann über `(help` *`command`*`)` angezeigt werden. Der Befehl `(undo)` nimmt das zuletzt getätigte Beweiskommando zurück. Mithilfe von `(postpone)`

wird der Fokus auf das nächste noch offene Unterziel gelenkt. Ein Beweis kann mit `(quit)` beendet werden.

PVS verfügt über einen Batch-Modus, mit dem ein bereits geführter Beweis noch einmal ausgeführt und, auf Wunsch im Einzelschrittmodus, durchlaufen werden kann. Der Beweisbaum zum aktuell bearbeiteten Beweis kann in einem Fenster grafisch veranschaulicht werden. Mithilfe einer integrierten Strategiesprache können neue Beweisstrategien erstellt werden.

3.3 Ein kleines Anwendungsbeispiel

Wir wollen folgenden mathematischen Sachverhalt mit vollständiger Induktion in PVS beweisen:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{3.1}$$

In 8 ist die zugehörige Spezifikation angegeben. Die Theorie `beispiel` umfasst neben einer Implementierung der Summenformel in Form der rekursiven Funktion `sum` auch das Theorem `gauss`, das der Gleichung 3.1 entspricht. Man beachte, dass die Variable `n` im Theorem von PVS als allquantifiziert angenommen wird, da eine explizite Quantifizierung fehlt.

8

```
beispiel: THEORY
 BEGIN

  n: VAR nat

  sum(n): RECURSIVE nat =
    IF n = 0
      THEN 0
      ELSE n + sum(n - 1)
    ENDIF
   MEASURE n

  gauss: THEOREM
    sum(n) = (n * (n + 1)) / 2

 END beispiel
```

Bevor der Beweis für das Theorem gestartet werden kann, muss die Theorie auf eventuelle Typfehler überprüft werden. Der Typüberprüfer wird entweder über das PVS Menü im GNU/Emacs bzw. XEmacs oder über die Tastenkombination `M-x tcp` gestartet. Durch die rekursive Definition von `sum` werden zwei TCC's generiert, die vom System aber selbstständig bewiesen werden können. Die TCC's einer Theorie lassen sich mittels `M-x show-tccs` anzeigen und eventuell beweisen.

Um nun einen Beweis für das Theorem zu starten, platziert man den Cursor auf das zu beweisende Theorem und startet dann per `M-x prove` die Beweiserkomponente von PVS:

9

```
gauss :

  |-------
{1}   FORALL (n: nat): sum(n) = (n * (n + 1)) / 2

Rule?
```

Das Beweisziel steht zu Beginn im Sukzedent. Am Prompt `Rule?` wartet das System nun auf die Eingabe eines Beweiskommandos. Mit `(induct)` starten wir einen Induktionsbeweis über die Variable `n`:

```
Rule? (induct "n")
Inducting on n on formula 1,
this yields  2 subgoals:
gauss.1 :

  |-------
{1}   sum(0) = (0 * (0 + 1)) / 2
```

Das Beweisziel wird automatisch in zwei Unterziele aufgespaltet: dem Induktionsanfang und dem Induktionsschritt. Eine Expansion der Funktionsdefinition von `sum` und ein anschließendes `(assert)` schließen bereits den Beweis des Induktionsanfangs ab:

```
Rule? (expand "sum")
Expanding the definition of sum,
this simplifies to:
gauss.1 :

  |-------
{1}   0 = 0 / 2

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of gauss.1.
```

Im Anschluss daran wendet sich der Fokus des Beweisers dem nächsten noch unbewiesenen Unterziel, in unserem Fall dem Induktionsschritt, zu:

```
gauss.2 :

  |-------
{1}   FORALL j:
        sum(j) = (j * (j + 1)) / 2 IMPLIES
         sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2

Rule?
```

Vor `IMPLIES` steht die Induktionsvoraussetzung, dahinter die Behauptung. Mit dem Befehl `(skosimp)` wird die Variable `j` durch eine Skolemkonstante `j!1` ersetzt und die Implikation im Sukzedent eliminiert.

```
Rule? (skosimp)
Skolemizing and flattening,
this simplifies to:
gauss.2 :

{-1}  sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   sum(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2

Rule?
```

Anschließend wird in der Teilformel `1` im Sukzedent die Funktion `sum` expandiert.

```
Rule? (expand "sum" 1)
Expanding the definition of sum,
this simplifies to:
gauss.2 :

[-1]  sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   1 + sum(j!1) + j!1 = (2 + j!1 + (j!1 * j!1 + 2 * j!1)) / 2

Rule?
```

Jetzt kann die im Antezedent stehende Induktionsvoraussetzung in die Gleichung im Sukzedent eingesetzt werden. Dies geschieht mit dem Kommando `(replace -1)`. Mit einem abschließenden `(assert)` kann PVS von der Gültigkeit der Gleichung im Sukzedent überzeugt werden. Hiermit ist der Beweis erfolgreich beendet.

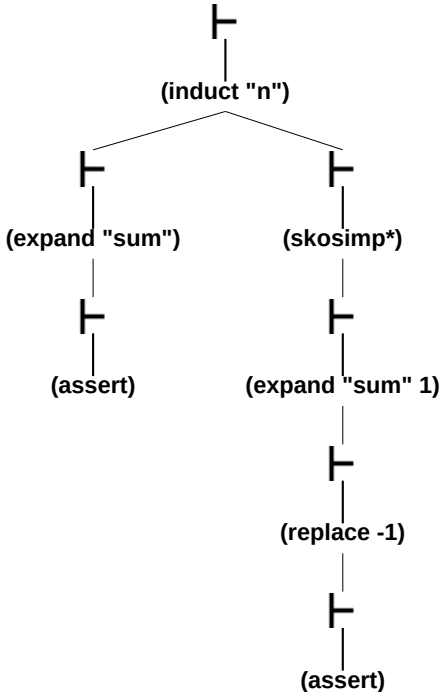


Abbildung 3.1: *Beweisbaum zum betrachteten Beispiel*

```
Rule? (replace -1)
Replacing using formula -1,
this simplifies to:
gauss.2 :

[-1]  sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   1 + (j!1 * (j!1 + 1)) / 2 + j!1 =
       (2 + j!1 + (j!1 * j!1 + 2 * j!1)) / 2

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of gauss.2.

Q.E.D.
```

Einfache Induktionsbeweise wie dieser können aber auch automatisch mit der Strategie `(induct-and-simplify)` geführt werden. Dies zeigt, dass durch den Einsatz von bestehenden oder auch selbst erstellten Beweisstrategien der Aufwand für die Beweisführung reduziert werden kann:

```
gauss :

  |-------
{1}   FORALL (n: nat): sum(n) = (n * (n + 1)) / 2

Rule? (induct-and-simplify "n")
sum rewrites sum(0)
  to 0
sum rewrites sum(1 + j!1)
  to 1 + sum(j!1) + j!1
By induction on n, and by repeatedly rewriting and simplifying,
Q.E.D.
```

4 Lipton's Experiment zur Lösung von SAT

In diesem Kapitel wollen wir ein DNA-basiertes Lösungsverfahren für das bekannte NP-vollständige Erfüllbarkeitsproblem der Aussagenlogik und die zugehörige Verifikation beschreiben.

4.1 Kurze Beschreibung der Aussagenlogik

Aussagen und ihre Relation zueinander sind Gegenstand der Aussagenlogik. Der Sprache der Aussagenlogik liegt ein Alphabet zugrunde, aus dem die Sätze der Sprache, die man als *Formeln* bezeichnet, gebildet werden können [Höl01].

Definition 4.1. *Das Alphabet der Aussagenlogik besteht aus*

- *einer abzählbar unendlichen Menge von aussagenlogischen Variablen $\mathcal{V}$,*
- *der immer wahren Aussage „1“ und der immer falschen Aussage „0“*
- *dem einstelligen Junktor $\neg$ sowie den zweistelligen Junktoren $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$,*
- *den Sonderzeichen „ (“ und „) “.*

Die aussagenlogischen Variablen, sowie die Konstanten „0“ und „1“ werden auch als *atomare Formeln* oder kurz *Atome* bezeichnet und stehen für die Elementaraussagen, deren innere Struktur nicht weiter untersucht wird. Junktoren sind logische Operatoren, mit denen einfache Aussagen zu komplexeren Aussagen verknüpft werden.

Definition 4.2. *Die Menge der* aussagenlogischen Formeln $\mathcal{A}$ *ist induktiv definiert als kleinste Menge, die die Menge $\mathcal{V}$ beinhaltet und den folgenden Bedingungen genügt:*

1. *Falls $P \in \mathcal{A}$, dann ist $\neg P \in \mathcal{A}$*
2. *Falls $P, Q \in \mathcal{A}$ und $\circ$ ein zweistelliger Junktor, dann ist $(P \circ Q) \in \mathcal{A}$*

Die Aufgabe der Semantik ist es nun, den Ausdrücken der Sprache der Aussagenlogik eine Bedeutung zuzuordnen. Man spricht in diesem Zusammenhang von einer *zweiwertigen* Logik, da in der Aussagenlogik Atome aber auch komplexere Formeln nur als entweder *wahr* oder *falsch* interpretiert werden. Formal gesehen ist die Bedeutungszuordnung für atomare Formeln, die als *Interpretation* bezeichnet wird, eine Abbildung von der Menge der Atome in die Menge der Wahrheitswerte $\{0, 1\}$, wobei hier 0 für *falsch* und 1 für *wahr* steht. Um zusammengesetzte Formeln bewerten zu können, muss zunächst für jeden Junktor definiert werden, welcher Wahrheitswert für die einzelnen Wahrheitswertkombinationen seiner Argumente zurückgeliefert werden soll.

Definition 4.3. *Die Bedeutungszuordnung eines n-stelligen Junktors sei definiert durch die Abbildung*

$$h : \{0,1\}^n \longrightarrow \{0,1\}.$$

Diese Abbildung, die als Wahrheitstabelle *bezeichnet wird, ist für obige Junktoren in der Tabelle 4.1 dargestellt.*

a	$h_\neg(a)$
0	1
1	0

(a)

a	b	$h_\vee(a,b)$	$h_\wedge(a,b)$	$h_\rightarrow(a,b)$	$h_\leftrightarrow(a,b)$
0	0	0	0	1	1
0	1	1	0	1	0
1	0	1	0	0	0
1	1	1	1	1	1

(b)

Tabelle 4.1: *Die Wahrheitstabellen für (a) den unären Junktor Negation und (b) die binären Junktoren Disjunktion, Konjunktion, Implikation und Äquivalenz.*

Die Junktoren $\neg$ und $\vee$ bilden eine vollständige Menge von Junktoren, da durch sie alle anderen Junktoren gebildet werden können. Zum Beispiel gilt $(a \wedge b) = \neg(\neg a \vee \neg b)$ und $(a \rightarrow b) = (\neg a \vee b)$.

Die Interpretation von zusammengesetzten Formeln kann nun mithilfe einer Abbildung induktiv über der Menge der Formeln definiert werden.

Definition 4.4. *Die aussagenlogische Interpretation von Formeln ist eine Abbildung von der Menge $\mathcal{A}$ der Formeln auf die Menge $\{0,1\}$ der Wahrheitswerte, die die folgenden Bedingungen erfüllt. Seien $P, Q \in \mathcal{A}$ und $\circ$ ein zweistelliger Junktor, so gelte*

1. $\pi(\neg P) = h_\neg(\pi(P))$
2. $\pi(P \circ Q) = h_\circ(\pi(P), \pi(Q))$

Wichtige semantische Eigenschaften von Formeln sind die *Allgemeingültigkeit*, *Erfüllbarkeit*, *Widerlegbarkeit* und *Unerfüllbarkeit*.

- Eine Formel heißt *allgemeingültig*, wenn sie unter allen Interpretationen auf *wahr* abgebildet wird. Eine solche Formel nennt man auch *Tautologie*.
- Eine Formel heißt *erfüllbar*, wenn es eine Interpretation gibt, unter der die Formel *wahr* ist.
- Eine Formel heißt *widerlegbar*, wenn es eine Interpretation gibt, unter der die Formel *falsch* ist.

- Eine Formel heißt *unerfüllbar*, wenn es keine Interpretation gibt, unter der die Formel *wahr* ist.

Für die Spezifikation von Algorithmen ist es oft von Vorteil, wenn sich die aussagenlogischen Formeln in einer standardisierten Form, einer so genannten *Normalform*, befinden. Wir wollen hier die für die weiteren Betrachtungen relevante konjunktive Normalform definieren.

Definition 4.5.

- *Ein* Literal *ist eine aussagenlogische Variable (positives Literal), oder die Negation einer aussagenlogischen Variable (negatives Literal).*
- *Eine* Klausel *ist eine Disjunktion von Literalen. Seien $L_1, L_2, \ldots, L_n$ Literale. Eine Klausel wird wie folgt notiert*

$$[L_1, L_2, \ldots, L_n] := (L_1 \vee L_2 \vee \ldots \vee L_n)$$

- *Eine aussagenlogische Formel befindet sich in* Klauselform *oder in* konjunktiver Normalform*, falls sie eine Konjunktion von Klauseln ist. Seien $C_1, C_2, \ldots, C_n$ Klauseln. Eine Formel in Klauselform wird wie folgt notiert*

$$\langle C_1, C_2, \ldots, C_n \rangle := (C_1 \wedge C_2 \wedge \ldots \wedge C_n)$$

Es kann gezeigt werden, dass es einen Algorithmus gibt, der jede aussagenlogische Formel in eine semantisch äquivalente Formel in konjunktiver Normalform umformt [Fit96, Höl01].

4.2 Das Erfüllbarkeitsproblem der Aussagenlogik

Das *Erfüllbarkeitsproblem* der Aussagenlogik, auch als *SAT*-Problem bezeichnet (von *satisfiability*), ist ein klassisches Entscheidungsproblem, das in vielen Bereichen der Informatik, besonders in ihrem theoretischen Zweig wie der Komplexitätstheorie und der Verifikation, eine hohe Bedeutung besitzt.

Definition 4.6. *Das Erfüllbarkeitsproblem der Aussagenlogik beschäftigt sich mit der Fragestellung, ob eine gegebene aussagenlogische Formel erfüllbar ist.*

Wir wollen davon ausgehen, dass die Formel in konjunktiver Normalform vorliegt. Es gilt nun zu entscheiden, ob es eine Interpretation mit Wahrheitswerten für die einzelnen Variablen gibt, unter der die gesamte Formel auf *wahr* abgebildet wird. Da die Formel eine konjunktive Verknüpfung von Klauseln ist, muss jede der Klauseln den Wert *wahr* zurückliefern.

4.3 Komplexität des Erfüllbarkeitsproblems

Bevor auf die Komplexität des Erfüllbarkeitsproblem eingegangen wird, sollen hier kurz wichtige Begriffe der Komplexitätstheorie skizziert werden.

Die Komplexität von algorithmisch behandelbaren Problemen wird innerhalb der Komplexitätstheorie, einem Teilgebiet der theoretischen Informatik, untersucht. Um den Bedarf verschiedener Ressourcen wie Zeit und Speicherplatz für die Lösung eines algorithmischen Problems zu bestimmen, bezieht man sich auf formal definierte Berechnungsmodelle wie die Turingmaschine. Ziel der Komplexitätstheorie ist nun die Klassifizierung von Problemen gemäß ihrer Lösbarkeit in Bezug auf den Ressourcenbedarf. Die Einteilung erfolgt in Komplexitätsklassen, wobei die Probleme als formale Sprachen aufgefasst werden. So lässt sich jedes Entscheidungsproblem auch auf das Wortproblem einer formalen Sprache überführen. Die folgenden Definitionen und Sätze wurden [Baa05] entnommen.

Definition 4.7. *Es sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion und M eine deterministische Turingmaschine (DTM) über dem Alphabet Σ. M heißt* $f(n)$-zeitbeschränkt*, falls für alle $w \in \Sigma^*$ M bei der Eingabe w nach $\leq f(|w|)$ Rechenschritten anhält.*

Die Definition 4.7 kann auf nichtdeterministische Turingmaschinen (NTM) übertragen werden. Dafür muss die Zeitbeschränkung für alle möglichen Berechnungen bei der Eingabe w zutreffen.

Definition 4.8. *Die Klasse $DTIME(f(n))$ besteht aus allen Sprachen L, für die es eine $f(n)$-zeitbeschränkte DTM gibt, die L entscheidet. Die Klasse $NTIME(f(n))$ besteht aus allen Sprachen L, für die es eine $f(n)$-zeitbeschränkte NTM gibt, die L akzeptiert.*

Definition 4.9. *Die Klasse* P *ist definiert als*

$$\mathrm{P} := \bigcup_{p \text{ Polynom}} DTIME(p(n))$$

Die Klasse NP *ist definiert als*

$$\mathrm{NP} := \bigcup_{p \text{ Polynom}} NTIME(p(n))$$

Die Probleme der Klasse P gelten allgemein als praktisch effizient lösbar. Die Klasse NP enthält viele Probleme, für die nur Algorithmen mit exponentiellem Zeitaufwand bekannt sind. Diese Probleme gelten als nicht effizient lösbar.

Satz 4.1. $\mathrm{P} \subseteq \mathrm{NP}$

Die Frage, ob NP = P oder NP $\subset$ P wird als P/NP-Problem bezeichnet und gilt als das wichtigste offene Problem der theoretischen Informatik.

Definition 4.10. *Die Sprache $L \subseteq \Sigma^*$ sei auf die Sprache $L' \subseteq \Gamma^*$ polynomial reduzierbar, bezeichnet als $L \leq_p L'$, falls es eine totale und mit polynomialer Komplexität berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ gibt, so dass für alle $w \in \Sigma^*$ gilt: $w \in L \Leftrightarrow f(w) \in L'$*

Definition 4.11. *Eine Sprache L heißt* NP*-vollständig, falls $L \in$ NP gilt und für alle $L' \in$ NP gilt: $L' \leq_p L$*

Die NP-vollständigen Probleme sind die schwierigsten Probleme aus NP. Sie sind mindestens so schwer zu lösen, wie jedes andere Problem aus NP. Für keines der NP-vollständigen Probleme sind zeiteffiziente Lösungsalgorithmen bekannt.

Satz 4.2 (Satz von Cook und Levin). Das Erfüllbarkeitsproblem der Aussagenlogik ist NP-vollständig.

Der Beweis des Satzes 4.2 ist zum Beispiel in [Blö03], [Rei90] oder [Bre92] nachzuvollziehen.

4.4 Lipton's Experiment

Richard J. Lipton, Professor an der Princeton University, verwendete zur Entwicklung eines DNA-basierten Lösungsprinzips für das SAT-Problem [Lip95] das von ihm und Adleman entwickelte Filtering Modell (siehe Abschnitt 2.3.3), welches die Modelloperationen *separate*, *merge* und *detect* zur Verfügung stellt.

Alle Operationen und Vorgänge werden auf Reagenzgläsern, den Tubes, angewendet. Die DNA-Stränge im Tube, die man innerhalb des Filtering Modells als Aggregate bezeichnet, repräsentieren hierbei die Pfade eines Graphen G_n, der die Knoten $a_1, x_1^0, x_1^1, a_2, x_2^0, x_2^1, \ldots, a_{n+1}$ beinhaltet, siehe Abbildung 4.1. Es bestehen in G_n Kanten von a_k zu x_k^0 sowie von a_k zu x_k^1, und jeweils von x_k^0 sowie x_k^1 zu a_{k+1}. Es wird vorausgesetzt, dass alle möglichen Pfade, die von Knoten a_1 nach a_{n+1} führen, DNA-kodiert zu Beginn im Reagenzglas vorhanden sind. Jeder dieser Pfade kodiert eine n-stellige Bitkette auf folgende Art: der Pfad $a_1 \to x_1^0 \to a_2 \to x_2^1 \to a_3$ repräsentiert beispielsweise die Bitkette 01. Je nachdem ob der Pfad durch x_k^0 oder x_k^1 führt, wird dies an dieser Stelle als 0 oder 1 interpretiert.

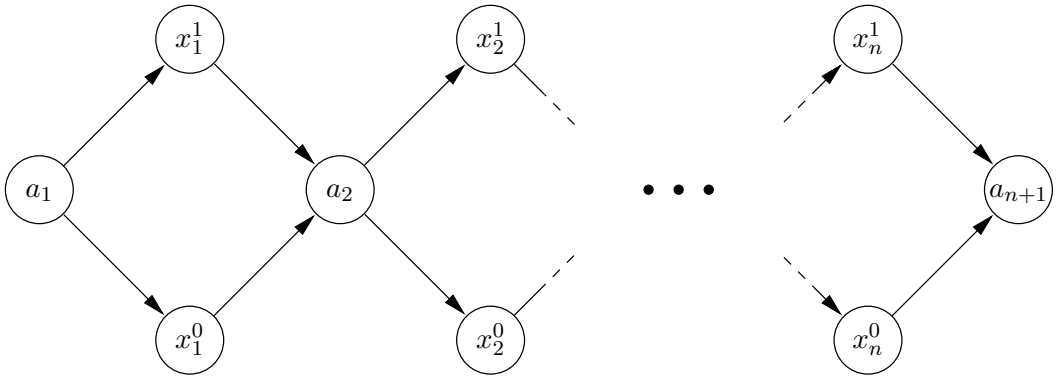


Abbildung 4.1: *Der Graph G_n im allgemeinen Fall für eine Formel mit n unterschiedlichen Variablen*

Die Bitketten stehen im Algorithmus für die aussagenlogischen Interpretationen der Formel, deren Erfüllbarkeit letztendlich bewiesen werden soll. Besitzt die Formel n unterschiedliche aussagenlogische Variablen, so werden durch die DNA-Stränge im Tube, und somit auch durch die Pfade der Länge $n+1$ im Graphen G_n, alle möglichen Variablenbelegungen kodiert. Es gilt nun herauszufinden, ob eine davon die gegebene Formel auf *wahr* abbildet.

Um die Pfade des Graphen geeignet in DNA-Moleküle zu kodieren, wird wie folgt vorgegangen. Jeder Knoten wird mittels eines DNA-Einzelstrangs der Länge l mit zufälliger Basenabfolge dargestellt. Die Länge sollte so gewählt werden, dass jeder Knoten mit hoher Wahrscheinlichkeit eine eindeutige Repräsentation besitzt. Man stelle sich den besagten Einzelstrang als zweigeteilt vor: die erste Hälfte sei mit p_i und die zweite mit q_i bezeichnet. Die Kanten werden ebenfalls durch Einzelstränge der Länge l kodiert. Für die Kante vom Knoten i zum Knoten j ist jedoch die erste Hälfte $\tilde{q}_i$ entgegengesetzt komplementär zu q_i des Knotens i, und die zweite Hälfte $\tilde{p}_j$ entgegengesetzt komplementär zu p_j des Knotens j.

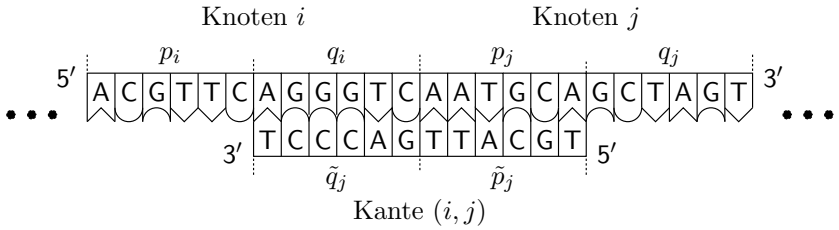


Abbildung 4.2: *Beispielhafte konkrete Kodierung zweier Knoten und ihrer zugehörigen Kante eines Pfades, hier nach der Hybridisierung*

Zur Bildung der vollständigen Pfade werden viele Millionen Kopien der Knoten- und Kantenrepräsentationen in das Reagenzglas gegeben. Für den Start- und Endknoten des Graphen reicht es aus, nur jeweils die q- beziehungsweise p-Hälfte zu verwenden. Durch die Operation *Hybridisierung* lagern sich nun die jeweils passend komplementären Sequenzen aneinander und bilden so bei ausreichender Anzahl an Strangkopien mit hoher Wahrscheinlichkeit alle verschiedenen Pfade des Graphen G_n, und mithin alle möglichen Variablenbelegungen der aussagenlogischen Formel. Es wird also der gesamte Suchraum des Problems DNA-kodiert im initialen Tube bereitgestellt.

Für eine Formel mit n unterschiedlichen Variablen gibt es insgesamt 2^n verschiedene Variablenbelegungen und ebenso viele Pfade der Länge $n+1$ im Graphen G_n. Trotz dieser exponentiellen Anzahl ist die Herstellung des Tubeinhalts zeiteffizient: es müssen für jeden Knoten und jede Kante, deren Anzahl linear in Abhängigkeit von n wächst, die entsprechenden DNA-Stränge synthetisiert und vervielfältigt werden. Im Anschluss daran ist nur eine einzelne Hybridisierungs-Operation zur Bildung der Pfade nötig, deren Zeitaufwand unabhängig von der Problemgröße ist, da sie echt parallel auf den gesamten Reagenzglasinhalt wirkt.

Um nun aus der Vielzahl der Lösungsmöglichkeiten im initialen Tube die richtigen

herauszufiltern, wird die *separate*-Operation auf die Tubes angewendet. Hierbei werden mit $+(T, i, b)$ all jene Bitketten im Tube T bezeichnet, deren i-tes Bit gleich b ist, wobei $b \in \{0, 1\}$. In der laborpraktischen Implementierung der *separate*-Operation entspricht dies dem Separieren aller DNA-Stränge im Tube T, die die Sequenz für x_i^b enthalten. In manchen Fällen wird auch der Rest $-(T, i, b)$ benötigt, also die DNA-Stränge, die die Bedingung der *separate*-Operation nicht erfüllen. Es werden noch zwei weitere Operationen des Filtering-Modells angewendet: die *merge*-Operation ermöglicht die Vermischung zweier Tubeinhalte und die *detect*-Operation implementiert den Test auf Vorhandensein von DNA im übergebenen Tube (siehe Abschnitt 2.3.3).

Bevor die allgemeine Vorgehensweise dargestellt wird, wollen wir zum besseren Verständnis hier ein einfaches Beispiel betrachten. Es sei folgende Formel gegeben:

$$\varphi = (p \vee q) \wedge (\neg p \vee \neg q)$$

Wie leicht zu erkennen ist, führen die Interpretationen $p = 1$, $q = 0$ sowie $p = 0$, $q = 1$ zu einer Abbildung von φ auf den Wahrheitswert 1. Im initialen Tube T befinden sich zu Beginn alle möglichen Variablenbelegungen für die Formel φ, in unserem Fall also alle möglichen Bitketten der Länge zwei. Um nun die erfüllenden Variablenbelegungen herauszufiltern, wird φ elementeweise wie folgt abgearbeitet:

$T := \{00, 01, 10, 11\}$
$T' := \emptyset$

- Literal p

  $T^+ := +(T, 1, 1) \quad = \{10, 11\}$
  $T := -(T, 1, 1) \quad = \{00, 01\}$
  $T' := \texttt{merge}(T', T^+) \quad = \{10, 11\}$

- Literal q

  $T^+ := +(T, 2, 1) \quad = \{01\}$
  $T := -(T, 2, 1) \quad = \{00\}$
  $T' := \texttt{merge}(T', T^+) \quad = \{10, 11, 01\}$

$T := T' = \{10, 11, 01\}$
$T' := \emptyset$

- Literal $\neg p$

  $T^+ := +(T, 1, 0) \quad = \{01\}$
  $T := -(T, 1, 0) \quad = \{10, 01\}$
  $T' := \texttt{merge}(T', T^+) \quad = \{01\}$

- Literal $\neg q$

$$\begin{aligned} T^+ &:= +(T,2,0) && = \{10\} \\ T &:= -(T,2,0) && = \{01\} \\ T' &:= \texttt{merge}(T',T^+) && = \{01,10\} \end{aligned}$$

$T := T' = \{10, 01\}$
$\texttt{detect}(T) = \text{ja}$

Für alle Literale der ersten Klausel werden erst alle Variablenbelegungen „aufgesammelt", die diese Literale – und damit auch die Klausel – auf den Wahrheitswert 1 abbilden. Aus dieser Menge werden dann im Anschluss analog diejenigen Belegungen herausgefiltert, die darüber hinaus auch die zweite Klausel auf 1 abbilden – im Beispiel sind dies genau die Belegungen 10 und 01.

Nehmen wir nun aber an, wir haben eine allgemeine Formel in Klauselform

$$\begin{aligned} \varphi &= C_1 \wedge C_2 \wedge \ldots \wedge C_m \qquad \text{mit} \\ C_i &= L_{i,1} \vee L_{i,2} \vee \ldots \vee L_{i,r_i} \end{aligned}$$

gegeben. Der Prozess der Erstellung und DNA-Kodierung der Variablenbelegungen für die einzelnen Variablen der Formel φ ist hier in der Funktion `make_tube(`φ`)` zusammengefasst. Sie ist also für die Bildung des initialen Tubeinhalts verantwortlich. Um zu entscheiden, ob es in dieser Menge eine oder mehrere erfüllende Belegungen für φ gibt, kann man folgenden molekularen Algorithmus [Día03, DMJ02] angeben:

$T = \texttt{make\_tube}(\varphi)$
for $i := 0$ to m **do**
  $T' := \emptyset$
  for $j := 0$ to r_i **do**
    if $L_{i,j} = x_k$ **then**
      $T^+ := +(T,k,1)$
      $T := -(T,k,1)$
    else if $L_{i,j} = \neg x_k$ **then**
      $T^+ := +(T,k,0)$
      $T := -(T,k,0)$
    end if
    $T' := \texttt{merge}(T', T^+)$
  end for
  $T := T'$
end for
$\texttt{detect}(T)$

Mit dieser Vorgehensweise zur Lösung des Erfüllbarkeitsproblems für eine aussagenlogische Formel mit m Klauseln wächst die Anzahl der notwendigen `separate`-Aufrufe linear in m [Lip95]. Somit ist die Anzahl an insgesamt nötigen Experimentierschritte

nur linear von der Anzahl der Klauseln der gegebenen Formel abhängig. Der Algorithmus hat also eine lineare Zeitkomplexität. Der Platzbedarf hingegen ist exponentiell von der Anzahl der vorkommenden Variablen abhängig, da im initialen Tube 2^n unterschiedliche Variablenbelegungen bereitgestellt werden müssen, unter der Annahme, die Formel beinhaltet n unterschiedliche Variablen. Trotzdem ist die Herstellung des initialen Tubeinhalts wie beschrieben zeiteffizient möglich. Durch den exponentiellen Platzbedarf können jedoch auch nur Probleme bis zu einer bestimmten Problemgröße praktisch behandelt werden, da die Datendichte bei der DNA-Kodierung verglichen mit konventionellen Rechnern zwar sehr hoch, aber natürlich begrenzt ist.

Bisher wurde implizit vorausgesetzt, dass die molekularen Operationen idealisiert, also ohne Fehler ablaufen. Fast alle molekularbiologischen Operation besitzen aber Seiteneffekte, die den Ablauf eines solchen Experiments stören und zu einer Verfälschung der Ergebnisse führen können [HS04]. In Bezug auf die *separate*-Operation ist noch anzumerken, dass die Annahme über eine vollständige Separierung der Stränge nicht nötig ist. Befinden sich im initialen Tube viele Kopien der einzelnen Stränge, so muss nur sichergestellt sein, dass mit einer gewissen Wahrscheinlichkeit eine Anzahl von Strängen korrekt extrahiert wird [Lip95].

4.5 Verifikation von Lipton's Experiment

In den Veröffentlichungen [Día03] und [DMJ02] von C. Graciani Díaz et al. wurden erstmalig molekulare Programme bestimmter Modelle des DNA-Computing der formalen Verifikation zugeführt. Unter den untersuchten Programmen befindet sich auch die beschriebene Lösung des Erfüllbarkeitsproblems nach Lipton. Für die Spezifikation des Filtering-Modells und des molekularen Programms sowie die Beweisführung verwendete die Autorin das Verifikationssystem PVS. Im folgenden wollen wir ihre Implementation kurz beschreiben.

Die Aggregate eines Tubes werden durch Multimengen von Symbolen eines Alphabets Σ modelliert. Ein Tube ist dann eine Multimenge von Aggregaten. Multimengen werden in der Implementation als Funktionen mit dem Typ `[E -> nat]` modelliert, die jedes Element `E` auf eine natürliche Zahl, die Exemplaranzahl, abbilden. Die Aggregate im Tube sollen die einzelnen Pfade des Graphen G_n – und damit Variablenbelegungen der gegebenen aussagenlogischen Formel – kodieren, weshalb die Elemente von Σ Paare von einer aussagenlogischen Variable und einem Wahrheitswert, 0 oder 1, sind. Ein Paar $(x, 0)$ bedeutet somit, dass die aussagenlogische Variable x auf den Wahrheitswert 0 abgebildet wird. Die auf Erfüllbarkeit zu überprüfende aussagenlogische Formel `PF` ist in konjunktiver Normalform gegeben und dargestellt als endliche Sequenz von Klauseln. Eine Klausel wiederum ist eine endliche Sequenz von Literalen. Ein Literal ist ein geordnetes Paar von einem Marker `positive/negative` und einer aussagenlogischen Variable.

Die Operationen des Filtering-Modells, *separate*, *merge*, *detect*, sind durch entsprechend definierte Funktionen verfügbar. So wird die Operation *merge*, also die Vermischung des Inhaltes zweier Tubes, beispielsweise durch die Vereinigung zweier Multimengen realisiert.

Mit der Funktion `sat_lipton` wird zuerst rekursiv in Abhängigkeit von der gegebenen Formel der initiale Tubeinhalt gebildet. Im Tube befinden sich dann alle 2^n verschiedenen Variablenbelegungen, wobei n die Anzahl der in der Formel vorkommenden aussagenlogischen Variablen ist. Aus diesen werden nun mittels des im vorangegangen Abschnittes vorgestellten Algorithmus nach und nach die korrekten Lösungen, d. h. diejenigen Variablenbelegungen separiert, die die Formel erfüllen. Den Abschluss bildet ein Aufruf der `detect`-Operation, die das Tube auf das Vorhandensein von Aggregaten hin überprüft. Rückgabewert der Funktion `sat_lipton` ist entweder `YES` oder `NO`.

Das entsprechend aufgestellte Theorem zur Überprüfung der Korrektheit des Verfahrens lautet in PVS-Notation:

10

```
soundness: THEOREM
  sat_lipton(PF) = YES IMPLIES (EXISTS PI: PVal(PI, PF) = one)
```

Die Formel ist mit `PF` bezeichnet, während `PI` die aussagenlogische Interpretation darstellt. Mittels `PVal` wird eine Formel mithilfe einer Interpretation `PI` auf einen Wahrheitswert abgebildet, hier entweder `one` oder `zero`.

Über folgendes Theorem wurde die Eigenschaft der Vollständigkeit zum Ausdruck gebracht:

11

```
completeness: THEOREM
  (PVar(PF)`length > 0 AND (EXISTS PI: PVal(PI, PF) = one)) IMPLIES
   sat_lipton(PF) = YES
```

Hiervon ausgenommen ist die Formel der Länge Null, die per Definition allgemeingültig ist, jedoch zu einem leeren Tube und damit fälschlicherweise zur Entscheidung `NO` führen würde. Beide Theoreme wurden mithilfe zahlreicher Hilfslemmata in PVS bewiesen, womit das DNA-basierte Vorgehen von Lipton zur Lösung des Erfüllbarkeitsproblems basierend auf dem Filtering-Modell als korrekt und vollständig anzusehen ist.

5 DNA-basierte Implementation der aussagenlogischen Resolution

Es sind bereits mehrfach Ansätze für eine geeignete DNA-basierte Implementation von logischer Inferenz veröffentlicht worden, z.B. [WJMP00], [UHK02], [LPJ+03]. In diesem Kapitel wollen wir die Implementation der aussagenlogischen Resolution [LPJ+03] vorstellen. Die Implementation wird geeignet abgeändert, um beliebige aussagenlogische Formeln dem Unerfüllbarkeits- und Allgemeingültigkeitstest mittels DNA-basierter Resolution unterziehen zu können.

5.1 Resolution in der Aussagenlogik

Die Resolution ist ein Verfahren, um die *Unerfüllbarkeit* einer gegebenen Formel nachzuweisen. Soll die *Allgemeingültigkeit* einer gegebenen Formel gezeigt werden, so lässt sich dies auf den Unerfüllbarkeitstest zurückführen: es genügt dann, die Negation der Formel auf Unerfüllbarkeit zu untersuchen.

Zur Anwendung der Resolution sei hier vorausgesetzt, dass die Formel in Klauselform vorliegt. Die Klauseln einer solchen Formel sind konjunktiv miteinander verknüpft. Ist nun eine der Klauseln unerfüllbar, so ist auch die Formel unerfüllbar. Die leere Klausel, mithin die leere Disjunktion, die man als neutrales Element der Disjunktion betrachten kann, ist per Definition unerfüllbar. Die Idee der Resolution ist es, die gegebene Formel geeignet in eine äquivalente Formel zu transformieren, die eine leere Klausel enthält.

Bei genauerer Betrachtung der Resolution handelt es sich um einen Kalkül, der aus einer einzigen syntaktischen Umformungsregel, der so genannten *Resolutionsregel*, besteht.

Definition 5.1 (Resolutionsregel). *Sei C_1 eine Klausel, die das Literal p enthält und C_2 eine Klausel, die das Literal $\neg p$ enthält. Die Klausel R heißt* Resolvente *von C_1 und C_2 und ergibt sich durch die Ausführung folgender Schritte:*

1. *Entfernung aller Vorkommen von p aus C_1. Die so entstandene Klausel wird mit C_1' bezeichnet.*
2. *Entfernung aller Vorkommen von $\neg p$ aus C_2. Die so entstandene Klausel wird mit C_2' bezeichnet.*
3. *Die Resolvente R ergibt sich als disjunktive Verknüpfung von C_1' und C_2'.*

Die Resolvente R ist eine logische Konsequenz der beiden Ausgangsklauseln C_1 und C_2, die Aussage $C_1 \wedge C_2 \rightarrow R$ ist also allgemeingültig. Dies ist leicht nachvollziehbar:

wenn das Literal p als *wahr* interpretiert wird, muss auch C_2' *wahr* sein. Wird hingegen $\neg p$ auf *wahr* abgebildet, muss die Klausel C_1' *wahr* sein. Es ist also entweder C_1' oder C_2' *wahr*, woraus folgt, dass auch R *wahr* sein muss. Damit gilt aber auch, dass die Ausgangsklauseln nur dann beide gleichzeitig erfüllbar sind, wenn auch die Resolvente erfüllbar ist.

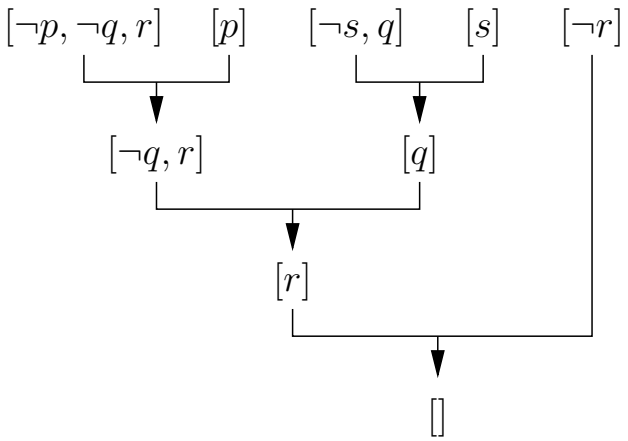
Die Resolvente kann in einer nachfolgenden Anwendung der Resolutionsregel wieder zur Bildung einer weiteren Resolvente dienen, usw. Die Anwendung der Resolutionsregel wird auch als Resolutionsschritt bezeichnet. Das Resolutionsverfahren besteht nun in der wiederholten Ausführung von Resolutionsschritten. Kann dabei die leere Klausel gebildet werden, ist die Unerfüllbarkeit der Formel nachgewiesen. Man spricht in diesem Fall auch davon, dass für die gegebene Formel ein aussagenlogischer Resolutionsbeweis gefunden werden konnte.

Das Resolutionsverfahren ist sowohl *korrekt* als auch *vollständig*. Die Korrektheit in diesem Zusammenhang sagt aus, dass falls ein Resolutionsbeweis für eine Formel gefunden werden kann, diese auch wirklich unerfüllbar ist. Die Vollständigkeit hingegen verkörpert die Aussage, dass sich zu jeder unerfüllbaren Formel auch ein Resolutionsbeweis finden lässt. Die Beweise zu diesen beiden Aussagen finden sich in der entsprechenden Literatur [Sch95, Höl01]. Alle heute bekannten Algorithmen für die Resolution weisen eine exponentielle worst-case Laufzeit auf [Sch95].

Beispiel. Es sei folgende aussagenlogische Formel in Klauselform gegeben:

$$\varphi = \langle [\neg p, \neg q, r], [p], [\neg s, q], [s], [\neg r] \rangle$$

Es soll nun mithilfe des Resolutionsverfahrens die leere Klausel [] abgeleitet werden. Das folgende Diagramm stellt eine mögliche Resolutionsableitung dar. Die Ausgangsklauseln eines Resolutionsschrittes stehen jeweils an einem Anfang, und das Ergebnis, also die Resolvente, am Ende des Pfeils.



5.2 DNA-basiertes Experiment zur aussagenlogischen Resolution

Byoung-Tak Zhang et al. beschreiben in ihrer Veröffentlichung „DNA Implementation of Theorem Proving with Resolution Refutation in Propositional Logic“ [LPJ+03] eine DNA-basierte Implementierung eines Spezialfalls der aussagenlogischen Resolution, der besonders in der logischen Inferenz und beim Theorembeweisen – beides klassische Problemstellungen der Künstlichen Intelligenz – seine Anwendung findet. Die Fragestellung lautet: Folgt die Formel ω, die aus nur einem Literal L_z besteht, aus einer gegebenen Formelmenge $\{\varphi_1, \varphi_2, \ldots, \varphi_k\}$? Dies ist äquivalent zur Frage, ob die Aussage

$$\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \rightarrow \omega \quad \text{mit } \omega = L_z$$

eine Tautologie, also allgemeingültig, ist. Dies wiederum ist gleichwertig mit dem Nachweis der Unerfüllbarkeit von

$$\neg(\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \rightarrow \omega),$$

beziehungsweise, nach elementarer Umwandlung, von

$$\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\omega.$$

Wird vorausgesetzt, dass sich $\varphi_1, \varphi_2, \ldots, \varphi_k$ in Klauselform befindet, kann dieser Unerfüllbarkeitstest mit dem allgemeinen Resolutionsverfahren gelöst werden.

Die Klauseln dieser Formel werden nun in geeigneter Weise DNA-kodiert. Im Anschluss daran wird mithilfe verschiedener molekularbiologischer Operationen ein DNA-basierter Resolutionsprozess ausgeführt. Wird letztendlich ein DNA-Molekül erzeugt, welches im übertragenen Sinne der leeren Klausel entspricht, war das Resolutionsverfahren erfolgreich, das heißt, die Zielformel ω folgt aus der gegebenen Formelmenge. Auf die einzelnen Schritte, die dazu nötig sind sowie ihre experimentelle Ausführung, wird in den folgenden Abschnitten detailliert eingegangen.

5.2.1 DNA-Kodierung der Klauseln

Eine Klausel mit n Literalen wird durch ein verzweigtes DNA-Molekül mit n Verzweigungen repräsentiert. Ausgenommen davon sind Klauseln, die nur ein einziges Literal beinhalten: sie werden mittels eines Hairpin-Moleküls dargestellt. Jeder Verzweigungsarm der Moleküle – auch der des Hairpin-Moleküls – steht stellvertretend für ein Literal der repräsentierten Klausel und besitzt ein sticky-Ende.

Die Einzelstrangüberhänge an den Verzweigungsarmen haben jeweils die gleiche Länge und sind eindeutig für jedes Literal. Die einem positiven Literal zugeordnete Sequenz ist jedoch entgegengesetzt komplementär zur Sequenz des negativen Literals der gleichen Variable. Wird also das Literal p durch die Sequenz a verkörpert, so wird das Literal $\neg p$ durch die zu a komplementäre Sequenz $\tilde{a}$ am sticky-Ende dargestellt. Die restlichen Sequenzen der selbst-hybridisierten Teile der Moleküle, die nicht der Repräsentation

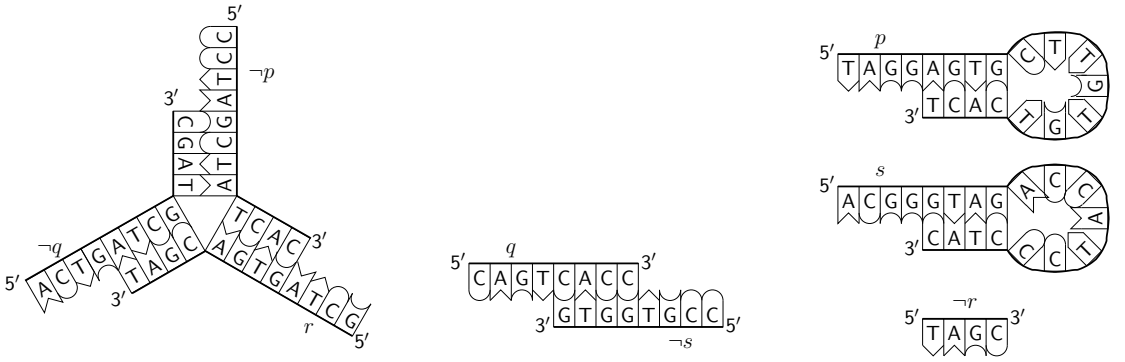


Abbildung 5.1: *Eine Kodierung der Formel* $(\neg p \vee \neg q \vee r) \wedge p \wedge (\neg s \vee q) \wedge s \wedge \neg r$*, wobei* $\neg r$ *das Zielliteral ist.*

der Literale sondern nur der geforderten Molekülstruktur an sich dienen, sind beliebig wählbar.

Die Zielformel ω, die wegen der verwendeten Detektionsmethode nur aus einem einzigen Literal bestehen darf, wird nur durch einen Einzelstrang jener Sequenz repräsentiert, die in der Kodierung für dieses Literal steht. Diese Sequenz soll für die weiteren Ausführungen mit *Zielsequenz* bezeichnet werden. In der Abbildung 5.1 ist eine mögliche Kodierung der Beispielformel

$$(\neg p \vee \neg q \vee r) \wedge p \wedge (\neg s \vee q) \wedge s \wedge \neg r$$

dargestellt.

5.2.2 Ablauf der Resolution

Die DNA-Moleküle, die nun also repräsentativ für die einzelnen Klauseln der Formel stehen, werden für die weitere Bearbeitung in ein Tube gegeben. Dabei kommt jedes der DNA-Moleküle millionenfach im Tube vor, um einerseits eine möglichst vollständige Resolventenbildung zu ermöglichen und andererseits die Fehlerwahrscheinlichkeit zu reduzieren. Die Resolution selbst wird durch die Anwendung unterschiedlicher molekularbiologischer Operationen realisiert. Die generelle Vorgehensweise lässt sich wie folgt unterteilen:

1. Zentrifugieren des Tubeinhalts.
2. Hybridisierung zur Realisierung des Resolutionsbeweises.
3. Ligation, um kovalente Bindungen zwischen den hybridisierten Molekülen herzustellen.

4. Polymerase-Kettenreaktion (PCR) zur Vervielfältigung der Ligationsprodukte. Es werden nur diejenigen Moleküle vervielfältigt, die einem gültigen Beweis, im übertragenen Sinne also der leeren Klausel, entsprechen.

5. Gel-Elektrophorese zur Ergebnisausgabe.

Das Zentrifugieren hat eine möglichst gute Durchmischung des Tubeinhalts zum Ziel. Der eigentliche Resolutionsvorgang verbirgt sich hinter Schritt 2: zwei DNA-kodierte Klauseln hybridisieren genau an ihren komplementären Sequenzen und bilden damit ihre Resolvente. Der Resolutionsschritt zwischen zwei Klauseln wird somit abgebildet auf die Hybridisierung zwischen den diese Klauseln repräsentierenden DNA-Molekülen. Dies geschieht an genau den Stellen, an denen die Moleküle über komplementäre Sequenzen und mithin die korrespondierenden Klauseln über resolvierbare Literale verfügen.

Man beachte, dass die Hybridisierung massiv parallel, also faktisch gleichzeitig auf alle im Tube befindlichen DNA-Moleküle einwirkt. Der gesamte Resolutionsvorgang wird in nur einem Schritt realisiert. Die benötigte Zeit für die Hybridisierung beträgt ca. zwei Stunden, ist jedoch praktisch unabhängig von der Anzahl der Moleküle im Tube.

Die folgende Ligation bewirkt die Ausbildung von kovalenten Bindungen zwischen den frisch hybridisierten Molekülen. Die Ligation dient als vorbereitender Schritt für die sich anschließende PCR und hat einen Zeitbedarf von mehreren Stunden im Labor. Wurde in Schritt 2 bei der Hybridisierung die leere Klausel gebildet, so manifestiert sich dies nun in einem zusammenhängenden DNA-Einzelstrang, der mit der Zielsequenz beginnt, mit der komplementären Zielsequenz endet und an einigen Stellen mit sich selbst hybridisiert ist. Die Existenz eines DNA-Strangs mit diesen Eigenschaften zeigt also an, dass die Resolution erfolgreich verlaufen ist.

Die auf die Ligationsprodukte ausgeführte PCR verwendet als einzigen Primer die Zielsequenz. Dies garantiert, dass nur solche Stränge exponentiell vervielfältigt werden, die sowohl die Zielsequenz, als auch die komplementäre Zielsequenz aufweisen. Dies sind genau jene Stränge, die die leere Klausel verkörpern. Der Prozess der PCR ist am Beispiel in Abbildung 5.3 wiedergegeben. Für die Durchführung einer PCR mit den vorgestellten Bedingungen muss rund eine Stunde im Labor veranschlagt werden.

Abschließend werden die DNA-Stränge durch die Gel-Elektrophorese ihrer Länge nach separiert. Bildet sich für eine bestimmte Länge eine sehr starke Bande aus, ist dies auf eine hohe Konzentration von DNA-Strängen mit dieser Länge zurückzuführen – eine Konzentration, die nur durch die Vervielfältigung durch die vorhergehende PCR bedingt sein kann. Eine starke Bande bei der Gel-Elektrophorese deutet also auf einen gefundenen Resolutionsbeweis hin. Entstehen während der Resolution mehrere leere Klauseln, so wird jede von ihnen durch die PCR amplifiziert und jede von ihnen bildet dann in der Gel-Elektrophorese je nach Länge ihre eigene Bande. So kann im Idealfall jede Resolutionswiderlegung für eine Formel entdeckt werden.

5.2.3 Laborpraktische Lösung einer Instanz

Im Labor wurde durch die Autoren von [LPJ+03] eine kleine Instanz der Resolution praktisch gelöst und damit die prinzipielle Durchführbarkeit ihrer Idee untermauert.

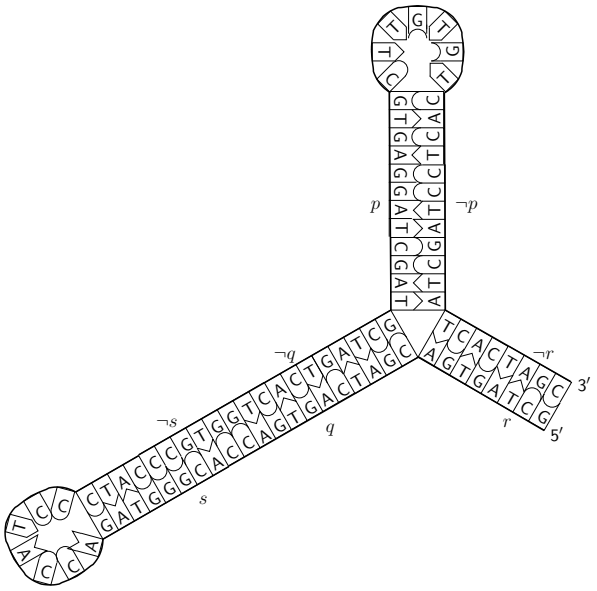


Abbildung 5.2: *Nach der Hybridisierung der Stränge aus Abbildung 5.1 hat sich ein Molekül gebildet, das der leeren Klausel entspricht*

Das Problem bestand in der Fragestellung, ob q aus der Formelmenge $p \wedge (p \rightarrow q)$ folgt. Nach Umformung ergibt sich eine Formel in Klauselform $\langle [p], [\neg p, q], [\neg q] \rangle$, deren Unerfüllbarkeit mit dem DNA-basierten Resolutionsverfahren nachgewiesen wurde. Die bei der Kodierung der Klauseln verwendeten DNA-Sequenzen sind in Abbildung 5.4(a) dargestellt.

Nach der Kodierung folgten die Schritte Hybridisierung, Ligation, PCR und Gel-Elektrophorese wie beschrieben. Die Abbildung 5.4(b) zeigt das durch die Hybridisierung entstandene, der leeren Klausel entsprechende Molekül. Als Primer für die PCR hätte normalerweise die Sequenz des Literals $\neg q$ dienen sollen. Diese ist jedoch mit nur fünf Nukleotiden für eine Primerverwendung zu kurz, so dass hier der untere Teil des zur Klausel $[\neg p, q]$ gehörenden DNA-Doppelstrangs als Primersequenz eingesetzt wurde. Die Auswertung der Gel-Elektrophorese ergab eine deutlich sichtbare Bande bei 23 bp (Basenpaaren). Da das doppelsträngige Molekül, das die leere Klausel für diesen Resolutionsbeweis repräsentiert, über 23 bp verfügt (siehe auch Abbildung 5.4(b)), wurde hier also das richtige Molekül mithilfe der PCR vervielfältigt und durch die Gel-Elektrophorese angezeigt. Die genauen Angaben über Prozessparameter wie Temperaturverläufe, Flüssigkeitskonzentrationen oder Zeitbedarf der einzelnen Schritte können [LPJ+03] entnommen werden.

5.2.4 Probleme und Einschränkungen

Laut der Autoren ist aufgrund verschiedener Umstände die Handhabung der Polymerase-Kettenreaktion sehr schwierig. Bedingt durch die Verwendung von verzweigten DNA-

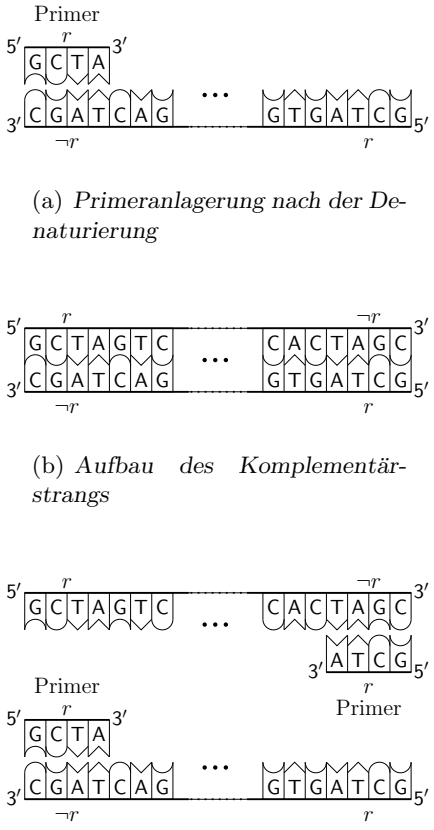
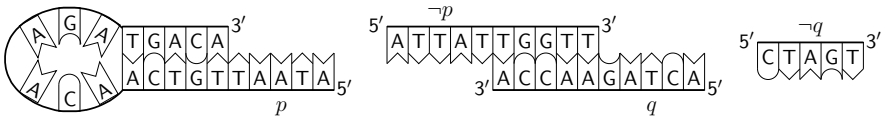


Abbildung 5.3: *Ablauf der Polymerase-Kettenreaktion (PCR) mit dem Beispiel-Molekül der Abbildung 5.2*

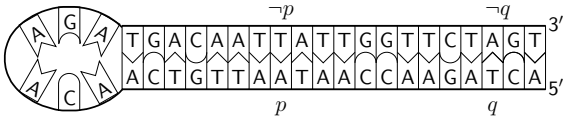
Molekülen sowie Hairpins sind Teilsequenzen des in der PCR zu amplifizierenden DNA-Einzelstrangs zu sich selbst komplementär. Dies könnte während der Primeranlagerung zur Hybridisierung einzelner Teilsequenzen führen und damit den Fortgang der Kettenreaktion stoppen. In diesem Fall würde die Gel-Elektrophorese trotz erfolgreicher Resolutionswiderlegung möglicherweise keine Bande formen und somit ein falsches Ergebnis anzeigen.

Tritt die Zielvariable mehrfach in einer der Klauseln oder Resolventen auf, so könnte dies zu einem Einzelstrang führen, der mit der PCR vervielfältigt werden kann und somit zum Anzeigen einer Resolutionswiderlegung führt, obwohl unter Umständen keine leere Klausel vorliegt. In Abbildung 5.5 ist ein Beispiel dieses Problems dargestellt. Abhilfe könnte hier die Verwendung einer anderen Kodierungs- oder Detektionsmethode schaffen.

Es kann außerdem zu so genannten *Self-Loop-Strukturen* kommen. Existieren in einem



(a) *Kodierung der Klauseln*



(b) *Nach der Hybridisierung*

Abbildung 5.4: *Im Experiment verwendete DNA-Moleküle vor und nach der Hybridisierung.*

DNA-Molekül zwei Verzweigungsarme, die an ihrem sticky-Ende komplementär zueinander sind, so könnten sie sich während der Hybridisierung miteinander verbinden und damit die Hybridisierung mit anderen Molekülen an diesen Stellen verhindern. Jedoch tritt dies nur dann auf, wenn in der korrespondierenden Klausel oder Resolvente gleichzeitig das positive und das negative Literal der gleichen Variable auftreten – jedoch wäre sie dann unter allen Interpretationen *wahr*, und bräuchte somit für die weitere Resolution nicht mehr betrachtet werden.

Ein weiteres Problem, das in [LPJ+03] nicht angesprochen wird, betrifft das mehrfache Vorkommen eines beliebigen Literals in einer Klausel oder Resolvente. Bei einem Vergleich eines regulären Resolutionsschritts mit seiner molekularbiologischen Entsprechung – der Hybridisierung zweier entsprechender DNA-Moleküle – wird ein wichtiger Unterschied deutlich: während des regulären Resolutionsschritts werden *alle* Vorkommen des positiven Literals der einen Klausel und *alle* Vorkommen des negativen Literals der anderen Klausel entfernt, und dann beide Klauseln disjunktiv zusammengeführt. Bei der Hybridisierung zweier DNA-Moleküle hingegen wird immer nur jeweils ein sticky-Ende „geschlossen“ – im übertragenen Sinne also nur jeweils ein Vorkommen des Literals innerhalb der Klausel entfernt. In jener Situation, in der ein Literal mehrfach in einer Klausel vorkommt, das entgegengesetzte Literal aber nur einfach in den restlichen Klauseln vorkommt, so stellt dies auch in der molekularen Variante kein Problem dar: gegeben seien zum Beispiel die Klauseln $[p, p], [\neg p]$. Hier würde sich jeweils ein Vorkommen des Moleküls, das die Klausel $[\neg p]$ repräsentiert, an jedes der sticky-Enden des entsprechenden Moleküls der Klausel $[p, p]$ binden und somit korrekterweise die leere Klausel bilden. Anders sieht dies jedoch aus, wenn in beiden Klauseln die entsprechenden Literale doppelt vorkommen: zum Beispiel $[p, p], [\neg p, \neg p]$. Hier würde immer jeweils eines

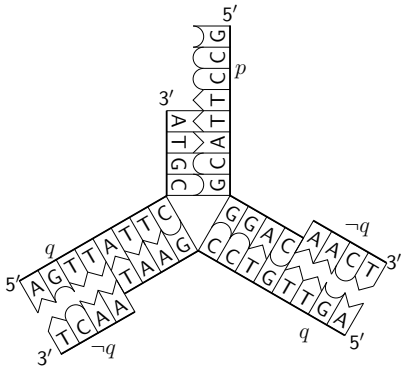


Abbildung 5.5: *Nach der Hybridisierung und Ligation erfüllt der untere Einzelstrang bereits fälschlicherweise die Bedingungen der leeren Klausel. Diese Situation tritt immer dann auf, wenn sich die Zielvariable mehrmals in einer Klausel oder Resolvente befindet.*

der sticky-Enden offen bleiben. Auch wenn man solche Klauseln innerhalb der gegebenen Formel ausschließt, behebt dies nicht das Problem, da sich eine solche Situation später innerhalb der Resolution ergeben kann: gegeben seien beispielsweise die Klauseln $[\neg p, \neg q], [p, \neg q], [\neg p, q], [p, q]$. Im DNA-basierten Verfahren entstehen unter anderem genau die problematischen Resolventen $[\neg q, \neg q], [q, q]$. Eine Lösung für dieses Problem scheint unter der gegebenen Kodierungs- und Detektionsmethode nicht einfach möglich zu sein.

5.3 Änderung der Detektionsmethode

Das bis hierhin vorgestellte Verfahren [LPJ+03] ist gegenüber dem regulären Resolutionsverfahren stark eingeschränkt: es können nur Formeln der Form

$$\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\omega \quad \text{mit } \omega = L_z$$

wobei die Zielformel ω nur aus einem einzigen Literal besteht, behandelt werden. Dies liegt an der verwendeten Detektionsmethode, die aus der beschriebenen Kombination von Ligation, PCR und Gel-Elektrophorese besteht. Das Literal L_z der Zielformel dient in der PCR als Primer. Wünschenswert wäre natürlich ein weniger eingeschränktes Verfahren. Tatsächlich ist dies durch den Austausch der Detektionsfunktion möglich.

Um das Verfahren auf beliebige aussagenlogische Formeln in Klauselform zu erweitern, also die Unerfüllbarkeit einer Formel der Form

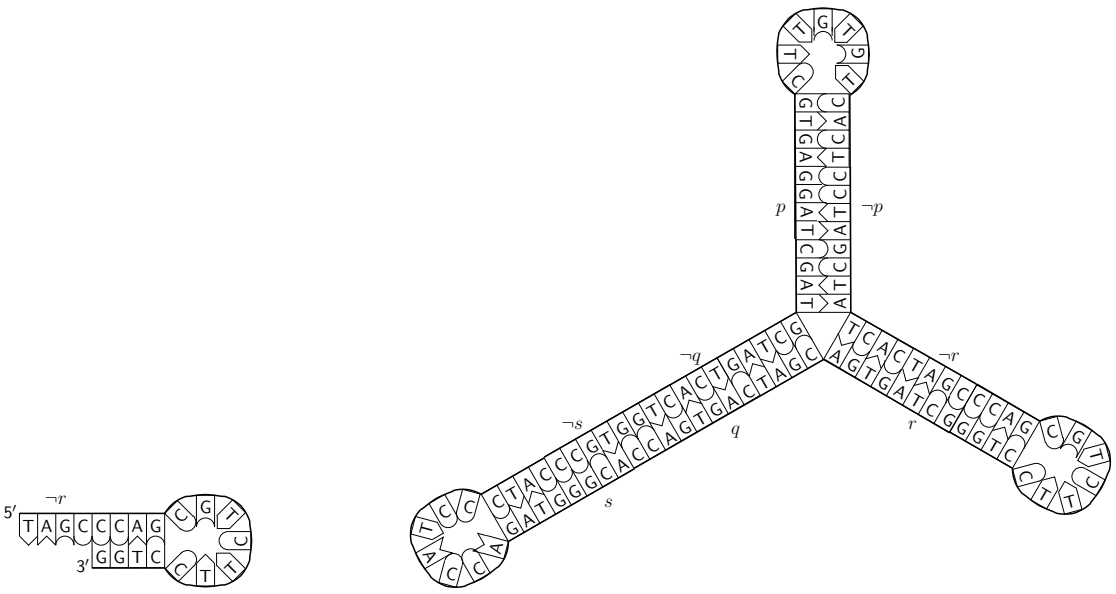
$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

DNA-basiert nachzuweisen, gehen wir wie folgt vor. Die Kodierungsmethode wird wie vorgestellt beibehalten. Es gibt in der Formel nun keine Zielformel mehr, so dass alle Klauseln entweder durch Hairpin- oder verzweigte DNA-Moleküle repräsentiert werden.

Eine leere Klausel stellt sich dann als selbsthybridisierter zirkulärer DNA-Einzelstrang dar. Als Beispiel hierfür soll noch einmal die Formel

$$(\neg p \vee \neg q \vee r) \wedge p \wedge (\neg s \vee q) \wedge s \wedge \neg r$$

aus den vorangegangenen Abschnitten dienen. Eine mögliche Kodierung kann die in Abbildung 5.1 dargestellte sein – nur dass nun die Klausel $(\neg r)$ auch, so wie jede andere einelementige Klausel, mittels eines Hairpin-Moleküls kodiert wird (siehe Abbildung 5.6(a)). Das Hybridisierungsergebnis entspricht der leeren Klausel als zirkulärer DNA-Strang, wie in Abbildung 5.6(b) dargestellt.



(a) *Kodierung der Klausel* $(\neg r)$

(b) *Zirkulärer DNA-Strang repräsentiert die leere Klausel*

Abbildung 5.6: *Anwendung der neuen Detektionsmethode auf das betrachtete Beispiel*

Die leeren Klauseln müssen nun innerhalb einer neuen Detektionsmethode als solche erkannt werden. In [UHK02] wird ein Verfahren zur DNA-basierten Inferenz von Hornklauseln vorgestellt. Für die Kodierung der Hornklauseln werden ebenfalls Hairpin- und verzweigte DNA-Moleküle eingesetzt. Interessant ist jedoch die verwendete *Detect*-Operation: sie ermöglicht genau die gewünschte Erkennung von zirkulären DNA-Strängen durch die Nutzung von Exonuklease-Enzymen. Exonukleasen bauen in fortgesetzten Reaktionen DNA-Stränge entweder vom $3'$ oder $5'$-Ende her ab. Da natürlich der zirkuläre Strang kein Ende besitzt, würde er von diesem Enzym unangetastet im Reaktionsgefäß zurückbleiben. Leider wurde die Operationen in [UHK02] nicht durch ein Laborexperiment getestet, jedoch kann die Exonuklease-Reaktion im Allgemeinen als gut studiert angesehen werden.

Zusammenfassend besteht die auf unser DNA-basiertes Resolutionsverfahren angepasste neue Detektionsmethode für leere Klauseln aus den Schritten:

- Ligation zur Ausbildung der kovalenten Bindungen zwischen den hybridisierten Molekülen
- Exonuklease-Reaktion zum Verdau aller nicht-zirkulärer DNA-Moleküle
- Test auf Vorhandensein intakter DNA

Mit der Verwendung dieser Detektionsmethode ist jedoch nicht nur eine Erweiterung auf beliebige Formeln möglich. Die ersten beiden Probleme, die im Abschnitt 5.2.4 auf Seite 40 erläutert wurden, sind beide mit der Verwendung der PCR verbunden und treten somit hier nicht mehr auf. Self-Loop-Strukturen könnten hier hingegen ebenso auftreten und eventuell durch die Formung zirkulärer DNA-Stränge falsche Ergebnisse erzeugen. Auch bleibt das geschilderte Problem mit Mehrfachvorkommen von Literalen in Klauseln oder Resolventen bestehen.

5.4 Zusammenfassung des Modells

Das entwickelte Modell zur DNA-basierten Resolution soll hier noch einmal zusammenfassend formal beschrieben werden.

Das Modell dient dem Nachweis der Unerfüllbarkeit beliebiger aussagenlogischer Formeln in Klauselform. Die Klauseln der Formel werden wie folgt DNA-kodiert: Klauseln, die nur ein Literal enthalten, werden durch Hairpins, alle anderen Klauseln durch verzweigte DNA-Moleküle mit n Verzweigungsarmen, wobei n der Anzahl der Literale in der Klausel entspricht, repräsentiert. Jeder Verzweigungsarm und jedes Hairpin-Molekül verfügt über ein sticky-Ende, dessen Einzelstrangüberhangssequenz eindeutig ein Literal der Klausel repräsentiert. Die Sequenzen entgegengesetzter Literale sind entgegengesetzt komplementär. Die so synthetisierten DNA-Stränge werden im Tube T für die weitere Bearbeitung bereitgestellt.

Definition 5.2. *Das Modell besteht aus folgenden Basisoperationen:*

- Hybridize. *Sei T ein Tube. Die Operation* `hybridize(T)` *erzeugt ein Tube, das die miteinander unter Beachtung der spezifischen Basenpaarung hybridisierten Moleküle enthält.*
- Detect. *Sei T ein Tube. Die Operation* `detect(T)` *gibt 'ja' aus, wenn T mindestens einen vollständig zirkulären DNA-Einzelstrang enthält, ansonsten gibt sie 'nein' aus.*

Die Operation *Hybridize* wird durch ihre molekularbiologische Entsprechung realisiert. Die molekularbiologische Realisierung der *Detect*-Operation besteht in einer Kombination der Operationen Ligation, Exonuklease-Behandlung sowie Test auf Vorhandensein von DNA.

Für eine gegebene aussagenlogische Formel φ in Klauselform lässt sich ein Resolutionsbeweis durch die aufeinanderfolgende Ausführung der beiden Modelloperationen führen:

`detect(hybridize(`T`))`

Hierbei enthalte das Tube T zu Beginn die entsprechend der Kodierungsvorschrift synthetisierten und die Klauseln von φ repräsentierenden DNA-Moleküle.

6 Verifikation der DNA-basierten Resolution

6.1 Spezifikation des Modells in PVS

Für eine Spezifikation in PVS muss zuerst eine geeignete Datenstruktur für die DNA-Moleküle im Tube gefunden werden. Die Moleküle für die Klauselrepräsentation haben eine *nichtlineare* Struktur (Hairpins, verzweigte DNA-Stränge), von der wir jedoch abstrahieren wollen. Das Charakteristische sind ihre sticky-Enden an den Verzweigungsarmen, über die die Literale kodiert werden. Sie werden dargestellt als geordnetes Paar von aussagenlogischer Variable, einem Marker für die Komplementarität und einem Stickymarker, um anzuzeigen, ob die Sequenz noch „offen“ ist, oder ob sich schon ein anderes Molekül dort angelagert hat, also „geschlossen“ ist:

```
MARKERS: TYPE = {positive, negative}
STICKYMARKER: TYPE = {open, closed}
PLITN: TYPE = [PVAR, MARKERS, STICKYMARKER]          % Verbindungsstelle
```

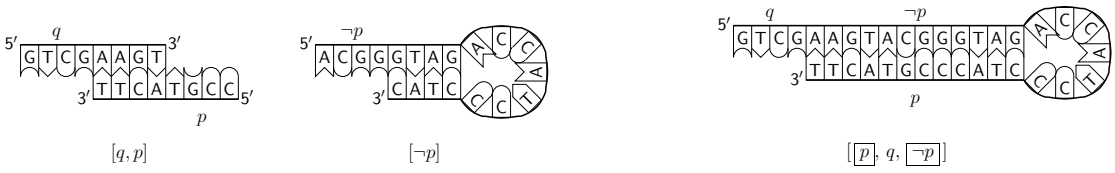
Für die weitere Betrachtung wollen wir dieses 3-Tupel `PLITN` als *Verbindungsstelle* bezeichnen. Eine Verbindungsstelle ist *offen*, wenn der Stickymarker den Wert `open` annimmt, ansonsten ist sie *geschlossen*. Jede Verbindungsstelle steht für ein Literal einer bestimmten aussagenlogischen Variable `PVAR` und kann positiv oder negativ sein. Zwei Verbindungsstellen sind komplementär zueinander, wenn sie die gleiche aussagenlogische Variable enthalten, jedoch unterschiedliche Werte des Typs `MARKERS`.

Ein Klausel-repräsentierendes DNA-Molekül, als `AGGREGATE` bezeichnet, kann nun einfach als Liste solcher Verbindungsstellen angesehen werden. Der Inhalt eines Tubes ist eine Liste von Aggregaten, siehe 12.

12

```
AGGREGATE: TYPE = list[PLITN]
TUBECONTENT: TYPE = list[AGGREGATE]
```

Die Hybridisierung zweier Aggregate wird durch die Funktion `hybridize` mit der Typung `[AGGREGATE, AGGREGATE, PVAR -> TUBECONTENT]` nachgebildet. Eine solche Hybridisierung ist nur möglich, wenn beide zueinander komplementäre und offene Verbindungsstellen beinhalten:



(a) (b)

Abbildung 6.1: *Darstellung der Moleküle als Listen in PVS (a) vor und (b) nach einer Hybridisierung. Die geschlossenen Verbindungsstellen werden hier durch Kästchen verdeutlicht.*

13

```
hybridize(AGG_Pos, AGG_Neg, PV): TUBECONTENT =
    IF member((PV, positive, open), AGG_Pos) AND
        member((PV, negative, open), AGG_Neg)
      THEN cons(append(agg_shift(close_lit(AGG_Pos, PV, positive),
                                 (PV, positive, closed), null),
                       agg_shift(close_lit(AGG_Neg, PV, negative),
                                 (PV, negative, closed), null)),
                null)
    ELSE cons(AGG_Neg, cons(AGG_Pos, null))
    ENDIF
```

Die Zusammenführung der beiden Aggregate `AGG_Pos` und `AGG_Neg` wird durch ein `append` ermöglicht. Hierbei werden mit `close_lit` zuvor *alle* positiven Vorkommen von Verbindungsstellen mit der Variable `PV` in `AGG_Pos`, und *alle* negativen in `AGG_Neg` „geschlossen“, so dass dort keine anderen Moleküle anlagern können. Man beachte, dass hiermit das Problem der Mehrfachvorkommen von Literalen, das im Abschnitt 5.2.4 erläutert wurde, einfach übergangen wird. Dies ist notwendig, da ansonsten die Vollständigkeit des Verfahrens nicht zu beweisen wäre, was später noch deutlich wird. Zur Definition der Funktion `close_lit` siehe S. 65 im Anhang.

Die Funktion `agg_shift` ist für die Beweise und Betrachtungen zu vernachlässigen: sie nimmt nur eine Umordnung der Elemente in den Aggregaten vor. So wird das geschlossene Literal an den Kopf der jeweiligen Liste gestellt. Dies wurde implementiert, um die Molekülgestalt der Ausgangsaggregate (d.h. aus wie vielen Literalen sie bestehen) nach ihrer Hybridisierung sichtbar bleibt. Die Definition von `agg_shift` ist auf S. 65 ersichtlich. In der Abbildung 6.1 ist eine beispielhafte Anwendung der `hybridize`-Funktion abgebildet.

Eine leere Klausel ist im Sinne der verwendeten Kodierung ein Aggregat, das keine sticky-Verzweigungsarme mehr aufweist und somit einen zirkulären DNA-Strang ergibt. Für unsere Listen bedeutet dies, dass das Aggregat keine offenen Verbindungsstellen mehr beinhaltet. Die Definition der `detect`-Operation sucht in einem Tube nach genau

solch einem Aggregat und gibt in diesem Fall `YES` zurück, im anderen Fall `NO`:

```
DECISION: TYPE = {YES, NO}
AGG: VAR AGGREGATE
detect(TC): DECISION =
    IF (EXISTS AGG: member(AGG, TC) AND is_compl_closed(AGG)) THEN YES
    ELSE NO
    ENDIF
```

Das Prädikat `is_compl_closed` prüft, ob alle Verbindungsstellen des übergebenen Aggregats geschlossen sind. Die Definition ist auf S. 66 des Anhangs zu finden.

Als Eingabe für die Resolution dient eine beliebige aussagenlogische Formel in Klauselform. Die entsprechenden Spezifikationen für aussagenlogische Formeln und Interpretationen wurden mit Änderungen aus [DMJ02] übernommen. Als Typ stellt sich eine Formel als Liste von Klauseln dar, die selbst wieder Listen von Literalen sind, siehe 14. Ein Literal ist hierbei das Paar eines Markers positiv/negativ und einer aussagenlogischen Variable.

14

```
PLIT: TYPE = [MARKERS, PVAR]
PCL: TYPE = list[PLIT]
PFORM_fnc: TYPE = list[PCL]
```

Eine aussagenlogische Interpretation spezifizieren wir als Abbildung von den aussagenlogischen Variablen auf die Menge der Wahrheitswerte $0, 1$.

```
TRUTH_VALUES: TYPE = {zero, one}
VV: VAR TRUTH_VALUES
opp_value(VV): TRUTH_VALUES = IF VV = zero THEN one ELSE zero ENDIF
VALUATIONS: TYPE = [PVAR -> TRUTH_VALUES]
```

Für ein gegebenes Literal `PL` ergibt sich der über die Interpretation `PI` vom Typ `VALUATIONS` assoziierte Wahrheitswert wie folgt:

```
PVal(PI, PL): TRUTH_VALUES =
    IF plit_positive?(PL) THEN PI(PVar(PL))
    ELSE opp_value(PI(PVar(PL)))
    ENDIF
```

Ist also das Literal `PL` positiv, wird der Wahrheitswert zurückgegeben, der von `PI` für die Variable des Literals definiert ist, ansonsten genau der entgegengesetzte Wert. Das Prädikat `plit_positive?` und die Funktion `PVAR`, die die Variable eines Literals zurückliefert, sind auf S. 61 definiert.

Die Funktion `PVal` ist auch für Klauseln und für Formeln definiert: einer Klausel wird dann der Wahrheitswert 1 zugeordnet, wenn ein in ihr enthaltenes Literal auf 1 abgebildet wird. Einer Formel hingegen wird der Wahrheitswert 1 zugeordnet, wenn

alle in ihr enthaltenen Klauseln auf 1 abgebildet werden. Die entsprechenden rekursiven Funktionsdefinitionen von `PVal` befinden sich auf S. 64.

Anhand der Struktur der zu Beginn gegebenen aussagenlogischen Formel muss der Tubeinhalt gebildet werden. Für jede Klausel der Formel erstellen wir ein Aggregat, welches diese Klausel repräsentiert. Es wird jedes Literal der Klausel um einen Stickymarker erweitert und bildet somit eine Verbindungsstelle. Diese Umwandlung von Formel in Tubeinhalt leistet die Funktion `build_tube`, zu finden auf S. 68. In der laborpraktischen Durchführung würde sich jedes Aggregat viele Millionen mal im Tube befinden. Wir betrachten repräsentativ jeweils nur ein Exemplar eines jeden Aggregats.

Bisher wurde die Hybridisierungs-Operation in der Spezifikation nur zwischen zwei einzelnen Aggregaten betrachtet, siehe Definition von `hybridize` in 13. In der laborpraktischen Implementation wirkt die Hybridisierung aber auf alle Moleküle eines Tubes ein. Da sich jedes DNA-Molekül millionenfach im Tube befindet, kann mit hoher Wahrscheinlichkeit davon ausgegangen werden, dass sich alle möglichen Verbindungen zwischen kompatiblen DNA-Molekülen auch ausbilden. Sicher wird es Moleküle geben, an deren sticky-Enden kein kompatibles Partnermolekül anlagern konnte, weil in der näheren räumlichen Umgebung eben zufällig keines vorhanden war. Dies ist jedoch kein Problem, da es mit hoher Wahrscheinlichkeit andere Moleküle des gleichen Typs gibt, bei denen sich diese Verbindung ausbildete. Wir wollen somit davon ausgehen, dass sich *alle* möglichen Kombinationen ausbilden und im Tube vorhanden sind. Ein weiterer wichtiger Punkt ist die Parallelität der Hybridisierung: die Operation wirkt echt parallel auf alle DNA-Stränge. In der PVS-Spezifikation müssen wir diesen Prozess geeignet serialisieren.

Um also den Prozess der Hybridisierung geeignet nachzubilden, wollen wir wie folgt vorgehen. Zuerst bilden wir eine Liste aller in der gegebenen Formel vorkommenden aussagenlogischen Variablen. Diese Liste wird dann elementeweise abgearbeitet. Für die aktuell betrachtete Variable `PV` werden nun alle Aggregate in einer Liste aufgesammelt, die `PV` innerhalb einer positiven Verbindungsstelle beinhalten. Eine weitere Liste mit allen Aggregaten, die `PV` im Rahmen einer negativen Verbindungsstelle aufweisen, wird ebenfalls gebildet. Jedes Aggregat der einen Liste ist nun kompatibel mit jedem Aggregat der anderen Liste, so dass wir diese kombinatorisch mittels `hybridize` hybridisieren können. Die Hybridisierungsergebnisse entsprechen den Resolventen und werden dem Tube wieder hinzugefügt. Dieser Prozess wird für alle Variablen der Formel vorgenommen.

Um diese Strategie in PVS umzusetzen, sind einige Funktionsdefinitionen nötig, die nun vorgestellt werden. Den Anfang bildet die Funktion `resolution`:

15
```
PF: VAR PFORM_fnc

resolution(PF): TUBECONTENT =
    resolution(PVarSingle(PVar(PF)), build_tube(PF))
```

Die Funktion `resolution` besitzt als Argument die aussagenlogische Formel `PF` und ruft eine andere Funktion gleichen Namens auf (sie ist überladen definiert). Das erste Argument der aufgerufenen Funktion ist eine Liste aller Variablen, die in `PF` vorkommen. Dies wird durch die Funktionen `PVar` und `PVarSingle` bewerkstelligt. `PVar` erstellt

rekursiv eine Liste mit den Variablen und `PVarSingle` sorgt dafür, dass jede Variable in der Liste nur genau einmal vorkommt. Die entsprechenden Definitionen sind auf S. 62 zu finden. Durch die bereits angesprochene Funktion `build_tube` wird aus `PF` der Tubeinhalt vom Typ `TUBECONTENT` gebildet und als zweites Argument an `resolution` übergeben. Die aufgerufene Funktion selbst ist wie folgt definiert:

```
resolution(LPV, TC): RECURSIVE TUBECONTENT =
  CASES LPV
    OF null: TC,
       cons(h, t):
         resolution(t,
                    append(TC,
                           hybridize(aggfilter(TC, h, positive),
                                     aggfilter(TC, h, negative),
                                     h)))
    ENDCASES
   MEASURE length(LPV)
```

Sie arbeitet die Liste der Variablen `LPV` vom Typ `list[PVAR]` rekursiv ab. Das zweite Argument `TC` ist vom Typ `TUBECONTENT` und beinhaltet sämtliche Aggregate. Über die als überladen definierte Funktion `hybridize` werden kombinatorisch alle möglichen Hybridisierungsprodukte anhand der behandelten Variable `h` gebildet. Hierfür ist es notwendig, über die Funktion `aggfilter` zuerst alle Aggregate getrennt aufzusammeln, die `h` in Form einer positiven bzw. negativen offenen Verbindungsstelle aufweisen. Die Funktion `aggfilter` befindet sich in ihrer Definition auf S. 68. Wurden nun alle Hybridisierungsprodukte von `h` gebildet, werden diese dem Tube wieder hinzugefügt und ein neuer Durchlauf startet. Wie aus den beiden über `aggfilter` gebildeten Listen nun kombinatorisch die „Resolventen“ gebildet werden, ist über die Funktion `hybridize` des Typs `[TUBECONTENT, TUBECONTENT, PVAR -> TUBECONTENT]` definiert:

```
TC_Pos, TC_Neg: TUBECONTENT
PV: PVAR

hybridize(TC_Pos, TC_Neg, PV): RECURSIVE TUBECONTENT =
  CASES TC_Pos
    OF null: null,
       cons(h, t):
         append(hybridize(h, TC_Neg, PV), hybridize(t, TC_Neg, PV))
    ENDCASES
   MEASURE length(TC_Pos)
```

Diese Funktion ist einfach dafür zuständig, dass jedes Element von `TC_Pos` mit jedem Element `TC_Neg` hybridisiert werden. Da sowohl `TC_Pos` als auch `TC_Neg` über `aggfilter` gebildet wurden, sind sie in jedem Falle kompatibel zueinander. Die Funktionsweise besteht in einem rekursiven Abstieg in der Liste `TC_Pos`. Für jedes Element `AGG_Pos` von `TC_Pos` wird eine weitere `hybridize` Funktion aufgerufen, die dieses `AGG_Pos` mit den

Elementen von `TC_Neg` mittels der in 13 vorgestellten `hybridize` Funktion hybridisiert.

```
AGG_Pos: AGGREGATE

hybridize(AGG_Pos, TC_Neg, PV): RECURSIVE TUBECONTENT =
  CASES TC_Neg
    OF null: null,
       cons(h, t):
         append(hybridize(AGG_Pos, h, PV), hybridize(AGG_Pos, t, PV))
    ENDCASES
   MEASURE length(TC_Neg)
```

Hier schließt sich somit der Kreis zur in 13 vorgestellten Hybridisierung zweier Aggregate. Jedes Element von `TC_Neg` wird mit dem Aggregat `AGG_Pos` hybridisiert.

Zusammenfassend soll noch einmal kurz auf den Gesamtablauf eingegangen werden: über ein `detect(resolution(PF))` würde also das Entscheidungsverfahren für die DNA-basierte Resolution in Gang gesetzt werden. Kombinatorisch werden nun alle möglichen Hybridisierungsprodukte gebildet, was im übertragenen Sinne einer Resolventenbildung entspricht. Wenn dabei ein Aggregat entsteht, das nur geschlossene Verbindungsstellen besitzt, dies entspricht in der regulären Resolution der leeren Klausel, würde die Funktion `detect` den Wert `YES` zurückliefern, ansonsten den Wert `NO`.

6.2 Nachweis der Korrektheit und Vollständigkeit

Durch die Nähe des spezifizierten Verfahrens zur regulären Resolution orientiert sich der Beweis der Korrektheit und Vollständigkeit an den Beweisideen für den Korrektheits- und Vollständigkeitsnachweis der regulären Resolution [Höl01]. Aufgrund des begrenzten Platzes kann nicht bis ins Detail auf die Beweisführung eingegangen werden, sondern diese nur in den wichtigsten Grundzügen dargestellt werden. Sämtliche Beweise der Lemmata und Theoreme können jedoch mittels der zur Verfügung stehenden PVS-Quellen ausführlich nachvollzogen werden.

6.2.1 Korrektheit

Das entsprechende Theorem zur Korrektheit des spezifizierten Verfahrens lautet in PVS Syntax:

16

```
soundness: THEOREM
  detect(resolution(PF)) = YES IMPLIES (FORALL PI: PVal(PI, PF) = zero)
```

Wird also nach Ausführung der Funktion `resolution` auf eine aussagenlogische Formel `PF` ein vollständig geschlossenes Aggregat aufgefunden, so folgt daraus, dass `PF` unerfüllbar ist.

In Anlehnung an das reguläre Resolutionslemma wird folgendes Lemma aufgestellt:

17
```
reslem_agg_agg: LEMMA
  (member(AGG_Pos, aggfilter(TC, PV, positive)) AND
    member(AGG_Neg, aggfilter(TC, PV, negative)))
   IMPLIES
   PVal(PI, asoc_pf(TC)) =
    PVal(PI, asoc_pf(append(TC, hybridize(AGG_Pos, AGG_Neg, PV))))
```

In diesem Lemma spielt die Funktion `asoc_pf` eine Rolle, die von einem Tubeinhalt die assoziierte aussagenlogische Formel zurückgibt. Sie ist sozusagen der Umkehrprozess von `build_tube`. So wie zwischen dem Tubeinhalt und einer Formel eine Assoziation besteht, korrespondiert ein Aggregat mit einer Klausel. Die Definition von `asoc_pf` und der Hilfsfunktion `asoc_pc`, die ein Aggregat in eine Klausel umwandelt, befinden sich auf S. 69. Hierbei werden natürlich nur die offenen Verbindungsstellen in entsprechende Literale umgewandelt.

Im Endeffekt sagt das Lemma 17 folgendes aus: wenn das Hybridisierungsprodukt zweier kompatibler Aggregate `AGG_Pos` und `AGG_Neg` eines Tubeinhalts `TC` zu diesem hinzugefügt wird, so ist die damit assoziierte Formel semantisch äquivalent zu der zu `TC` assoziierten Formel. Wir wissen nun durch folgendes Lemma,

18
```
is_compl_closed: LEMMA
  is_compl_closed(AGG) IMPLIES PVal(PI, asoc_pc(AGG)) = zero
```

dass das komplett geschlossene Aggregat, dass der leeren Klausel entspricht, unerfüllbar ist. Entsteht ein solches komplett geschlossenes Aggregat also irgendwann während der Hybridisierung, so folgt aus obigem Lemma 17, dass die mit dem Tubeinhalt assoziierte Formel unerfüllbar ist. Da der Tubeinhalt zu Beginn aus der Formel `PF` entsteht, muss diese also unerfüllbar gewesen sein.

Die obige Äquivalenz muss natürlich für alle während des Resolutionsverfahrens gebildeten Hybridisierungsprodukte gelten:

19
```
reslem: LEMMA
  PVal(PI, asoc_pf(TC)) = PVal(PI, asoc_pf(resolution(LPV, TC)))
```

Da die Resolution so organisiert ist, dass nach den in der Formel vorkommenden Variablen nach und nach kombinatorisch die Hybridisierungsprodukte gebildet werden, sind noch die Lemmata `reslem_tc_tc` und `reslem_agg_tcneg` sowie 17 notwendig, um das Lemma `reslem` zu beweisen.

Entsteht nun also während des Aufrufs von `resolution` das komplett geschlossene Aggregat, so folgt aus 19 (mit folgender Instantiierung: `TC` wird zu `build_tube(PF)` und `LPV` zu `PVarSingle(PVar(PF))`) und 18, dass `PF` unerfüllbar ist.

6.2.2 Vollständigkeit

Das Theorem zur Vollständigkeit des spezifizierten Verfahrens lautet in PVS:

20
```
completeness: THEOREM
  (FORALL PI: PVal(PI, PF) = zero) IMPLIES detect(resolution(PF)) = YES
```

Ist also die aussagenlogische Formel PF unerfüllbar, dann folgt daraus, dass durch die Anwendung des spezifizierten Verfahrens auf PF ein komplett geschlossenes Aggregat gebildet wird.

Die Vollständigkeit wird gezeigt durch Induktion über der Liste der in PF vorkommenden aussagenlogischen Variablen. Hierzu wird folgendes Hilfslemma konstruiert.

21
```
LPV1, LPV2: VAR list[PVAR]

compl_aux: LEMMA
  allmembers(PVarSingle(PVar(PF)), LPV1) AND
   LPV1 = PVarSingle(LPV2) AND (FORALL (PI): PVal(PI, PF) = zero)
   IMPLIES detect(resolution(LPV1, build_tube(PF))) = YES
```

Die Funktion allmembers mit der Typung [list[T], list[T] -> bool] gibt *true* zurück, wenn alle Elemente der ersten Liste auch Element der zweiten Liste sind, ansonsten ist der Rückgabewert *false*, siehe ihre Definition auf S. 76.

Somit haben wir in 21 eine Variablenliste LPV1, die alle Variablen der Formel PF enthält. Außerdem ist durch LPV1 = PVarSingle(LPV2) sichergestellt, dass jede Variable in LPV1 nur genau einmal vorkommt. Die Induktion selbst wird nun über LPV1 geführt, jedoch nicht mit dem standardmäßigen cons-Induktionsschema, sondern zur Erleichterung des Beweises mithilfe einer konstruierten append-Induktion:

22
```
list_append_induction: THEOREM
  FORALL (p: [list -> boolean]):
    (p(null) AND
      (FORALL (cons1_var: T, cons2_var: list):
         p(cons2_var) IMPLIES
          p(append(cons2_var, cons(cons1_var, null)))))
     IMPLIES (FORALL (l: list): p(l));
```

Der Induktionsanfang ist nach wie vor über die leere Liste null geregelt. Der Induktionsschritt behandelt dann jedoch die mittels append um ein Element vergrößerte Liste. Dieses Induktionsschema ist in der Theorie l_append_induction auf S. 77 definiert. Am Ende des Vollständigkeitsbeweises erklärt sich der Nutzen der Verwendung dieses Induktionsschemas.

Induktionsanfang. LPV1 sei also die leere Liste null. Aus der Voraussetzung allmembers(PVarSingle(PVar(PF)), null) folgt nach Lemma allmembers_null (S. 76), dass die Formel PF keine aussagenlogische Variable beinhaltet. Es gibt zwei Möglichkeiten: entweder PF ist die leere Formel, was im Widerspruch zur Unerfüllbarkeit von PF steht, oder aber PF besteht nur aus leeren Klauseln. In diesem Fall würde der Tubeinhalt nur

aus leeren Aggregaten bestehen, die wir als komplett geschlossene Aggregate verstehen wollen. Somit ergibt `detect` in diesem Fall `YES`.

Induktionsvoraussetzung. Die obige Aussage 21 gilt für eine beliebige Liste `l` aussagenlogischer Variablen an Stelle von `LPV1`.

Induktionsschritt. Wir betrachten nun den Fall, dass die Liste `LPV1` im Vergleich zu `l` die um die aussagenlogische Variable `v` verlängerte Liste `append(l, cons(v, null))` ist. Da wir wissen, dass alle Variablen, die in `PF` vorkommen, auch in der Liste `append(l, cons(v, null))` sind, kann die Variable `v` entweder eine Variable aus `PF` sein oder nicht.

Wir wollen nun mithilfe folgender Funktion

```
subst_pf(PF, PV, MK): RECURSIVE PFORM_fnc =
  CASES PF
    OF null: null,
       cons(h, t):
         IF member((MK, PV), h) THEN subst_pf(t, PV, MK)
         ELSIF member((opp_mk(MK), PV), h)
           THEN cons(remove((opp_mk(MK), PV), h), subst_pf(t, PV, MK))
         ELSE cons(h, subst_pf(t, PV, MK))
         ENDIF
    ENDCASES
   MEASURE length(PF)
```

zwei neue Formeln aus `PF` konstruieren, um die Induktionsvoraussetzung anwenden zu können:

- $\texttt{PF}_1$ = `subst_pf(PF, v, positive)`.
  $\texttt{PF}_1$ entsteht aus `PF`, indem bei Vorkommen des Literals `(positive, v)` innerhalb einer Klausel die gesamte Klausel gestrichen wird, und jedes Vorkommen von `(negative, v)` in einer Klausel gestrichen wird. Dies entspricht der Ersetzung von `v` durch den Wahrheitswert `one`.

- $\texttt{PF}_0$ = `subst_pf(PF, v, negative)`.
  $\texttt{PF}_0$ entsteht aus `PF`, indem jedes Vorkommen des Literals `(positive, v)` innerhalb einer Klausel gestrichen wird, und bei dem Vorkommen von `(negative, v)` innerhalb einer Klausel die gesamte Klausel gestrichen wird. Dies entspricht der Ersetzung von `v` durch den Wahrheitswert `zero`.

Durch die Prämisse aus 21 gilt `allmembers(PVarSingle(PVar(PF)), append(l, cons(v, null)))`. Daraus folgt nun `allmembers(PVarSingle(PVar(`$\texttt{PF}_1$`)), l)`, da die Variable `v` ersetzt wurde. Gleiches gilt für $\texttt{PF}_0$. Siehe dazu auch das entsprechende Lemma `allmembers_subst_app` auf S. 74.

Um nun die Induktionsvoraussetzung auf $\texttt{PF}_1$ und $\texttt{PF}_0$ endgültig anwenden zu können, muss erst noch gezeigt werden, dass die beiden Formeln unerfüllbar sind, wenn `PF` unerfüllbar ist. Dies leistet das Lemma `subst_pf`:

```
subst_pf: LEMMA
  (FORALL (PI): PVal(PI, PF) = zero) IMPLIES
   (FORALL (PI): PVal(PI, subst_pf(PF, PV, positive)) = zero) AND
    (FORALL (PI): PVal(PI, subst_pf(PF, PV, negative)) = zero)
```

Es wird bewiesen durch Widerspruch. Angenommen es gibt eine Interpretation PI, die $\texttt{PF}_1$ erfüllt, dann existiert eine Interpretation PI′:

$$\texttt{PI}'(a) = \begin{cases} \texttt{PI}(a) & \text{falls } a \text{ eine Variable aus } \texttt{PF}_1 \\ \texttt{one} & \text{falls } a = \texttt{v} \end{cases}$$

die PF erfüllt (Widerspruch). Analog wird dies für $\texttt{PF}_0$ gezeigt.

Die Induktionsvoraussetzung kann nun auf $\texttt{PF}_1$ und $\texttt{PF}_0$ angewendet werden (hierzu muss LPV2 noch geeignet instantiiert werden, siehe Lemma pvarsingle_apppv auf S. 63). Dadurch wissen wir nun, dass gilt: detect(resolution(l, build_tube($\texttt{PF}_1$))) = YES und auch detect(resolution(l, build_tube($\texttt{PF}_0$))) = YES.

Um nun das letztendliche Beweisziel:

$$\texttt{detect(resolution(append(l, cons(v, null)), build\_tube(PF)))} = \texttt{YES}$$

zu zeigen, gehen wir weiter wie folgt vor. Alle Klauseln der Formel $\texttt{PF}_1$ kommen in PF ebenfalls vor, nur mit dem Unterschied, dass in ihnen dort eventuell noch das Literal (negative, v) vorkommt (denn dieses wurde ja bei der Bildung von $\texttt{PF}_1$ aus den Klauseln gestrichen). Würde man nun die gleichen Hybridisierungsschritte mit den korrespondierenden Aggregaten aus build_tube(PF) vornehmen, so würde entweder das komplett geschlossene Aggregat entstehen, oder aber ein Aggregat, das nur noch aus den offenen Verbindungsstellen (v, negative, open) besteht (alle anderen wären geschlossen). Ähnlich verhält es sich mit den Klauseln aus $\texttt{PF}_0$: auch sie kommen in PF vor, nur mit dem Unterschied, dass in ihnen eventuell noch das Literal (positive, v) vorhanden ist. Würde man nun wieder die gleichen Hybridisierungsschritte (die im übrigen für $\texttt{PF}_1$ und $\texttt{PF}_0$ die gleichen sind, nämlich die Abarbeitung der Liste l) mit den korrespondierenden Aggregaten aus build_tube(PF) vornehmen, so würde entweder das komplett geschlossene Aggregat entstehen, oder aber ein Aggregat, dessen einzige offene Verbindungsstellen (v, positive, open) sind. Im jeweils ersten Fall, also wenn direkt schon das komplett geschlossene Aggregat gebildet wird, wird die detect-Operation des Beweisziels wie gewünscht YES zurückgeben. Im übrig bleibenden Fall würden nach der Abarbeitung der Variablen aus l die beiden Aggregate im Tube vorhanden sein, von denen eins als einzig offene Verbindungsstellen (v, negative, open) aufweist, während das andere als offene Verbindungsstellen (v, positive, open) besitzt. Diese beiden Aggregate würden jedoch im abschließenden Hybridisierungsschritt, den die Variable v bildet, zum komplett geschlossenen Aggregat zusammengefügt werden und somit ebenfalls das Beweisziel erfüllen. Durch das verwendete Induktionsschema 22 ist nun tatsächlich v diejenige Variable, mittels derer der letzte Hybridisierungsschritt durchgeführt wird.

Beispiel. Bevor wir uns der Umsetzung dieses Vorgehens in PVS zuwenden wollen, betrachten wir noch ein kleines Beispiel, das nur zur Verdeutlichung dienen soll. Angenommen, wir hätten die folgende konkrete Formel

$$\texttt{PF} = [[\neg p, \neg q], [\neg r, q], [r], [p]]$$

gegeben. Die Liste `LPV1`, die u.a. alle Variablen aus `PF` enthält, könnte dann z.B. `LPV1` $= [p, r, q]$ sein, was auch geschrieben werden kann als `LPV1 = append(`$[p, r]$`,`$[q]$`)`. Die beiden aus `PF` gebildeten Formeln sind dann: $\texttt{PF}_1 = [[\neg p], [r], [p]]$ sowie $\texttt{PF}_0 = [[\neg r], [r], [p]]$.

Es gilt nun `detect(resolution(`$[p, r]$`, build_tube(`$\texttt{PF}_1$`)) = YES` durch das Aggregat $[\boxed{\neg p}, \boxed{p}]$ sowie `detect(resolution(`$[p, r]$`, build_tube(`$\texttt{PF}_0$`)) = YES` durch das Aggregat $[\boxed{\neg r}, \boxed{r}]$.

Die zu $\texttt{PF}_1$ korrespondierenden Klauseln in `PF` sind $M_1 = [[\neg p, \neg q], [r], [p]]$ und die zu $\texttt{PF}_0$ korrespondierenden Klauseln in `PF` sind $M_0 = [[\neg r, q], [r], [p]]$.

Wird die Funktion `resolution(`$[p, r]$`, build_tube(`M_1`))` ausgeführt, so entsteht das Aggregat $[\boxed{\neg p}, q, \boxed{p}]$, bzw. das Aggregat $[\boxed{\neg r}, q, \boxed{r}]$ bei der Ausführung mit M_0.

Da $M_1 \subset$ `PF` und $M_0 \subset$ `PF` müssen die beiden Aggregate auch bei der Anwendung von `resolution(`$[p, r]$`, build_tube(PF))` entstehen. Wenn nun abschließend noch ein Hybridisierungsschritt mit $[q]$, also insgesamt `resolution(`$[p, r, q]$`, build_tube(PF))` ausgeführt wird, entsteht das komplett geschlossene Aggregat.

In PVS werden die zu $\texttt{PF}_1$ bzw. $\texttt{PF}_0$ korrespondierenden Klauseln in `PF` mithilfe der Funktion `subst_pf_corresp` spezifiziert.

```
subst_pf_corresp(PF, PV, MK): RECURSIVE PFORM_fnc =
  CASES PF
    OF null: null,
       cons(h, t):
         IF member((MK, PV), h) THEN subst_pf_corresp(t, PV, MK)
         ELSE cons(h, subst_pf_corresp(t, PV, MK))
         ENDIF
    ENDCASES
   MEASURE length(PF)
```

Sie ist der Funktion `subst_pf` sehr ähnlich, nur das keine Ersetzungen innerhalb der Klauseln vorgenommen werden. Somit ist klar, dass alle über `subst_pf_corresp` spezifizierten Klauseln auch in `PF` vorkommen. Da wir nun die gleichen Hybridisierungsschritte, die mit $\texttt{PF}_1$ bzw. $\texttt{PF}_0$ ausgeführt wurden und dort zum komplett geschlossenen Aggregat führten, mit den korrespondierenden Klauseln aus `PF` ausführen wollen, ersetzen wir `subst_pf` geeignet:

```
subst_pf_corresp: LEMMA
  del_plitn(build_tube(subst_pf_corresp(PF, PV, MK)), PV, opp_mk(MK)) =
   build_tube(subst_pf(PF, PV, MK))
```

Die Funktion `del_plitn` ersetzt das Herausstreichen von Literalen, wie es in `subst_pf` vorgenommen wurde, nun durch ein Entfernen der entsprechenden Verbindungsstellen. Zur Definition von `del_plitn` siehe S. 71.

Durch Umformungen und die Anwendung weiterer Lemmata erhalten wir:

```
detect(del_plitn(resolution(l, build_tube(PF)), v, positive)) = YES
detect(del_plitn(resolution(l, build_tube(PF)), v, negative)) = YES
```

Wir wissen also, dass die Ausführung der Funktion `resolution` mit der Liste `l` und der Formel `PF` zum richtigen Ergebnis führt, wenn vor der Ausführung von `detect` noch einmal `del_plitn` ausgeführt wird.

Dies bedeutet jedoch auch, dass durch `resolution(l, build_tube(PF))` entweder das komplett geschlossene Aggregat `is_compl_closed(AGG)` erzeugt wird. In diesem Fall wäre das Beweisziel bereits bewiesen. Oder aber es existieren zwei Aggregate `AGG1` und `AGG2` im Tube, von denen `AGG1` nur `(v, positive, open)` als offene Verbindungsstellen aufweist und `AGG2` nur `(v, negative, open)`. In PVS drücken wir dies durch die Funktion `openplits_equal` aus (siehe S. 72).

23

```
del_plitn_detect: LEMMA
  detect(del_plitn(TC, PV, MK)) = YES IMPLIES
   (EXISTS AGG:
      member(AGG, TC) AND
       (is_compl_closed(AGG) OR openplits_equal((PV, MK, open), AGG)))
```

Das Beweisziel

```
detect(resolution(append(l, cons(v, null)), build_tube(PF))) = YES
```

kann mithilfe von `resolution_app_lpv` (S. 75) zu

```
detect(resolution(cons(v, null), resolution(l, build_tube(PF)))) = YES
```

umgeschrieben werden.

Durch das Lemma `closeaggs_tc_tc` (S. 75) ist geklärt, dass die abschließende Hybridisierung mittels der Variable `v` bei Vorhandensein zweier Aggregate wie `AGG1` und `AGG2` ebenfalls zum komplett geschlossenen Aggregat führt. Das so bewiesene Lemma `compl_aux` muss nun nur noch geeignet intantiiert werden, um das Theorem `completeness` abschließend zu beweisen.

In `AGG1` und `AGG2` könnten die angesprochenen Verbindungsstellen natürlich auch mehrmals auftreten. Somit *müssen*, wie vorher schon angesprochen wurde, für die Vollständigkeit des Verfahrens in der Hybridisierungs-Operation *alle* Vorkommen von `(v, positive, open)` in `AGG1` und *alle* Vorkommen von `(v, negative, open)` in `AGG2` geschlossen werden. Hieraus ist ersichtlich, dass unbedingt eine Lösung für das Abschnitt 5.2.4 aufgezeigte Problem gefunden werden muss, um mit der beschriebenen Methode auch ein theoretisch vollständiges DNA-basiertes Verfahren zur Resolution zur Verfügung zu haben. Wird jedoch, wie hier geschehen, von diesem Problem in der Spezifikation abgesehen, besitzt diese die Eigenschaft der Vollständigkeit.

7 Schlussbetrachtung und Ausblick

In der vorliegenden Arbeit haben wir ein molekulares Programm für die DNA-basierte aussagenlogische Resolution auf Korrektheit und Vollständigkeit hin untersucht. Für die Spezifikation der molekularbiologischen Operationen und des Verfahrensablaufs sowie für die Beweisführung wurde das Verifikationssystem PVS eingesetzt. Die Vorgehensweise für die DNA-basierte Implementation entstammt einer Veröffentlichung, in der der entwickelte Ansatz durch ein erfolgreiches Laborexperiment untermauert werden konnte. Die Anzahl der notwendigen Laborschritte ist, abgesehen von der DNA-Kodierung der Klauseln, konstant und damit unabhängig von der Problemgröße.

Im Rahmen der Spezifizierung haben wir das Verfahren in zwei Punkten verändert. Zum Einen haben wir die Detektionsmethode derart modifiziert, so dass *beliebige* aussagenlogische Formeln in konjunktiver Normalform DNA-basiert auf ihre Unerfüllbarkeit (und mithin auf ihre Allgemeingültigkeit) untersucht werden können. Für eine Umsetzung der neuen Detektionsmethode existieren entsprechende molekularbiologische Operationen. Die zweite von uns vorgenommene Änderung betrifft das mehrfache Vorkommen eines Literalen innerhalb einer Klausel. In bestimmten Situationen kann dies die Bildung der Molekülstruktur verhindern, die innerhalb der Detektionsmethode als leere Klausel erkannt wird. Die Vollständigkeit des Verfahrens wäre so nicht gewährleistet. Eine molekularbiologische Lösung für dieses Problem ist nicht offensichtlich und steht noch aus. Kann diese nicht gefunden werden, muss eventuell über eine neue Kodierungsmethode oder eine gänzlich andere Vorgehensweise nachgedacht werden. In der Spezifikation wurde das Problem übergangen und die Operation im Vergleich zur biologischen entsprechend abstrahiert definiert. Mit den vorgenommenen Änderungen haben wir die Spezifikation als korrekt und vollständig nachgewiesen.

Mithilfe der Verifikation kann die Fehlerfreiheit eines molekularen Programms garantiert und damit auch Unzulänglichkeiten in seiner Konstruktion aufgedeckt werden. Außerdem sind gesicherte Aussagen über Eigenschaften und Wirkungen von Programmbestandteilen möglich. Gerade im Vorfeld einer Laborimplementation, die mit hohen Kosten und großem Aufwand verbunden ist, ist eine Verifikation deshalb sinnvoll. Eine Spezifikation von biologischen Vorgängen des DNA-Computing ist dabei nicht trivial: so ist es beispielsweise nötig, im Labor massiv parallel ablaufende Prozesse geeignet zu serialisieren. PVS eignet sich als maschinengestützter interaktiver Theorembeweiser aufgrund seiner ausdrucksstarken Spezifikationssprache für die Verifikation molekularer Programme. Für die Zukunft wäre es für die Entlastung des Anwenders wünschenswert, dass die Beweisführung z.B. durch die Definition von leistungsstarken Beweisstrategien noch stärker automatisiert ablaufen würde.

A PVS Quelltext zur DNA-basierten Resolution

Die Spezifikation ist auf die Dateien `cnf_properties.pvs`, `val_properties.pvs`, `hairpin_model.pvs`, `resolution.pvs`, `l_properties.pvs`, `l_append_induction.pvs` verteilt. Die „zentrale“ Theorie befindet sich in `resolution.pvs`, die alle restlichen Theorien importiert.

A.1 Theorie cnf_properties

```
% Theorie cnf_properties:
%   - Spezifikation der aussagenlog. Formel in KNF
%
cnf_properties[PVAR: TYPE]: THEORY
 BEGIN

  MARKERS: TYPE = {positive, negative}

  PLIT: TYPE = [MARKERS, PVAR]

  PCL: TYPE = list[PLIT]

  PFORM_fnc: TYPE = list[PCL]

  PV: VAR PVAR

  MK: VAR MARKERS

  PL: VAR PLIT

  PC: VAR PCL

  PF: VAR PFORM_fnc

  LPV: VAR list[PVAR]

  plit_positive?((MK), (PV)): bool = MK = positive

  PVar((MK), (PV)): PVAR = PV

  % gibt entgegengesetztes MK zurueck
  opp_mk(MK): MARKERS = IF (MK = positive) THEN negative ELSE positive ENDIF

  opp_mk_prop: LEMMA opp_mk(opp_mk(MK)) = MK
```

```
% Liste der Variablen aus PC
PVar(PC): RECURSIVE list[PVAR] =
  CASES PC OF null: null, cons(pl, pc): cons(PVar(pl), PVar(pc)) ENDCASES
   MEASURE length(PC)

pvar_pc_member: LEMMA
  member(PV, PVar(PC)) IFF
   member((MK, PV), PC) OR member((opp_mk(MK), PV), PC)

% Liste der Variablen aus PF
PVar(PF): RECURSIVE list[PVAR] =
  CASES PF OF null: null, cons(pc, pf): append(PVar(pc), PVar(pf))
    ENDCASES
   MEASURE length(PF)

% Jedes Element von LPV darf in der Ergebnisliste nur einmal vorkommen
PVarSingle(LPV): RECURSIVE list[PVAR] =
  CASES LPV
    OF null: null,
       cons(h, t):
         IF member(h, t) THEN PVarSingle(t)
         ELSE cons(h, PVarSingle(t))
         ENDIF
    ENDCASES
   MEASURE length(LPV)

pvarsingle_aux: LEMMA
  FORALL LPV: member(PV, PVarSingle(LPV)) IFF member(PV, LPV)

pvarsingle: LEMMA
  FORALL LPV:
    PVarSingle(LPV) = PVarSingle(cons(PV, LPV)) IFF
     member(PV, PVarSingle(LPV))

pvarsingle_null_aux: LEMMA null = PVarSingle(LPV) IFF LPV = null

pvar_null_pc: LEMMA null = PVar(PC) IFF PC = null

IMPORTING l_properties

pvar_null_pf: LEMMA
  null = PVar(PF) IFF
   (PF = null) OR (FORALL (PC): member(PC, PF) IMPLIES PC = null)

pvarsingle_null: LEMMA
  null = PVarSingle(PVar(PF)) IFF
   PF = null OR (FORALL (PC): member(PC, PF) IMPLIES PC = null)

pvarsingle_cons_remove: LEMMA
  PVarSingle(append(LPV, cons(PV, null))) =
   PVarSingle(append(remove(PV, LPV), cons(PV, null)))

pvarsingle_remove: LEMMA
  remove(PV, PVarSingle(LPV)) = PVarSingle(remove(PV, LPV))
```

```
  pvarsingle_elem_count: LEMMA
    member(PV, LPV) IMPLIES elem_count(PV, PVarSingle(LPV)) = 1

  pvarsingle_apppv: LEMMA
    FORALL (LPV1, LPV2: list[PVAR]):
      append(LPV1, cons(PV, null)) = PVarSingle(LPV2) IMPLIES
       LPV1 = remove(PV, PVarSingle(LPV2))
 END cnf_properties
```

A.2 Theorie val und val_properties

```
% Theorie val:
%   - definiert die Wahrheitswerte
%
val: THEORY
 BEGIN

  TRUTH_VALUES: TYPE = {zero, one}

  VV: VAR TRUTH_VALUES

  % gibt entgegengesetzten Wahrheitswert zurueck
  opp_value(VV): TRUTH_VALUES = IF VV = zero THEN one ELSE zero ENDIF

  prop_opp_value: LEMMA opp_value(opp_value(VV)) = VV
 END val



% Theorie val_properties:
%   - aussagenlogische Interpretation fuer Formeln
%   - Berechnung eines Wahrheitswertes fuer Formeln, Klauseln, Literale
%
val_properties[PVAR: TYPE]: THEORY
 BEGIN

  IMPORTING val, cnf_properties[PVAR]

  VALUATIONS: TYPE = [PVAR -> TRUTH_VALUES]

  VV, VV1, VV2: VAR TRUTH_VALUES

  PI: VAR VALUATIONS

  PV: VAR PVAR

  PL: VAR PLIT

  PC: VAR PCL

  PF: VAR PFORM_fnc
```

```
  PVal(PI, PL): TRUTH_VALUES =
      IF plit_positive?(PL) THEN PI(PVar(PL))
      ELSE opp_value(PI(PVar(PL)))
      ENDIF

  PVal(PI, PC): RECURSIVE TRUTH_VALUES =
    CASES PC
      OF null: zero,
         cons(pl, pc):
           IF PVal(PI, pl) = one THEN one ELSE PVal(PI, pc) ENDIF
      ENDCASES
     MEASURE length(PC)

  prop_PVal_pcl: LEMMA
    PVal(PI, PC) = one IFF
     (EXISTS PL: (member(PL, PC) AND PVal(PI, PL) = one))

  % Zuordnung eines Wahrheitswertes zu PF gemaess PI
  PVal(PI, PF): RECURSIVE TRUTH_VALUES =
    CASES PF
      OF null: one,
         cons(pc, pf):
           IF PVal(PI, pc) = zero THEN zero ELSE PVal(PI, pf) ENDIF
      ENDCASES
     MEASURE length(PF)

  prop_PVal_pform: LEMMA
    PVal(PI, PF) = one IFF
     (FORALL PC: (member(PC, PF) IMPLIES PVal(PI, PC) = one))
 END val_properties
```

A.3 Theorie hairpin_model

```
% Theorie hairpin_model:
%   - Spezifikation fast aller Modelloperationen (ausser resolution)
%   - Definitionen der Aggregate, Tubeinhalte etc.
%
hairpin_model[PVAR: TYPE]: THEORY
 BEGIN

  IMPORTING cnf_properties[PVAR]

  STICKYMARKER: TYPE = {open, closed}

  PLITN: TYPE = [PVAR, MARKERS, STICKYMARKER]

  IMPORTING l_properties[PLITN]

  AGGREGATE: TYPE = list[PLITN]

  TUBECONTENT: TYPE = list[AGGREGATE]

  DECISION: TYPE = {YES, NO}
```

```
PV: VAR PVAR

MK: VAR MARKERS

SMK: VAR STICKYMARKER

PLN, PLN2: VAR PLITN

AGG, AGG1, AGG2, AGG_Pos, AGG_Neg, AGG_aux: VAR AGGREGATE

TC, TC_Pos, TC_Neg: VAR TUBECONTENT

% Schiebe die Elemente aus AGG solange, bis PLN vorn steht
agg_shift(AGG, PLN, AGG_aux): RECURSIVE AGGREGATE =
  CASES AGG
    OF null: AGG_aux,
       cons(h, t):
         IF PLN = h THEN append(AGG, AGG_aux)
         ELSE agg_shift(t, PLN, append(AGG_aux, cons(h, null)))
         ENDIF
    ENDCASES
   MEASURE length(AGG)

agg_shift_member_aux: LEMMA
  member(PLN, agg_shift(AGG, PLN2, AGG_aux)) IFF
   member(PLN, AGG) OR member(PLN, AGG_aux)

agg_shift_member: LEMMA
  member(PLN, AGG) IFF member(PLN, agg_shift(AGG, PLN2, null))

close_lit(PLN): PLITN = PLN WITH [`3 := closed]

% Schliesse alle offenen Verbindungsstellen, die PV und MK enthalten
close_lit(AGG, PV, MK): RECURSIVE AGGREGATE =
  CASES AGG
    OF null: null,
       cons(h, t):
         IF h = (PV, MK, open)
           THEN cons(close_lit(h), close_lit(t, PV, MK))
         ELSE cons(h, close_lit(t, PV, MK))
         ENDIF
    ENDCASES
   MEASURE length(AGG)

close_lit_app: LEMMA
  close_lit(append(AGG1, AGG2), PV, MK) =
   append(close_lit(AGG1, PV, MK), close_lit(AGG2, PV, MK))

close_lit_member: LEMMA NOT member((PV, MK, open), close_lit(AGG, PV, MK))

close_lit_notmem: LEMMA
  NOT member(PLN, AGG) AND PLN`3 = open IMPLIES
   NOT member(PLN, close_lit(AGG, PV, MK))
```

```
% Hybridisierung zweier passender Aggregate anhand PV
hybridize(AGG_Pos, AGG_Neg, PV): TUBECONTENT =
    IF member((PV, positive, open), AGG_Pos) AND
        member((PV, negative, open), AGG_Neg)
      THEN cons(append(agg_shift(close_lit(AGG_Pos, PV, positive),
                                 (PV, positive, closed), null),
                       agg_shift(close_lit(AGG_Neg, PV, negative),
                                 (PV, negative, closed), null)),
                null)
    ELSE cons(AGG_Neg, cons(AGG_Pos, null))
    ENDIF

% Hybridisiere alle Aggregate aus TC_Neg mit AGG_Pos anhand PV
hybridize(AGG_Pos, TC_Neg, PV): RECURSIVE TUBECONTENT =
  CASES TC_Neg
    OF null: null,
       cons(h, t):
         append(hybridize(AGG_Pos, h, PV), hybridize(AGG_Pos, t, PV))
    ENDCASES
   MEASURE length(TC_Neg)

hybridize_agg_tc_null: LEMMA
  hybridize(AGG_Pos, TC_Neg, PV) = null IFF TC_Neg = null

% Hybridisiere alle Agg's aus TC_Pos mit allen Agg's aus TC_Neg anhand PV
hybridize(TC_Pos, TC_Neg, PV): RECURSIVE TUBECONTENT =
  CASES TC_Pos
    OF null: null,
       cons(h, t):
         append(hybridize(h, TC_Neg, PV), hybridize(t, TC_Neg, PV))
    ENDCASES
   MEASURE length(TC_Pos)

hybridize_tc_tc_null: LEMMA
  hybridize(TC_Pos, TC_Neg, PV) = null IFF TC_Pos = null OR TC_Neg = null

is_closed((PV), (MK), (SMK)): bool = SMK = closed

% ist AGG komplett geschlossen?
is_compl_closed(AGG): RECURSIVE bool =
  CASES AGG
    OF null: TRUE,
       cons(h, t):
         IF is_closed(h) = FALSE THEN FALSE ELSE is_compl_closed(t) ENDIF
    ENDCASES
   MEASURE length(AGG)

is_compl_closed_prop: LEMMA
  is_compl_closed(AGG) IMPLIES
   (FORALL PLN: member(PLN, AGG) IMPLIES is_closed(PLN))

is_compl_closed_app: LEMMA
  is_compl_closed(append(AGG1, AGG2)) IFF
```

```
    is_compl_closed(AGG1) AND is_compl_closed(AGG2)

  is_compl_closed_agg_shift: LEMMA
    is_compl_closed(agg_shift(AGG, PLN, AGG_aux)) IFF
     is_compl_closed(AGG) AND is_compl_closed(AGG_aux)

  % enthaelt TC ein komplett geschlossenes Aggregat?
  detect(TC): DECISION =
      IF (EXISTS AGG: member(AGG, TC) AND is_compl_closed(AGG)) THEN YES
      ELSE NO
      ENDIF
 END hairpin_model
```

A.4 Theorie resolution

```
% Theorie resolution:
%   - enthaelt alle wichtigen Theoreme, Lemmata und noetigen Funktionen
%   - Startpunkt fuer die Hybridisierung und importiert alle anderen Theorien.
%
resolution[PVAR: TYPE]: THEORY
 BEGIN

  IMPORTING val, val_properties[PVAR], hairpin_model[PVAR],
            cnf_properties[PVAR], l_properties

  PV, PV1, PV2: VAR PVAR

  MK, MK1, MK2: VAR MARKERS

  PL: VAR PLIT

  PC, PC1, PC2: VAR PCL

  PF, PF1, PF2: VAR PFORM_fnc

  TC, TC1, TC2, TC_Pos, TC_Pos1, TC_Pos2,
    TC_Neg, TC_Neg1, TC_Neg2: VAR TUBECONTENT

  LPV, LPV1, LPV2: VAR list[PVAR]

  PI: VAR VALUATIONS

  PLN: VAR PLITN

  AGG, AGG1, AGG2, AGG_Pos, AGG_Neg, AGG_aux: VAR AGGREGATE

  extend_literal((MK), (PV)): PLITN = (PV, MK, open)

  % Umwandundlung Klausel in Aggregat
  extend_clause(PC): RECURSIVE AGGREGATE =
    CASES PC
      OF null: null,
         cons(pl, pc): cons(extend_literal(pl), extend_clause(pc))
```

```
    ENDCASES
   MEASURE length(PC)

extend_clause_rem: LEMMA
  extend_clause(remove((MK, PV), PC)) =
   remove((PV, MK, open), extend_clause(PC))

extend_clause_mem: LEMMA
  member((MK, PV), PC) IFF member((PV, MK, open), extend_clause(PC))

% Umwandlung der Formel PF in den Tubeinhalt
build_tube(PF): RECURSIVE TUBECONTENT =
  CASES PF
    OF null: null, cons(pc, pf): cons(extend_clause(pc), build_tube(pf))
    ENDCASES
   MEASURE length(PF)

build_tube_null: LEMMA build_tube(PF) = null IMPLIES PF = null

build_tube_member: LEMMA
  member(PC, PF) IMPLIES member(extend_clause(PC), build_tube(PF))

% Filtere alle Aggregate aus TC heraus, die PV und MK enthalten
aggfilter(TC, PV, MK): RECURSIVE TUBECONTENT =
  CASES TC
    OF null: null,
       cons(h, t):
         IF member((PV, MK, open), h) THEN cons(h, aggfilter(t, PV, MK))
         ELSE aggfilter(t, PV, MK)
         ENDIF
    ENDCASES
   MEASURE length(TC)

aggfilter_member: LEMMA
  member(AGG, aggfilter(TC, PV, MK)) IMPLIES member((PV, MK, open), AGG)

aggfilter_member_tc: LEMMA
  member(AGG, aggfilter(TC, PV, MK)) IMPLIES member(AGG, TC)

aggfilter_mem_agg_tc: LEMMA
  member(AGG, TC) AND member((PV, MK, open), AGG) IMPLIES
   member(AGG, aggfilter(TC, PV, MK))

aggfilter_allmem: LEMMA
  allmembers(TC1, TC2) IMPLIES
   allmembers(aggfilter(TC1, PV, MK), aggfilter(TC2, PV, MK))

% Fuehre den Hybridisierungsvorgang anhand der Liste LPV aus
resolution(LPV, TC): RECURSIVE TUBECONTENT =
  CASES LPV
    OF null: TC,
       cons(h, t):
         resolution(t,
                    append(TC,
```

```
                          hybridize(aggfilter(TC, h, positive),
                                    aggfilter(TC, h, negative),
                                    h)))
    ENDCASES
   MEASURE length(LPV)

resolution_allmembers: LEMMA allmembers(TC, resolution(LPV, TC))

% Mit AGG assoziierte Klausel
asoc_pc(AGG): RECURSIVE PCL =
  CASES AGG
    OF null: null,
       cons(h, t):
         IF h`3 = open THEN cons((h`2, h`1), asoc_pc(t))
         ELSE asoc_pc(t)
         ENDIF
    ENDCASES
   MEASURE length(AGG)

asoc_pc: LEMMA PVal(PI, PC) = PVal(PI, asoc_pc(extend_clause(PC)))

asoc_pc_app: LEMMA
  asoc_pc(append(AGG1, AGG2)) = append(asoc_pc(AGG1), asoc_pc(AGG2))

is_compl_closed: LEMMA
  is_compl_closed(AGG) IMPLIES PVal(PI, asoc_pc(AGG)) = zero

asoc_pc_mem: LEMMA
  member((PV, MK, open), AGG) IFF member((MK, PV), asoc_pc(AGG))

% Mit TC assoziierte Formel
asoc_pf(TC): RECURSIVE PFORM_fnc =
  CASES TC OF null: null, cons(h, t): cons(asoc_pc(h), asoc_pf(t))
    ENDCASES
   MEASURE length(TC)

asoc_pf_null: LEMMA asoc_pf(TC) = null IFF TC = null

asoc_pf: LEMMA PVal(PI, PF) = PVal(PI, asoc_pf(build_tube(PF)))

asoc_pf_pi: LEMMA
  (FORALL PI: PVal(PI, PF) = zero) IFF
   (FORALL PI: PVal(PI, asoc_pf(build_tube(PF))) = zero)

asoc_pf_app: LEMMA
  asoc_pf(append(TC1, TC2)) = append(asoc_pf(TC1), asoc_pf(TC2))

% Substitution des  (PV, MK) durch 'one', (PV, opp_mk(MK)) durch 'zero'
subst_pf(PF, PV, MK): RECURSIVE PFORM_fnc =
  CASES PF
    OF null: null,
       cons(h, t):
         IF member((MK, PV), h) THEN subst_pf(t, PV, MK)
         ELSIF member((opp_mk(MK), PV), h)
```

```
          THEN cons(remove((opp_mk(MK), PV), h), subst_pf(t, PV, MK))
        ELSE cons(h, subst_pf(t, PV, MK))
        ENDIF
    ENDCASES
   MEASURE length(PF)

subst_pc_posmem: LEMMA
  member((MK, PV), PC) IMPLIES
   PVal(LAMBDA (PV1: PVAR):
          IF PV1 = PV THEN IF MK = positive THEN one ELSE zero ENDIF
          ELSE PI(PV1)
          ENDIF,
        PC)
    = one

subst_pc_lem: LEMMA
  PVal(PI, remove((MK, PV), PC)) = one IMPLIES
   PVal(LAMBDA (PV1: PVAR):
          IF PV1 = PV
            THEN IF opp_mk(MK) = positive THEN one ELSE zero ENDIF
          ELSE PI(PV1)
          ENDIF,
        PC)
    = one

subst_pc_notmem: LEMMA
  NOT member((MK, PV), PC) AND
   NOT member((opp_mk(MK), PV), PC) AND PVal(PI, PC) = one
   IMPLIES
   PVal(LAMBDA (PV1: PVAR):
          IF PV1 = PV
            THEN IF opp_mk(MK) = positive THEN one ELSE zero ENDIF
          ELSE PI(PV1)
          ENDIF,
        PC)
    = one
    AND
    PVal(LAMBDA (PV1: PVAR):
           IF PV1 = PV THEN IF MK = positive THEN one ELSE zero ENDIF
           ELSE PI(PV1)
           ENDIF,
         PC)
     = one

subst_pf_lem: LEMMA
  PVal(PI, subst_pf(PF, PV, MK)) = one IMPLIES
   PVal(LAMBDA (PV1: PVAR):
          IF PV1 = PV THEN IF MK = positive THEN one ELSE zero ENDIF
          ELSE PI(PV1)
          ENDIF,
        PF)
    = one
```

```
subst_pf: LEMMA
  (FORALL (PI): PVal(PI, PF) = zero) IMPLIES
   (FORALL (PI): PVal(PI, subst_pf(PF, PV, positive)) = zero) AND
    (FORALL (PI): PVal(PI, subst_pf(PF, PV, negative)) = zero)

rem_asoc_pc: LEMMA
  remove((MK, PV), asoc_pc(AGG)) = asoc_pc(remove((PV, MK, open), AGG))

% Loesche aus dem Aggregat die Verbindungsstelle (PV, MK, open)
del_plitn(AGG, PV, MK): AGGREGATE = remove((PV, MK, open), AGG)

del_plitn_agg_app: LEMMA
  del_plitn(append(AGG1, AGG2), PV, MK) =
   append(del_plitn(AGG1, PV, MK), del_plitn(AGG2, PV, MK))

del_plitn_agg_closelit: LEMMA
  NOT PV1 = PV2 IMPLIES
   del_plitn(close_lit(AGG, PV1, MK1), PV2, MK2) =
    close_lit(del_plitn(AGG, PV2, MK2), PV1, MK1)

del_plitn_agg_shift: LEMMA
  NOT PV1 = PV2 IMPLIES
   del_plitn(agg_shift(AGG, (PV1, MK1, closed), AGG_aux), PV2, MK2) =
    agg_shift(del_plitn(AGG, PV2, MK2), (PV1, MK1, closed),
              del_plitn(AGG_aux, PV2, MK2))

% Fuehre del_plitn auf allen Aggregaten von TC aus
del_plitn(TC, PV, MK): RECURSIVE TUBECONTENT =
  CASES TC
    OF null: null,
       cons(h, t): cons(del_plitn(h, PV, MK), del_plitn(t, PV, MK))
    ENDCASES
   MEASURE length(TC)

del_plitn_tc_app: LEMMA
  del_plitn(append(TC1, TC2), PV, MK) =
   append(del_plitn(TC1, PV, MK), del_plitn(TC2, PV, MK))

del_plitn_tc_aggfilter: LEMMA
  NOT PV1 = PV2 IMPLIES
   del_plitn(aggfilter(TC, PV1, MK1), PV2, MK2) =
    aggfilter(del_plitn(TC, PV2, MK2), PV1, MK1)

% wie subst_pf, jedoch keine Ersetzung in den Klauseln
subst_pf_corresp(PF, PV, MK): RECURSIVE PFORM_fnc =
  CASES PF
    OF null: null,
       cons(h, t):
         IF member((MK, PV), h) THEN subst_pf_corresp(t, PV, MK)
         ELSE cons(h, subst_pf_corresp(t, PV, MK))
         ENDIF
    ENDCASES
   MEASURE length(PF)
```

```
subst_pf_corresp: LEMMA
  del_plitn(build_tube(subst_pf_corresp(PF, PV, MK)), PV, opp_mk(MK)) =
   build_tube(subst_pf(PF, PV, MK))

subst_pf_corresp_allmem: LEMMA
  allmembers(subst_pf_corresp(PF, PV, MK), PF)

% Sind die offenen Verb.stellen in AGG nur PLN's?
openplits_equal(PLN, AGG): RECURSIVE boolean =
  CASES AGG
    OF null: TRUE,
       cons(h, t):
         IF h`3 = closed OR h = PLN THEN openplits_equal(PLN, t)
         ELSE FALSE
         ENDIF
    ENDCASES
   MEASURE length(AGG)

openplits_equal_prop: LEMMA
  openplits_equal(PLN, AGG) IMPLIES
   (member(PLN, AGG) OR is_compl_closed(AGG))

openplits_equal_close_lit: LEMMA
  openplits_equal(PLN, AGG) IMPLIES
   is_compl_closed(close_lit(AGG, PLN`1, PLN`2))

openplits_equal_notmem: LEMMA
  openplits_equal(PLN, AGG) AND NOT member(PLN, AGG) IMPLIES
   is_compl_closed(AGG)

openplits_equal_closed: LEMMA
  is_compl_closed(AGG) IMPLIES openplits_equal(PLN, AGG)

pval_pc_or: LEMMA
  PVal(PI, append(PC1, PC2)) =
   IF (PVal(PI, PC1) = one OR PVal(PI, PC2) = one) THEN one
   ELSE zero
   ENDIF

pval_pf_and: LEMMA
  PVal(PI, append(PF1, PF2)) =
   IF (PVal(PI, PF1) = one AND PVal(PI, PF2) = one) THEN one
   ELSE zero
   ENDIF

pval_pf_member: LEMMA
  member(AGG, TC) AND PVal(PI, asoc_pf(TC)) = one IMPLIES
   PVal(PI, asoc_pc(AGG)) = one

pval_aggshift_aux: LEMMA
  PVal(PI, asoc_pc(agg_shift(AGG, PLN, AGG_aux))) =
   PVal(PI, append(asoc_pc(AGG), asoc_pc(AGG_aux)))

pval_aggshift: LEMMA
```

```
  PVal(PI, asoc_pc(agg_shift(AGG, PLN, null))) = PVal(PI, asoc_pc(AGG))

pval_close_lit: LEMMA
  PVal(PI, asoc_pc(AGG)) = one AND PVal(PI, (MK, PV)) = zero IMPLIES
   PVal(PI, asoc_pc(close_lit(AGG, PV, MK))) = one

pval_tc_agg_zero: LEMMA
  PVal(PI, asoc_pc(AGG)) = zero AND member(AGG, TC) IMPLIES
   PVal(PI, asoc_pf(TC)) = zero

reslem_agg_agg: LEMMA
  (member(AGG_Pos, aggfilter(TC, PV, positive)) AND
    member(AGG_Neg, aggfilter(TC, PV, negative)))
   IMPLIES
   PVal(PI, asoc_pf(TC)) =
    PVal(PI, asoc_pf(append(TC, hybridize(AGG_Pos, AGG_Neg, PV))))

reslem_agg_tcneg: LEMMA
  member(AGG_Pos, aggfilter(TC, PV, positive)) AND
   allmembers(TC_Neg, aggfilter(TC, PV, negative))
   IMPLIES
   PVal(PI, asoc_pf(TC)) =
    PVal(PI, asoc_pf(append(TC, hybridize(AGG_Pos, TC_Neg, PV))))

reslem_tc_tc: LEMMA
  allmembers(TC_Pos, aggfilter(TC, PV, positive)) AND
   allmembers(TC_Neg, aggfilter(TC, PV, negative))
   IMPLIES
   PVal(PI, asoc_pf(TC)) =
    PVal(PI, asoc_pf(append(TC, hybridize(TC_Pos, TC_Neg, PV))))

% Resolutionslemma
reslem: LEMMA
  PVal(PI, asoc_pf(TC)) = PVal(PI, asoc_pf(resolution(LPV, TC)))

% Startpunkt des Hybridisierungsvorgangs
resolution(PF): TUBECONTENT =
    resolution(PVarSingle(PVar(PF)), build_tube(PF))

soundness: THEOREM
  detect(resolution(PF)) = YES IMPLIES (FORALL PI: PVal(PI, PF) = zero)

allmembers_pvarsingle: LEMMA
  allmembers(PVarSingle(LPV1), LPV2) IFF allmembers(LPV1, LPV2)

allm_pc_rem: LEMMA
  NOT member((MK, PV), PC) IMPLIES
   NOT member(PV, PVar(remove((opp_mk(MK), PV), PC)))

allmem_pvar_rem: LEMMA
  allmembers(PVar(remove((opp_mk(MK), PV), PC)), PVar(PC))

allmem_pc_notmem: LEMMA
  NOT member((MK, PV), PC) AND NOT member((opp_mk(MK), PV), PC) IFF
```

```
    NOT member(PV, PVar(PC))

  allmembers_subst_app: LEMMA
    allmembers(PVarSingle(PVar(PF)), append(LPV, cons(PV, null))) IMPLIES
     allmembers(PVarSingle(PVar(subst_pf(PF, PV, MK))), LPV)

  del_agg_agg: LEMMA
    NOT PV1 = PV2 IMPLIES
     hybridize(del_plitn(AGG_Pos, PV1, MK), del_plitn(AGG_Neg, PV1, MK),
               PV2)
      = del_plitn(hybridize(AGG_Pos, AGG_Neg, PV2), PV1, MK)

  del_agg_tcneg: LEMMA
    NOT PV1 = PV2 IMPLIES
     hybridize(del_plitn(AGG_Pos, PV1, MK), del_plitn(TC_Neg, PV1, MK),
               PV2)
      = del_plitn(hybridize(AGG_Pos, TC_Neg, PV2), PV1, MK)

  del_tc_tc: LEMMA
    NOT PV1 = PV2 IMPLIES
     hybridize(del_plitn(TC_Pos, PV1, MK), del_plitn(TC_Neg, PV1, MK), PV2)
      = del_plitn(hybridize(TC_Pos, TC_Neg, PV2), PV1, MK)

  del_res: LEMMA
    NOT member(PV, LPV) IMPLIES
     resolution(LPV, del_plitn(TC, PV, MK)) =
      del_plitn(resolution(LPV, TC), PV, MK)

  mem_agg_tc_neg: LEMMA
    member(AGG_Neg, TC_Neg) IMPLIES
     allmembers(hybridize(AGG_Pos, AGG_Neg, PV),
                hybridize(AGG_Pos, TC_Neg, PV))

  allmem_agg_tcneg: LEMMA
    allmembers(TC_Neg1, TC_Neg2) IMPLIES
     allmembers(hybridize(AGG_Pos, TC_Neg1, PV),
                hybridize(AGG_Pos, TC_Neg2, PV))

  allmem_tc_tc_aux: LEMMA
    member(AGG_Pos, TC_Pos) AND allmembers(TC_Neg1, TC_Neg2) IMPLIES
     allmembers(hybridize(AGG_Pos, TC_Neg1, PV),
                hybridize(TC_Pos, TC_Neg2, PV))

  allmem_tc_tc: LEMMA
    allmembers(TC_Pos1, TC_Pos2) AND allmembers(TC_Neg1, TC_Neg2) IMPLIES
     allmembers(hybridize(TC_Pos1, TC_Neg1, PV),
                hybridize(TC_Pos2, TC_Neg2, PV))

  allmem_res_aux: LEMMA
    allmembers(TC1, TC2) IMPLIES
     allmembers(resolution(LPV, TC1), resolution(LPV, TC2))

  allmem_res: LEMMA
    allmembers(PF1, PF2) IMPLIES
```

```
    allmembers(resolution(LPV, build_tube(PF1)),
               resolution(LPV, build_tube(PF2)))

  del_plitn_closed: LEMMA
    is_compl_closed(del_plitn(AGG, PV, MK)) IMPLIES
     (is_compl_closed(AGG) OR openplits_equal((PV, MK, open), AGG))

  del_plitn_detect: LEMMA
    detect(del_plitn(TC, PV, MK)) = YES IMPLIES
     (EXISTS AGG:
        member(AGG, TC) AND
         (is_compl_closed(AGG) OR openplits_equal((PV, MK, open), AGG)))

  closeaggs_agg_agg: LEMMA
    openplits_equal((PV, positive, open), AGG_Pos) AND
     openplits_equal((PV, negative, open), AGG_Neg) AND
      NOT AGG_Pos = null AND NOT AGG_Neg = null
     IMPLIES detect(hybridize(AGG_Pos, AGG_Neg, PV)) = YES

  closeaggs_agg_tc: LEMMA
    openplits_equal((PV, positive, open), AGG_Pos) AND
     (EXISTS (AGG_Neg: AGGREGATE):
        member(AGG_Neg, TC_Neg) AND
         openplits_equal((PV, negative, open), AGG_Neg))
      AND NOT AGG_Pos = null AND NOT TC_Neg = null
     IMPLIES detect(hybridize(AGG_Pos, TC_Neg, PV)) = YES

  closeaggs_tc_tc: LEMMA
    (EXISTS (AGG_Pos: AGGREGATE):
       member(AGG_Pos, TC_Pos) AND
        openplits_equal((PV, positive, open), AGG_Pos))
     AND
     (EXISTS (AGG_Neg: AGGREGATE):
        member(AGG_Neg, TC_Neg) AND
         openplits_equal((PV, negative, open), AGG_Neg))
      AND NOT TC_Pos = null AND NOT TC_Neg = null
     IMPLIES detect(hybridize(TC_Pos, TC_Neg, PV)) = YES

  resolution_app_lpv: LEMMA
    resolution(append(LPV1, LPV2), TC) =
     resolution(LPV2, resolution(LPV1, TC))

  IMPORTING l_append_induction[PVAR]

  compl_aux: LEMMA
    allmembers(PVarSingle(PVar(PF)), LPV1) AND
     LPV1 = PVarSingle(LPV2) AND (FORALL (PI): PVal(PI, PF) = zero)
     IMPLIES detect(resolution(LPV1, build_tube(PF))) = YES

  completeness: THEOREM
    (FORALL PI: PVal(PI, PF) = zero) IMPLIES detect(resolution(PF)) = YES
END resolution
```

A.5 Theorie l_properties

```
l_properties[ELEMENTS: TYPE]: THEORY
 BEGIN

  l, l1, l2: VAR list[ELEMENTS]

  x, y: VAR ELEMENTS

  member_append: LEMMA
    member(x, append(l1, l2)) IFF member(x, l1) OR member(x, l2)

  % sind alle Elemente von l1 auch Element von l2?
  allmembers(l1, l2): RECURSIVE boolean =
    CASES l1 OF null: TRUE, cons(h, t): member(h, l2) AND allmembers(t, l2)
      ENDCASES
     MEASURE length(l1)

  prop_allmembers: LEMMA
    allmembers(l1, l2) IFF (FORALL x: member(x, l1) IMPLIES (member(x, l2)))

  allmembers_refl: LEMMA FORALL (l): allmembers(l, l)

  allmembers_null: LEMMA
    allmembers(l, null) IMPLIES l = null

  allmembers_app: LEMMA
    allmembers(append(l1, l2), l) IFF
     allmembers(l1, l) AND allmembers(l2, l)

  allmembers_cons: LEMMA
    allmembers(l1, cons(x, l2)) AND NOT member(x, l1) IMPLIES
     allmembers(l1, l2)

  allmembers_cons_back: LEMMA
    allmembers(l1, l2) IMPLIES allmembers(l1, cons(x, l2))

  allmembers_allmembers: LEMMA
    allmembers(l1, l2) AND allmembers(l, l1) IMPLIES allmembers(l, l2)

  remove(x, l): RECURSIVE list[ELEMENTS] =
    CASES l
      OF null: null,
         cons(h, t):
           IF h = x THEN remove(x, t) ELSE cons(h, remove(x, t)) ENDIF
     ENDCASES
    MEASURE length(l)

  remove_append: LEMMA
    remove(x, (append(l1, l2))) = append(remove(x, l1), remove(x, l2))

  prop_rem: LEMMA NOT member(x, l) IMPLIES l = remove(x, l)

  rem_notmem: LEMMA NOT member(x, remove(x, l))
```

```
  rem_member: LEMMA member(y, remove(x, l)) IFF member(y, l) AND NOT x = y

  allmembers_remove: LEMMA
    allmembers(l1, l2) AND NOT member(x, l1) IMPLIES
     allmembers(l1, remove(x, l2))

  allmembers_remove_back: LEMMA
    allmembers(l1, remove(x, l2)) IMPLIES allmembers(l1, l2)

  prop_app: LEMMA null = append(l1, l2) IFF l1 = null AND l2 = null

  length_null: LEMMA length(l) = 0 IFF l = null

  cons_equal: LEMMA cons(x, l1) = cons(y, l2) IFF x = y AND l1 = l2

  elem_count(x, l): RECURSIVE nat =
    CASES l
      OF null: 0,
         cons(h, t):
           IF h = x THEN 1 + elem_count(x, t)
           ELSE 0 + elem_count(x, t)
           ENDIF
     ENDCASES
    MEASURE length(l)

  elem_count_notmem: LEMMA NOT member(x, l) IFF elem_count(x, l) = 0

  elem_count_append: LEMMA
    elem_count(x, append(l1, l2)) = elem_count(x, l1) + elem_count(x, l2)
 END l_properties
```

A.6 Theorie l_append_induction

```
l_append_induction[T: TYPE]: THEORY
 BEGIN

  IMPORTING list_adt[T]

  l: VAR list[T]

  list_length_null: LEMMA length(l) = 0 IFF l = null

  list_length_notnull: LEMMA length(l) > 0 IFF NOT l = null

  remove_last(l): RECURSIVE list[T] =
    CASES l
      OF null: null,
         cons(h, t):
           IF t = null THEN null ELSE cons(h, remove_last(t)) ENDIF
     ENDCASES
    MEASURE length(l)
```

```
  remove_last_length: LEMMA
    NOT l = null IMPLIES length(l) = length(remove_last(l)) + 1

  remove_last_prop: LEMMA
    NOT l = null IMPLIES
     append(remove_last(l), cons(nth(l, length(l) - 1), null)) = l

  list_append_induction: THEOREM
    FORALL (p: [list -> boolean]):
      (p(null) AND
        (FORALL (cons1_var: T, cons2_var: list):
           p(cons2_var) IMPLIES
            p(append(cons2_var, cons(cons1_var, null)))))
       IMPLIES (FORALL (l: list): p(l));
END l_append_induction
```

A.7 PVS-Protokoll

```
Proof summary for theory val_properties
   PVal_TCC1.............................proved - complete   [shostak](0.75 s)
   prop_PVal_pcl.........................proved - complete   [shostak](1.18 s)
   PVal_TCC2.............................proved - complete   [shostak](0.41 s)
   prop_PVal_pform.......................proved - complete   [shostak](0.70 s)
   Theory totals: 4 formulas, 4 attempted, 4 succeeded (3.04 s)

Proof summary for theory val
   prop_opp_value........................proved - complete   [shostak](0.08 s)
   Theory totals: 1 formulas, 1 attempted, 1 succeeded (0.08 s)

Proof summary for theory hairpin_model
   agg_shift_TCC1........................proved - complete   [shostak](0.54 s)
   agg_shift_member_aux..................proved - complete   [shostak](1.13 s)
   agg_shift_member......................proved - complete   [shostak](0.47 s)
   close_lit_TCC1........................proved - complete   [shostak](0.36 s)
   close_lit_TCC2........................proved - complete   [shostak](0.36 s)
   close_lit_app.........................proved - complete   [shostak](0.29 s)
   close_lit_member......................proved - complete   [shostak](0.26 s)
   close_lit_notmem......................proved - complete   [shostak](0.32 s)
   hybridize_TCC1........................proved - complete   [shostak](0.52 s)
   hybridize_agg_tc_null.................proved - complete   [shostak](0.33 s)
   hybridize_tc_tc_null..................proved - complete   [shostak](0.37 s)
   is_compl_closed_TCC1..................proved - complete   [shostak](0.38 s)
   is_compl_closed_prop..................proved - complete   [shostak](0.19 s)
   is_compl_closed_app...................proved - complete   [shostak](0.26 s)
   is_compl_closed_agg_shift.............proved - complete   [shostak](0.43 s)
   Theory totals: 15 formulas, 15 attempted, 15 succeeded (6.21 s)

Proof summary for theory cnf_properties
   opp_mk_prop...........................proved - complete   [shostak](0.08 s)
   PVar_TCC1.............................proved - complete   [shostak](0.36 s)
   pvar_pc_member........................proved - complete   [shostak](0.50 s)
   PVar_TCC2.............................proved - complete   [shostak](0.36 s)
   PVarSingle_TCC1.......................proved - complete   [shostak](0.37 s)
   PVarSingle_TCC2.......................proved - complete   [shostak](0.37 s)
   pvarsingle_aux........................proved - complete   [shostak](0.29 s)
   pvarsingle............................proved - complete   [shostak](0.72 s)
   pvarsingle_null_aux...................proved - complete   [shostak](0.27 s)
   pvar_null_pc..........................proved - complete   [shostak](0.18 s)
   pvar_null_pf..........................proved - complete   [shostak](0.68 s)
```

```
   pvarsingle_null.....................proved - complete   [shostak](0.14 s)
   pvarsingle_cons_remove..............proved - complete   [shostak](0.61 s)
   pvarsingle_remove...................proved - complete   [shostak](0.38 s)
   pvarsingle_elem_count...............proved - complete   [shostak](0.45 s)
   pvarsingle_apppv....................proved - complete   [shostak](0.92 s)
   Theory totals: 16 formulas, 16 attempted, 16 succeeded (6.68 s)

Proof summary for theory l_properties
   member_append.......................proved - complete   [shostak](0.33 s)
   allmembers_TCC1.....................proved - complete   [shostak](0.50 s)
   prop_allmembers.....................proved - complete   [shostak](0.32 s)
   allmembers_refl.....................proved - complete   [shostak](0.04 s)
   allmembers_null.....................proved - complete   [shostak](0.17 s)
   allmembers_app......................proved - complete   [shostak](0.25 s)
   allmembers_cons.....................proved - complete   [shostak](0.24 s)
   allmembers_cons_back................proved - complete   [shostak](0.21 s)
   allmembers_allmembers...............proved - complete   [shostak](0.23 s)
   remove_TCC1.........................proved - complete   [shostak](0.36 s)
   remove_TCC2.........................proved - complete   [shostak](0.33 s)
   remove_append.......................proved - complete   [shostak](0.24 s)
   prop_rem............................proved - complete   [shostak](0.22 s)
   rem_notmem..........................proved - complete   [shostak](0.20 s)
   rem_member..........................proved - complete   [shostak](0.38 s)
   allmembers_remove...................proved - complete   [shostak](0.24 s)
   allmembers_remove_back..............proved - complete   [shostak](0.21 s)
   prop_app............................proved - complete   [shostak](0.33 s)
   length_null.........................proved - complete   [shostak](0.49 s)
   cons_equal..........................proved - complete   [shostak](0.19 s)
   elem_count_notmem...................proved - complete   [shostak](0.73 s)
   elem_count_append...................proved - complete   [shostak](0.79 s)
   Theory totals: 22 formulas, 22 attempted, 22 succeeded (7.00 s)

Proof summary for theory l_append_induction
   list_length_null....................proved - complete   [shostak](0.52 s)
   list_length_notnull.................proved - complete   [shostak](0.57 s)
   remove_last_TCC1....................proved - complete   [shostak](0.42 s)
   remove_last_length..................proved - complete   [shostak](0.63 s)
   remove_last_prop_TCC1...............proved - complete   [shostak](2.00 s)
   remove_last_prop....................proved - complete   [shostak](0.54 s)
   list_append_induction...............proved - complete   [shostak](0.62 s)
   Theory totals: 7 formulas, 7 attempted, 7 succeeded (5.30 s)

Proof summary for theory resolution
   extend_clause_TCC1..................proved - complete   [shostak](0.59 s)
   extend_clause_rem...................proved - complete   [shostak](0.67 s)
   extend_clause_mem...................proved - complete   [shostak](0.29 s)
   build_tube_TCC1.....................proved - complete   [shostak](0.34 s)
   build_tube_null.....................proved - complete   [shostak](0.10 s)
   build_tube_member...................proved - complete   [shostak](0.25 s)
   aggfilter_TCC1......................proved - complete   [shostak](0.54 s)
   aggfilter_TCC2......................proved - complete   [shostak](0.41 s)
   aggfilter_member....................proved - complete   [shostak](0.30 s)
   aggfilter_member_tc.................proved - complete   [shostak](0.34 s)
   aggfilter_mem_agg_tc................proved - complete   [shostak](0.27 s)
   aggfilter_allmem....................proved - complete   [shostak](0.27 s)
   resolution_TCC1.....................proved - complete   [shostak](0.33 s)
   resolution_allmembers...............proved - complete   [shostak](0.30 s)
   asoc_pc_TCC1........................proved - complete   [shostak](0.36 s)
   asoc_pc_TCC2........................proved - complete   [shostak](0.35 s)
   asoc_pc.............................proved - complete   [shostak](0.57 s)
   asoc_pc_app.........................proved - complete   [shostak](0.24 s)
   is_compl_closed.....................proved - complete   [shostak](0.27 s)
```

```
    asoc_pc_mem.....................proved - complete   [shostak](0.38 s)
    asoc_pf_TCC1....................proved - complete   [shostak](0.50 s)
    asoc_pf_null....................proved - complete   [shostak](0.14 s)
    asoc_pf.........................proved - complete   [shostak](0.24 s)
    asoc_pf_pi......................proved - complete   [shostak](0.09 s)
    asoc_pf_app.....................proved - complete   [shostak](0.21 s)
    subst_pc_posmem.................proved - complete   [shostak](0.40 s)
    subst_pc_lem....................proved - complete   [shostak](0.94 s)
    subst_pc_notmem.................proved - complete   [shostak](1.19 s)
    subst_pf_lem....................proved - complete   [shostak](1.04 s)
    subst_pf........................proved - complete   [shostak](0.18 s)
    rem_asoc_pc.....................proved - complete   [shostak](0.39 s)
    del_plitn_agg_app...............proved - complete   [shostak](0.33 s)
    del_plitn_agg_closelit..........proved - complete   [shostak](0.53 s)
    del_plitn_agg_shift.............proved - complete   [shostak](0.65 s)
    del_plitn_tc_app................proved - complete   [shostak](0.26 s)
    del_plitn_tc_aggfilter..........proved - complete   [shostak](0.42 s)
    subst_pf_corresp................proved - complete   [shostak](0.46 s)
    subst_pf_corresp_allmem.........proved - complete   [shostak](0.39 s)
    openplits_equal_TCC1............proved - complete   [shostak](0.36 s)
    openplits_equal_prop............proved - complete   [shostak](0.26 s)
    openplits_equal_close_lit.......proved - complete   [shostak](0.42 s)
    openplits_equal_notmem..........proved - complete   [shostak](0.23 s)
    openplits_equal_closed..........proved - complete   [shostak](0.20 s)
    pval_pc_or......................proved - complete   [shostak](0.64 s)
    pval_pf_and.....................proved - complete   [shostak](0.40 s)
    pval_pf_member..................proved - complete   [shostak](0.40 s)
    pval_aggshift_aux...............proved - complete   [shostak](1.19 s)
    pval_aggshift...................proved - complete   [shostak](0.08 s)
    pval_close_lit..................proved - complete   [shostak](0.56 s)
    pval_tc_agg_zero................proved - complete   [shostak](0.24 s)
    reslem_agg_agg..................proved - complete   [shostak](0.78 s)
    reslem_agg_tcneg................proved - complete   [shostak](0.70 s)
    reslem_tc_tc....................proved - complete   [shostak](0.71 s)
    reslem..........................proved - complete   [shostak](0.24 s)
    soundness.......................proved - complete   [shostak](0.22 s)
    allmembers_pvarsingle...........proved - complete   [shostak](0.35 s)
    allm_pc_rem.....................proved - complete   [shostak](0.49 s)
    allmem_pvar_rem.................proved - complete   [shostak](0.32 s)
    allmem_pc_notmem................proved - complete   [shostak](0.47 s)
    allmembers_subst_app............proved - complete   [shostak](0.98 s)
    del_agg_agg.....................proved - complete   [shostak](0.74 s)
    del_agg_tcneg...................proved - complete   [shostak](0.37 s)
    del_tc_tc.......................proved - complete   [shostak](0.33 s)
    del_res.........................proved - complete   [shostak](0.40 s)
    mem_agg_tc_neg..................proved - complete   [shostak](0.60 s)
    allmem_agg_tcneg................proved - complete   [shostak](0.30 s)
    allmem_tc_tc_aux................proved - complete   [shostak](0.71 s)
    allmem_tc_tc....................proved - complete   [shostak](0.32 s)
    allmem_res_aux..................proved - complete   [shostak](0.59 s)
    allmem_res......................proved - complete   [shostak](0.26 s)
    del_plitn_closed................proved - complete   [shostak](0.36 s)
    del_plitn_detect................proved - complete   [shostak](0.43 s)
    closeaggs_agg_agg...............proved - complete   [shostak](0.47 s)
    closeaggs_agg_tc................proved - complete   [shostak](1.12 s)
    closeaggs_tc_tc.................proved - complete   [shostak](1.27 s)
    resolution_app_lpv..............proved - complete   [shostak](0.22 s)
    compl_aux.......................proved - complete   [shostak](2.77 s)
    completeness....................proved - complete   [shostak](0.08 s)
    Theory totals: 78 formulas, 78 attempted, 78 succeeded (37.11 s)

Grand Totals: 143 proofs, 143 attempted, 143 succeeded (65.42 s)
```

Literaturverzeichnis

[Adl94] Leonard M. Adleman. Molecular Computation of Solutions to Combinatorial Problems. *Science*, 266:1021–1024, 1994.

[Adl96] Leonard M. Adleman. On Constructing a Molecular Computer. In R. Lipton and E. Baum, editors, *DNA Based Computers*, DIMACS: series in Discrete Mathematics and Theoretical Computer Science, pages 1–21. American Mathematical Society, 1996.

[Baa05] F. Baader. Grundlagen der Theoretischen Informatik. Skript zur Lehrveranstaltung, 2005.

[Blö03] Johannes Blömer. Einführung in Algorithmen und Komplexität. Script zur Vorlesung, Universität Paderborn, 2003.

[Bre92] Manfred Bretz. *Algorithmen und Berechenbarkeit*. Vieweg, 1992.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. *Workshop on Industrial-Strength Formal Specification Techniques*, April 1995. http://www.csl.sri.com/papers/wift-tutorial/.

[Día03] Carmen Graciani Díaz. *Especificación y verificación de programas moleculares en PVS*. Dissertation, Universidad de Sevilla, Spanien, 2003.

[DK] Mark J. Daley and Lila Kari. Trends and Developments in DNA Computing. Department of Computer Science, University of Western Ontario, London, Ontario, Canada.

[DMJ02] Carmen Graciani Díaz, F. J. Martín Mateos, and M. J. Pérez Jiménez. Specification of Adleman's Restricted Model Using an Automated Reasoning System: Verification of Lipton's Experiment. In C. Calude, M.J. Dinneen, and F. Peper, editors, *Unconventional Models of Computation. Third International Conference, UMC 2002, Kobe, Japan, October 15-19, 2002. Proceedings*, volume 2509 of *Lecture Notes in Computer Science*, pages 126–136. Springer, 2002.

[FCLL00] Dirk Faulhammer, Anthony R. Cukras, Richard J. Lipton, and Laura F. Landweber. Molecular computation: RNA solutions to chess problems. *PNAS*, pages 1385–1389, 2000.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, Berlin, 1996.

[Höl01] Steffen Hölldobler. *Logik und Logikprogrammierung*. Synchron Wissenschaftsverlag der Autoren, 2001.

[HS04] Thomas Hinze and Monika Sturm. *Rechnen mit DNA - Eine Einführung in Theorie und Praxis*. R. Oldenbourg Wissenschaftsverlag München, ISBN 3-486-27530-5, 2004.

[Lip95] Richard J. Lipton. Using DNA to Solve NP-Complete Problems. *Science*, 268:542–545, April 1995.

[LPJ+03] In-Hee Lee, Ji-Yoon Park, Hae-Man Jang, Young-Gyu Chai, and Byoung-Tak Zhang. DNA Implementation of Theorem Proving with Resolution Refutation in Propositional Logic. In *DNA 8: Revised Papers from the 8th International Workshop on DNA Based Computers*, pages 156–167, London, UK, 2003. Springer-Verlag.

[Mal98] Carlo C. Maley. DNA Computation: Theory, Practice, and Prospects. *Evolutionary Computation*, 6(3):201–229, 1998.

[OS97] Sam Owre and Natarajan Shankar. The Formal Semantics of PVS. Technischer Report SRI-CSL-97-2, SRI International, Menlo Park, CA, 1997.

[OSRSC01a] Sam Owre, Natarajan Shankar, John M. Rushby, and D. W. J. Springer-Calvert. PVS Language Reference. SRI International, Menlo Park, CA, November 2001.

[OSRSC01b] Sam Owre, Natarajan Shankar, John M. Rushby, and D. W. J. Springer-Calvert. PVS Prover Guide. SRI International, Menlo Park, CA, November 2001.

[OSRSC01c] Sam Owre, Natarajan Shankar, John M. Rushby, and D. W. J. Springer-Calvert. PVS System Guide. SRI International, Menlo Park, CA, November 2001.

[PRS98] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *DNA Computing: New Computing Paradigms*. Springer-Verlag, 1998.

[Rei90] Karl Rüdiger Reischuk. *Einführung in die Komplexitätstheorie*. B.G. Teubner Stuttgart, 1990.

[Sch95] Uwe Schöning. *Logik für Informatiker*. Spektrum, Akademischer Verlag, 1995.

[Tew05a] Hendrik Tews. Komplexpraktikum: Interaktives Beweisen mit PVS, 2005. Script, Technische Universität Dresden, `http://wwwtcs.inf.tu-dresden.de/~tews`.

[Tew05b] Hendrik Tews. PVS Überlebenshilfe, September 2005. http://wwwtcs.inf.tu-dresden.de/~tews.

[UHK02] Hiroki Uejima, Masami Hagiya, and Satoshi Kobayashi. Horn Clause Computation by Self-assembly of DNA Molecules. In *DNA 7: Revised Papers from the 7th International Workshop on DNA-Based Computers*, pages 308–320, London, UK, 2002. Springer-Verlag.

[vHP02] F. von Henke and H. Pfeifer. PVS Kurzanleitung, 2002. Universität Ulm. www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/Inferenzsysteme/WS0203/pvs-hilfe.pdf.

[vHP06] F. von Henke and H. Pfeifer. Maschinelles Beweisen mit PVS, WS 2005/2006. Vorlesungsskript, Universität Ulm. http://www.informatik.uni-ulm.de/ki/Edu/Praktika/Maschinelles.Beweisen/WS0506.

[Wik] Wikipedia. Die Freie Enzyklopädie. Desoxyribonukleinsäure. http://de.wikipedia.org/wiki/DNA.

[WJMP00] Piotr Wąsiewicz, Tomasz Janczak, Jan J. Mulawka, and Andrzej Płucienniczak. The Inference Based on Molecular Computing. *International Journal of Cybernetics and Systems*, 31(3):283–315, April 2000.

Stichwortverzeichnis