Simon Bäumler

Modulares Beweisen temporallogischer Eigenschaften paralleler Programme

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
Fakultät für Angewandte Informatik
Universität Augsburg, 2010

Gutachter: Prof. Dr. Wolfgang Reif
Prof. Dr. Alexander Knapp

Tag der mündlichen Prüfung: 15. Juli 2010

Prüfer: Prof. Dr. Wolfgang Reif
Prof. Dr. Alexander Knapp
Prof. Dr. Elisabeth Andre

Kurzfassung

Infolge der zunehmenden Vernetzung und des Einsatzes von Mehrkernprozessoren, gewinnt die Verifikation von parallelen Programmen zunehmend an Bedeutung. Mit dem Theorembeweiser KIV und der temporalen Logik ITL$^+$ steht für solche Programme ein intuitiv anzuwendendes Beweiswerkzeug zur Verfügung.

In der vorliegenden Arbeit wird die Einbettung einer modularen Beweismethode – des Rely-Guarantee-Paradigmas – in die Logik ITL$^+$ beschrieben. Mit dieser Methode kann ein komplexer Beweis auf einzelne Komponenten aufgeteilt und damit deutlich vereinfacht werden. Eine Besonderheit dieses Ansatzes ist, dass sowohl die Theorie als auch die praktischen Anwendungen direkt im ITL$^+$-Kalkül bewiesen werden können, was den Ansatz sehr flexibel macht. Diese Flexibilität wird anhand der Erweiterung der Rely-Guarantee-Methode um Verfeinerung und komplexe Lebendigkeitseigenschaften gezeigt, die Anwendung dieser Methode an komplexen Fallstudien demonstriert. Einen besonderen Schwerpunkt bilden dabei lock-freie Algorithmen, eine Klasse von parallelen Programmen, deren Verifikation ein sehr aktuelles Forschungsgebiet ist.

Danksagung

Bei allen, die mich bei der Erstellung dieser Arbeit unterstützt haben, möchte ich mich herzlich bedanken.

Mein besonderer Dank gilt Prof. Dr. Wolfgang Reif für seine Ratschläge, Geduld und Unterstützung, ohne die diese Arbeit nicht entstanden wäre.

Danken möchte ich außerdem:

Dr. Gerhard Schellhorn, der immer bereit war, in zahlreichen Diskussionen bis spät in die Nacht die Feinheiten temporaler Logik zu erörtern;

meinem Zweitgutachter, Prof. Dr. Alexander Knapp, der mich schon bei meiner Diplomarbeit betreut und so auf den Weg gebracht hat, diese Arbeit zu erstellen;

Dr. Michael Balser, der mit seiner Dissertation die Grundlagen für diese Arbeit geschaffen und mit hilfreichen Hinweisen geholfen hat;

Bogdan Tofan, Dr. Jonathan Schmitt und Florian Nafz, die durch ihre eigenen Arbeiten immer wieder Anregungen zur Verbesserung gegeben haben.

Allen Mitarbeitern am Lehrstuhl für Softwaretechnik, danke ich für die kollegiale Hilfsbereitschaft und das gute Arbeitsklima.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Nebenläufige Systeme spielen in der Informatik eine immer größere Rolle. Computer werden über das Internet zunehmend vernetzt und es werden ständig neuartige Dienste entwickelt, die auf verschiedenste Art Parallelität nutzen. Ein Beispiel dafür ist der aktuelle Trend in Richtung „Cloud Computing“. Ein anderes Einsatzgebiet, in dem mittlerweile Parallelität in hohem Maße eine Rolle spielt, stellen so genannte „embedded devices“ dar, also Steuerungseinrichtungen von Maschinen oder Fahrzeugen. Moderne Autos besitzen zum Beispiel ein ganzes Netzwerk aus Computern, Sensoren und Steuerungen, die alle miteinander interagieren.

In den letzten Jahren zeichnet sich ein weiterer Trend ab. Fortschritte in der Rechenleistung werden hauptsächlich durch Multi-Core-Architekturen erreicht. Mittlerweile haben fast alle im Handel verfügbaren Desktop-Prozessoren zwei oder mehr Kerne und Prozessoren mit zwölf Kernen sind bereits auf dem Markt erhältlich. Daher wird es auch in der Informatik immer wichtiger, parallele Algorithmen zu entwickeln, die den Zuwachs an Rechenleistung durch Multi-Core-Architekturen ausnutzen. In diesem Zusammenhang gewinnen lock-freie Algorithmen zunehmend an Bedeutung, die einen effizienten Zugriff nebenläufiger Prozesse auf gemeinsame Datenstrukturen ermöglichen. Wegen der hohen Komplexität solcher Algorithmen spielt die formale Analyse dieser Algorithmen eine wichtige Rolle.

Parallele Systeme sind besonders schwierig zu entwickeln, da hier nicht nur die Korrektheit der einzelnen Komponenten sichergestellt werden muss, sondern vor allem auch das korrekte Zusammenspiel dieser Komponenten. Dieses ist besonders komplex, da der zeitliche Ablauf, die unterschiedlichen Interaktionsmöglichkeiten der verschiedenen Komponenten und die verschiedenen Schedulings der Prozesse berücksichtigt werden müssen. Klassische Validierungstechniken wie Software-Testen sind bei der Vielzahl von Interaktionen aller Komponenten meistens nicht ausreichend, um die Korrektheit des Systems sicherzustellen.

Daher bilden die formale Analyse und Verifikation von parallelen Systemen ein aktuelles und wichtiges Forschungsgebiet, um die Korrektheit solcher Systeme sicherzustellen. Automatische Methoden – allen voran „Model Checking“ (siehe z. B. [CGP00, BK08]) – wurden in diesem Bereich erfolgreich eingesetzt,

um Fehler im Design oder in der Implementierung zu finden. Dabei benötigen viele dieser Methoden einen begrenzten Zustandsraum, um Verifikationsergebnisse liefern zu können. In der aktuellen Forschung versucht man, diese Einschränkung durch Abstraktionstechniken zu umgehen und so selbst Systeme mit unendlichem Zustandsraum beherrschbar zu machen. Diese spezialisierten Techniken können sehr gute Resultate erzielen, sind jedoch häufig auf eine Problemklasse spezialisiert.

Ein alternativer Ansatz zur Verifikation sind interaktive Techniken, die auf einer Logik mit Beweiskalkül basieren. Diese Techniken können sehr natürlich mit unendlichen Zustandsräumen umgehen und sind meist sehr flexibel einsetzbar. Dabei ist es wichtig, dass es die Technik erlaubt, Systeme verständlich zu spezifizieren und die Beweise intuitiv zu führen. Temporallogik-Kalküle spielen im Zusammenhang mit nebenläufigen Systemen eine besondere Rolle, da sie erlauben, den zeitlichen Ablauf der einzelnen Komponenten zu beschreiben. Viele dieser Kalküle sind jedoch nicht einfach oder nicht intuitiv anzuwenden. Mit ITL$^+$ wurde von Balser [Bal05] ein auf Vorarbeiten von Moszkowski [Mos86] basierender Temporallogik-Kalkül entwickelt, der als Beweisprinzip symbolische Ausführung erlaubt. Dieses Prinzip hat sich bereits bei sequenziellen Programmen als einfach zu verwendende und intuitive Technik bewährt, die es erlaubt Systeme in einer den Programmiersprachen ähnlichen Syntax zu spezifizieren. ITL$^+$ besitzt außerdem einen kompositionalen Interleavingoperator, der Eigenschaften wie Fairness und blockierende Programme unterstützt und die einfache Abstraktion von Komponenten erlaubt. Der Kalkül wurde in das KIV-System [BRSS99] integriert, so dass werkzeuggestützte Beweise möglich sind.

Für große interleaved-parallele Systeme ist allerdings die alleinige Abstraktion der Komponenten oft nicht ausreichend, um die komplexe Struktur eines Systems entscheidend zu reduzieren, da alle möglichen Abläufe berücksichtigt werden müssen. Eine allgemeine Technik, die einen Beweis über das gesamte System vermeidet, wird „modulares Beweisen“ (engl. „compositional verification“) genannt. Sie wurde zum ersten Mal von Dijkstra [Dij65] beschrieben. Das Grundprinzip dieser Technik wird von de Roever et al. in [dRdBH$^+$01] wie folgt beschrieben:

> „That a program meets its specification should be verified on the basis of the specification of the constituent components only, without additional need for information about the interior construction of those components.“

Die vielleicht verbreitetste modulare Beweistechnik ist das „Rely-Guarantee“-Paradigma. Dieses wurde zum ersten Mal von Jones [Jon83a] und von Misra & Chandy [MC81] (hier unter dem Namen „Assumption-Commitment“) beschrieben. Der Grundgedanke dieses Paradigmas ist, dass jedes Programm bestimmte Annahmen (Relys) über das Verhalten der Umgebung trifft, um das gewünschte Verhalten zu garantieren. Üblicherweise wird bei dieser Technik ein Modularisierungstheorem benutzt, das eine Menge von Beweisverpflichtungen angibt, die gezeigt werden müssen, um ein bestimmtes Verhalten des Gesamtsystems nachzuweisen. Im Idealfall kommen in diesen Beweisverpflichtungen nur einzelne Komponenten des nebenläufigen Systems vor. Dies führt zu mehreren Beweisen

von handhabbarer Größe, da vermieden wird, alle unterschiedlichen parallelen Abläufe des Gesamtsystems betrachten zu müssen. Beispiele aus der Literatur für diese Technik sind zu finden bei [AL95, CC96, CGP00, dRdBH+01].

Das Ziel dieser Arbeit ist es, eine modulare auf dem Rely-Guarantee-Paradigma basierende Beweistechnik für ITL$^+$ vorzustellen. Der wissenschaftliche Beitrag dieser Arbeit ist folgender:

Zum einen wird gezeigt, wie eine Rely-Guarantee Technik in die Logik ITL$^+$ integriert werden kann. Diese erlaubt es, aus den Garantien der Komponenten eine Gesamtgarantie für das System nachzuweisen. Sie ist im Prinzip ähnlich wie die bekannten Rely-Guarantee Techniken. Die ausdrucksstarke Logik ITL$^+$ kann aber darüber hinaus zur Spezifikation der einzelnen Komponenten des Gesamtsystems benutzt werden und symbolische Ausführung kann zum Nachweis der lokalen Garantien verwendet werden.

Aufbauend auf diese Technik werden außerdem die Verfeinerungs- und Lebendigkeitseigenschaften eines Systems mit Hilfe von ITL$^+$ untersucht. Bei den Verfeinerungsbeweisen kann dabei im Unterschied zu bisherigen Verfeinerungsverfahren mit anderen Techniken auf Rückwärtssimulation oder „Prophecy"-Variablen verzichtet werden. Als Lebendigkeitseigenschaften wurde die so genannte „Lock-Freedom"-Eigenschaft untersucht, mit der gezeigt wird, dass eine gegenseitige Blockierung nebenläufiger Prozesse nicht statt finden kann. Im Unterschied zu bisherigen Verfahren ist eine explizite Konstruktion einer wohlfundierten Ordnung nicht erforderlich.

Die praktische Anwendung all dieser Techniken wird anhand von Fallstudien demonstriert. In diesem Zusammenhang werden vor allem lock-freie Algorithmen eingehender untersucht. Als Beispiele dazu werden Treiber-Stack- und Michael-Scott-Queue-Algorithmen im Hinblick auf Korrektheit und Lebendigkeit analysiert.

Alle Beweise können im ITL$^+$-Kalkül durchgeführt werden und sind vollständig werkzeuggestützt. Das heißt, sowohl der Beweis der Theorie selbst als auch die notwendigen Beweise bei der Anwendung können mit dem gleichen Kalkül mit einem einzigen Beweiswerkzeug durchgeführt werden.

Die Arbeit ist wie folgt gegliedert: In Kapitel 2 wird zunächst die Logik ITL$^+$ und der dazugehörige Kalkül vorgestellt. Im Zuge dieser Arbeit wurden bezüglich der orginalen Arbeit zu ITL$^+$ von Balser [Bal05] einige Veränderungen vorgenommen. Diese sind am Ende des Kapitels erläutert.

Kapitel 3 erklärt die Grundkonzepte des Rely-Guarantee-Paradigmas. Als erläuterndes Beispiel wird dazu der bekannte Semaphor-Algorithmus benutzt.

In Kapitel 4 wird eine Rely-Guarantee Methode für ITL$^+$ entwickelt, die es erlaubt, aus den Garantien der Komponenten eine Gesamtgarantie für ein interleaved paralleles System nachzuweisen. Die Anwendbarkeit dieser Technik wird anhand der Producer-Channel-Consumer Fallstudie demonstriert.

Neben Interleaving wurde auch ein Operator für synchrone Parallelität in ITL$^+$ integriert. Eine der größten in ITL$^+$ durchgeführten Fallstudien, das PROTOCURE-Project, benutzt diesen Operator. In Kapitel 5 wird gezeigt, wie sich die Rely-Guarantee-Methode auch für synchron parallele Systeme umsetzen lässt und zeigt deren Anwendung anhand der PROTOCURE-Fallstudie.

Ein relativ neues aber aktuelles Forschungsgebiet ist die Analyse und Veri-

fikation von lock-freien Algorithmen. In Kapitel 6 werden die Grundprinzipien dieser Algorithmenklasse erläutert und anschaulich am Treiber-Stack-Beispiel erklärt.

Die Korrektheit von lock-freien Algorithmen wird in Kapitel 7 untersucht. Dabei wird das in Kapitel 4 vorgestellte Theorem erweitert, so dass auch Verfeinerungseigenschaften von Komponenten mit der Rely-Guarantee-Technik gezeigt werden können. Die praktische Anwendung wird anhand der Treiber-Stack Fallstudie gezeigt.

In Kapitel 8 wird eine modulare Technik zum Nachweis von Lebendigkeit von lock-freien Algorithmen beschrieben. Auch hier wird die praktische Anwendung dieser Technik anhand der Treiber-Stack Fallstudie demonstriert.

Eine weitere Fallstudie im Bereich lock-freie Algorithmen wird in Kapitel 9 vorgestellt. Der in diesem Kapitel beschriebene Michael-Scott-Queue Algorithmus ist komplexer als der Treiber-Stack Algorithmus. Anhand dieses Algorithmus werden einige Besonderheiten der in dieser Arbeit entwickelten Technik beschrieben.

Die wichtigsten Ergebnisse sind in Kapitel 10 zusammengefasst. Ein Überblick über verwandte wissenschaftliche Arbeiten ist jeweils am Ende der betreffenden Kapitel 2, 4, 7, 8 und 9 zu finden.

Kapitel 2

Grundlagen

In diesem Kapitel wird die in dieser Arbeit verwendete Logik ITL$^+$ mit dem Kalkül zur symbolischen Ausführung beschrieben. ITL$^+$ wurde von Michael Balser im Rahmen seiner Dissertation [Bal05] entwickelt (dort noch unter dem Namen ITL – *Interval Temporal Logic*). Die hier beschriebene Logik wurde aber im Vergleich zu [Bal05] um einige Punkte erweitert. Da viele dieser Änderungen große Teile der Logik und des Kalküls betreffen, wird zum besseren Verständnis und zugunsten der Vollständigkeit die gesamte Logik in der aktuellen Fassung vorgestellt.

In Kapitel 2.1 wird zunächst die Syntax von ITL$^+$ vorgestellt. Die Semantik ist im darauf folgenden Kapitel 2.2 beschrieben. In den Kapiteln 2.3 bis 2.5 wird der Kalkül für ITL$^+$ beschrieben. Ein kurzes Beispiel in Kapitel 2.6 beschreibt die Anwendung des Kalküls. Schließlich gibt das Kapitel 2.8 einen Überblick über die in dieser Arbeit vorgestellten Neuerungen im Vergleich zu [Bal05]. Teile der hier beschriebenen Logik wurden mit einem Beitrag vom Autor in [BBN$^+$10, BBR08] veröffentlicht.

2.1 Syntax

In diesem Abschnitt wird die Syntax von ITL$^+$ – wie bei algebraischen Spezifikationen üblich – mit Hilfe von Signaturen definiert. Zur besseren Lesbarkeit werden unterstrichene Bezeichner $\underline{t}$ für eine Folge $t_1, \ldots, t_n$ von Bezeichnern verwendet. Der Ausdruck $\underline{t} \in T^+$ bedeutet also $t_1 \in T, \ldots, t_n \in T$ mit $n \geq 1$ während $\underline{t} \in T^*$ auch die leere Folge enthalten kann. Diese Schreibweise wird auch in anderem Zusammenhang benutzt. So steht $\underline{e} \in X_{\underline{t}}$ für $e_1 \in X_{t_1}, \ldots, e_n \in X_{t_n}$, $\underline{X} = \underline{e}$ für $X_1 = e_1, \ldots, X_n = e_n$ und mit $\underline{X}'$ (bzw. $\underline{X}''$) wird die Folge mit den gestrichenen (bzw. doppelt gestrichenen) Variablen aus $\underline{X}$ bezeichnet. Zusätzlich bezeichnet $\{\underline{t}\}$ die Menge, die alle Elemente aus der Folge $\underline{t}$ enthält.

Definition 1 *(Signatur* **SIG***).* Eine Signatur **SIG** = (**S**, **OP**, **PROC**, **SV**, **DV**) besteht aus

- einer endlichen Menge von Sorten **S**. Zu **S** wird die Menge von Typen **T**

definiert, welche die kleinste Menge mit

$$\mathbf{T} = \mathbf{S} \cup \{\underline{t} \Rightarrow t' \mid \underline{t} \in \mathbf{T}^+, t' \in \mathbf{T}\}$$

ist. D. h. $\mathbf{T}$ enthält alle Sorten aus $\mathbf{S}$ und für jede Folge von Argumenttypen $\underline{t} \in \mathbf{T}^+$ und jeden Zieltyp $t' \in \mathbf{T}$ den Funktionstyp $\underline{t} \Rightarrow t'$,

- einer endlichen Familie
$$\mathbf{OP} := (\mathbf{OP}_t)_{t \in \mathbf{T}}$$
von Operationen $\mathbf{OP}_t$ vom Typ t. Für $t \in \mathbf{S}$ ist $\mathbf{OP}_t$ eine Menge von *Konstanten*. Falls $t = (\underline{t} \Rightarrow t')$, dann ist $\mathbf{OP}_t$ eine Menge von Funktionen mit *Argumenttypen* $\underline{t}$ und *Zieltyp* t',

- einer endlichen Familie
$$\mathbf{PROC} := (\mathbf{PROC}_{\underline{t}}^{\underline{t}'})_{\underline{t},\underline{t}' \in \mathbf{T}^*}$$
von Prozeduren $\mathbf{PROC}_{\underline{t}}^{\underline{t}'}$ mit *Value*-Parametersorten $\underline{t}$ und *Reference*-Parametersorten $\underline{t}'$,

- einer Familie
$$\mathbf{SV} := (SV_t)_{t \in \mathbf{T}}$$
von abzählbar unendlichen Mengen von *statischen Variablen* $\mathbf{SV}_t$ und

- einer Familie
$$\mathbf{DV} := (DV_t)_{t \in \mathbf{T}}$$
von abzählbar unendlichen Mengen von *dynamischen Variablen* $\mathbf{DV}_t$. □

Die Menge $\mathbf{S}$ bildet die Basissorten von **SIG**, aus denen sich die Menge der Typen $\mathbf{T}$ ableitet, die neben den Basissorten auch die Funktionssorten $\underline{t} \Rightarrow t'$ enthält. Darauf aufbauend wird die Menge $\mathbf{OP}$ aller Operationen und die Menge der Prozeduren $\mathbf{PROC}$ definiert. Schließlich ist $\mathbf{SV}$ die Menge aller statischen Variablen und $\mathbf{DV}$ die Menge aller dynamischen Variablen. Als Konvention werden statische Variablen mit Kleinbuchstaben und dynamische Variablen mit Großbuchstaben bezeichnet. Ferner bezeichnet die Menge $\mathbf{V}$ alle Variablen einer Signatur, d. h. $\mathbf{V} = \mathbf{SV} \cup \mathbf{DV}$. Die Menge $\mathbf{V}_t$ bezeichnet alle Variablen vom Typ t, d. h. $\mathbf{V}_t = \mathbf{SV}_t \cup \mathbf{DV}_t$.

Für Operationen

$$op \in \mathbf{OP}_{t_1,\ldots,t_n \Rightarrow t}$$

wird im Folgenden oft auch die Schreibweise

$$op : \ t_1 \times \ldots \times t_n \ \rightarrow t$$

benutzt. Weiter wird im Folgenden davon ausgegangen, dass in jeder Signatur die Menge der Sorten $\mathbf{S}$ mindestens die Sorten *Bool* und *Nat* enthält (diese stehen für boolsche Werte und natürliche Zahlen). Ferner enthält $\mathbf{OP}$ die für diese beiden Sorten üblichen Operatoren, das heißt für die Sorte *Bool* die Operatoren

$$\begin{aligned} \text{true, false} &: \ \mathit{Bool} \\ \neg &: \ \mathit{Bool} \rightarrow \mathit{Bool} \\ \wedge\ ,\ \vee\ ,\ \rightarrow\ ,\ \leftrightarrow &: \ \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \end{aligned}$$

und für die Sorte *Nat* die Operatoren:

$$\begin{aligned} 0 &: Nat \\ \text{succ, pred} &: Nat \to Nat \\ + &: Nat \times Nat \to Nat \end{aligned}$$

Als nächstes wird die Syntax von Ausdrücken definiert. Die Logik unterscheidet zwischen logischen Ausdrücken **E** und Programmausdrücken **P**. Zunächst werden die logischen Ausdrücke definiert, die Syntax der Programmausdrücke folgt im Anschluss.

Definition 2 *(Syntax der logischen Ausdrücke* **E***).* Für eine gegebene Signatur **SIG** ist die Menge der *logischen Ausdrücke* **E** eine Familie

$$\mathbf{E} := (\mathbf{E}_t)_{t \in \mathbf{T}} \ ,$$

von *getypten logischen Ausdrücken* $\mathbf{E}_t$. Dabei ist für alle $t \in \mathbf{T}$ die Menge $\mathbf{E}_t$ definiert durch die kleinste Menge mit folgenden Eigenschaften:

- (Variablen) Sei $x \in \mathbf{SV}_t$ und $V \in \mathbf{DV}_t$, dann ist
  - $x \ \in \mathbf{E}_t$ (statische Variable),
  - $V \ \in \mathbf{E}_t$ (ungestrichene dynamische Variable),
  - $V' \ \in \mathbf{E}_t$ (gestrichene dynamische Variable) und
  - $V'' \ \in \mathbf{E}_t$ (doppelt gestrichene dynamische Variable).
- (Ausdruck) Sei $e' \in \mathbf{E}_{\underline{t} \Rightarrow t'}$ und $\underline{e} \in \mathbf{E}_{\underline{t}}$, dann ist
  - $e'(\underline{e}) \ \in \mathbf{E}_{t'}$.
- (Lambda-Ausdruck) Sei $e \in \mathbf{E}_{t'}$ und $\underline{x} \in \mathbf{SV}_{\underline{t}}$, dann ist
  - $\lambda \underline{x}.e \ \in \mathbf{E}_{\underline{t} \Rightarrow t'}$.
- (Bedingter-Ausdruck) Sei $e \in \mathbf{E}_{Bool}$ und $\varphi, \psi \in \mathbf{E}_{\underline{t}}$, dann ist
  - $e \supset \varphi \, ; \, \psi \ \in \mathbf{E}_{\underline{t}}$.
- (Funktionsaufruf) Sei $f \in \mathbf{OP}_t$, dann ist
  - $f \in \mathbf{E}_t$.
- (Gleichung) Sei $e_1, e_2 \in \mathbf{E}$, dann ist
  - $e_1 = e_2 \ \in \mathbf{E}_{Bool}$.
- (Quantoren) Sei $\varphi \in \mathbf{E}_{Bool}$ und $V \in (\mathbf{SV} \cup \mathbf{DV})$, dann ist
  - $\exists V. \ \varphi \ \in \mathbf{E}_{Bool}$ (Existenzquantor) und
  - $\forall V. \ \varphi \ \in \mathbf{E}_{Bool}$ (Allquantor).
- (Temporallogische Operatoren) Sei $\varphi, \psi \in \mathbf{E}_{Bool}$, dann sind

  - φ **until** $\psi \in \mathbf{E}_{Bool}$,
  - $\circ\, \varphi \in \mathbf{E}_{Bool}$ (*Strong-Next*),
  - $\mathbf{A}\, \varphi \in \mathbf{E}_{Bool}$ (All-Pfadquantor)

- (Abgeleitete temporallogische Operatoren) Sei $\varphi, \psi \in \mathbf{E}_{Bool}$, dann sind
  - **last** $\in \mathbf{E}_{Bool}$,
  - $\Box\, \varphi \in \mathbf{E}_{Bool}$ (*Always*),
  - $\Diamond\, \varphi \in \mathbf{E}_{Bool}$ (*Eventually*),
  - φ **unless** $\psi \in \mathbf{E}_{Bool}$,
  - $\bullet\, \varphi \in \mathbf{E}_{Bool}$ (*Weak-Next*),
  - $\mathbf{E}\, \varphi \in \mathbf{E}_{Bool}$ (Existiert-Pfadquantor).

- (Programmabschluss) Sei $P \in \mathbf{P}$ und $\underline{V} \in \mathbf{DV}_{\underline{t}}$, dann ist
  - $[P]_{\underline{V}} \in \mathbf{E}_{Bool}$.

- (Programmblocking)
  - **blocked** $\in \mathbf{E}_{Bool}$.

□

Dynamische Variablen V können in ungestrichener, gestrichener und doppelt gestrichener Form als V, V' und V'' vorkommen. Diese Unterscheidung wird später benutzt, um Systemschritte und Umgebungsschritte voneinander zu unterscheiden. Ausdrücke, die keine temporallogischen Operatoren (auch keine abgeleiteten) und Programmabschlüsse enthalten, werden auch *prädikatenlogische Ausdrücke* genannt. Die hier benutzten temporallogischen Grundoperatoren *Strong-Next* (manchmal auch nur als *Next* bezeichnet) und **until** sind aus ITL [CMZ02] übernommen, kommen in ähnlicher Form aber auch in den meisten anderen Temporallogiken vor. Der All-Pfadquantor **A** entspricht dem gleichen Quantor aus CTL* [EH86]. Die weiteren üblichen LTL- und ITL-Operatoren werden von diesen drei Grundoperatoren abgeleitet.

Die Syntax der Programmausdrücke **P** ist wie folgt definiert:

Definition 3 *(Syntax der Programmausdrücke* **P***).* Für eine gegebene Signatur **SIG** ist die Menge der *Programmausdrücke* **P** die kleinste Menge, die folgende Bedingungen erfüllt:

- (Ausdrücke) Sei $e \in \mathbf{E}_{Bool}$, dann ist
  - $e \in \mathbf{P}$

- (Prozeduraufruf) Sei $proc \in \mathbf{PROC}^{\underline{t'}}_{\underline{t}}$, $\underline{e} \in \mathbf{E}_{\underline{t}}$ und $\underline{V} \in \mathbf{DV}_{\underline{t'}}$, dann ist
  - $proc(\underline{e}; \underline{V}) \in \mathbf{P}$

- (Programmoperatoren) Sei $V_i \in \mathbf{DV}_{t_i}$ und $e_i \in \mathbf{E}_{t_i}$. Dann sind
  - $V_0 := e_0, \ldots, V_n := e_n \in \mathbf{P}$ (parallele Zuweisung).
  - $V_0 := ? \in \mathbf{P}$ (nichtdeterministische Zuweisung) und

  - **skip** $\in \mathbf{P}$ (Stotterschritt).

- (Kontrollstruktur) Sei $V \in \mathbf{DV}_t$, $e \in \mathbf{E}_t$, $\alpha, \alpha_1, \alpha_2 \in \mathbf{P}$ und $\varphi \in \mathbf{E}_{Bool}$ ein prädikatenlogischer Ausdruck. Dann sind

  - **if*** φ **then** α_1 **else** α_2 $\in \mathbf{P}$ (Fallunterscheidung),
  - **while*** φ **do** α $\in \mathbf{P}$ (Schleife),
  - **if** φ **then** α_1 **else** α_2 $\in \mathbf{P}$ (Fallunterscheidung),
  - **while** φ **do** α $\in \mathbf{P}$ (Schleife).

- (lokale Variable) Sei $\underline{V} \in \mathbf{DV}_{\underline{t}}$, $\underline{e} \in \mathbf{E}_{\underline{t}}$, $\varphi \in \mathbf{E}_{Bool}$ und $\alpha, \beta \in \mathbf{P}$. Dann ist

  - **let** $\underline{V} = \underline{e}$ **in** α $\in \mathbf{P}$,
  - **choose** $\underline{V}$ **with** φ **in** α **ifnone** β $\in \mathbf{P}$.

- (Synchronisation) Sei $\varphi \in \mathbf{E}_{Bool}$ ein prädikatenlogischer Ausdruck. Dann ist

  - **await** φ $\in \mathbf{P}$.

- (Programmkomposition) Sei $\alpha, \beta \in \mathbf{P}$. Dann ist

  - $\alpha;\beta$ $\in \mathbf{P}$ (Chop-Operator) und
  - α^* $\in \mathbf{P}$ (Stern-Operator).

- (Paralleloperatoren) Sei $\alpha, \beta \in \mathbf{P}$, dann sind

  - $\alpha \mid\!\lfloor\ \beta$ $\in \mathbf{P}$ (Left-Merge-Operator),
  - $\alpha \mid\!\lfloor^b\ \beta \in \mathbf{P}$ (Blocked-Left-Merge-Operator),
  - $\alpha \parallel \beta$ $\in \mathbf{P}$ (Interleaved-Operator) □

Programmausdrücke werden hier und im Folgenden mit den Buchstaben α und β bezeichnet, während logische Ausdrücke mit φ und ψ bezeichnet werden. Alle logischen Ausdrücke können auch unter einem Programmabschluss als Programmausdrücke verwendet werden. Dies wird häufig zur Programmabstraktion benutzt.

Bei den Kontrollstrukturen **if** und **while** gibt es zwei Formen, einmal mit und einmal ohne Stern. Der Unterschied zwischen diesen beiden Formen ist, dass die Sternformen **if*** und **while*** keinen Schritt zur Auswertung der Bedingung φ benötigt, während dies bei **if** und **while** genau einen Schritt dauert.

Die *Chop*- und Stern-Operatoren wurden von ITL übernommen, werden hier aber nur im Kontext von Programmausdrücken eingesetzt. Der *Chop*-Operator entspricht der Komposition zweier Programme, während der Stern-Operator einem *Loop* entspricht.

Der Left-Merge-Parallel-Operator wird später zur Spezifikation der schwachen Fairness des Interleavings benötigt. Er besagt, dass das linke Teilprogramm den nächsten Schritt ausführt.

Um Klammern zu sparen, sind Präzedenzregeln für die Operatoren definiert. Diese sind in Anhang A zu finden. Die freien Variablen einer Formel φ werden

durch die Funktion free(φ) bestimmt. Deren Definition kann in Anhang B gefunden werden. Mit

$$\varphi_a^{a_0}$$

wird die Substitution der Variable a in φ durch die Variable a_0 definiert. Formal kann die Variablensubstitution analog zu [Bal05] definiert werden.

2.2 Semantik

Im folgenden Abschnitt wird die Semantik von ITL$^+$ beschrieben. Dabei werden zuerst Zustände und Intervalle basierend auf Algebren definiert. Diese werden dann benutzt, um eine Semantik für die im vorherigen Kapitel eingeführten Operatoren anzugeben.

2.2.1 Zustände und Intervalle

Die Semantik von ITL$^+$ basiert auf Algebren. Diese werden benutzt, um die Semantik der Sorten, Operationen und Prozeduren einer gegebenen Signatur **SIG** zu definieren.

Definition 4 *(Algebra $\mathcal{A}$).* Sei $\mathbf{SIG} = (\mathbf{S}, \mathbf{OP}, \mathbf{PROC}, \mathbf{SV}, \mathbf{DV})$ eine Signatur. Eine *Algebra* $\mathcal{A}_{\mathbf{SIG}}$ besteht aus

- nichtleeren Mengen $\mathbf{D}_s$ (genannt *Domains*) für alle Basissorten $s \in \mathbf{S}$,
- für jede Funktionssorte $t \in T$ mit $t = \underline{t} \Rightarrow t'$ wird die Menge $\mathbf{D}_t$ induktiv definiert als die Menge der Funktionen von $\mathbf{D}_{\underline{t}}$ nach $\mathbf{D}_{t'}$,
- Funktionen
$$f_{\mathcal{A}} \in \mathbf{D}_t$$
für jede Operation $f \in \mathbf{OP}_t$ und
- Relationen
$$\begin{aligned} proc_{\mathcal{A}} : \quad & \mathbf{D}_{t_1} \times \cdots \times \mathbf{D}_{t_i} \\ & \times \left(\mathbf{D}_{t'_1} \times \cdots \times \mathbf{D}_{t'_j}\right)^n \times \left(\mathbf{D}_{t'_1} \times \cdots \times \mathbf{D}_{t'_j}\right)^n \\ & \times \mathbf{D}_{\text{bool}}^n \times \mathbf{D}_{\text{bool}}^n \end{aligned}$$
für jede Prozedur $proc \in \mathbf{PROC}_{t_1,\ldots,t_i}^{t'_1,\ldots,t'_j}$ und alle $n \in \mathbb{N}_0 \cup \{\infty\}$. □

Für jede Algebra $\mathcal{A}$ wird angenommen, dass $\mathbf{D}_{Bool} = \{\text{tt}, \text{ff}\}$. Ferner wird angenommen, dass $\mathbf{D}_{Nat} = \mathbb{N}_0$. Für die Operationen „true“, „false“, $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$ auf *Bool* und 0, „succ“, „pred“ und + auf *Nat* wird jeweils die Standardsemantik angenommen. Die Semantik von Prozeduren $proc_{\mathcal{A}}$ wird durch eine Relation definiert. Diese gliedert sich in drei Teile: Für jeden *Value*-Parameter enthält sie einen Wert (dies ist der an die Prozedur übergebene Wert) und für jeden Referenz-Parameter zwei endliche oder unendliche Folgen von Werten (die erste Folge gibt die Werte vor jedem Programmschritt, die

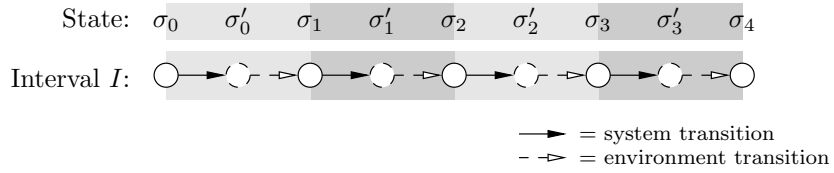


Abbildung 2.1: Ein Intervall als Folge von Zuständen

zweite Folge die Werte nach jedem Programmschritt an). Schließlich enthält die Relation noch zwei Folgen von booleschen Werten. Diese sind für die spezielle Variable *blk* reserviert und beschreiben, ob Schritte blockiert sind (siehe dazu auch Definition 5, bzw. Abschnitt 2.2.5).

Algebren geben den Operationen und Prozeduren eine Semantik. Um die Variablen einer gegebenen Signatur auszuwerten, werden Zustände benutzt:

Definition 5 *(Zustand).*
Sei **SIG** = (**S**, **OP**, **PROC**, **SV**, **DV**) eine Signatur und $\mathcal{A}_{\mathbf{SIG}}$ eine Algebra. Ein Zustand ist eine Familie $\sigma := (\sigma_t)_{t \in \mathbf{T}}$ von Abbildungen

$$\sigma_t : \mathbf{V}_t \to \mathbf{D}_t \ ,$$

die statische und dynamische Variablen auf Domainelemente des passenden Typs abbilden. Für σ_{Bool} gilt

$$\sigma_{Bool} : \mathbf{V}_{Bool} \cup \{blk\} \to \mathbf{D}_{Bool} \ ,$$

d. h., σ_{Bool} bildet zusätzlich noch die spezielle Variable *blk* auf einen booleschen Wert ab. □

Zustände bilden also Variablen auf Werte ab. Zusätzlich gibt es eine spezielle boolesche dynamische Variable $blk \notin \mathbf{V}$. Diese Variable wird benutzt, um anzuzeigen, ob ein Programm blockiert ist oder nicht.

Damit lassen sich Intervalle definieren, die eine endliche oder unendliche Folge von Zuständen sind:

Definition 6 *(Intervall).*
Sei **SIG** = (**S**, **OP**, **PROC**, **SV**, **DV**) eine Signatur, $\mathcal{A}_{\mathbf{SIG}}$ eine Algebra und $n \in \mathbb{N}_0$. Ein *Intervall* ist eine endliche Folge

$$I = (\sigma_0, \sigma'_0, \sigma_1, \ldots, \sigma'_{n-1}, \sigma_n)$$

von $2n + 1$ Zuständen oder eine unendliche Folge

$$I = (\sigma_0, \sigma'_0, \sigma_1, \ldots)$$

mit $\sigma_0(a) = \sigma'_i(a) = \sigma_{i+1}(a)$ für alle statischen Variablen $a \in \mathbf{SV}$ für alle $i < n$ bzw. für alle $i \in \mathbb{N}_0$, falls I unendlich.

Ungestrichene Zustände σ_i werden *Grundzustände* genannt und gestrichene Zustände σ'_i werden *Zwischenzustände* genannt. Das Paar (σ_i, σ'_i) bestehend aus je einem aufeinanderfolgenden Grund- und Zwischenzustand heißt *Systemtransition* und das Paar $(\sigma'_i, \sigma_{i+1})$ bestehend aus je einem aufeinanderfolgenden Zwischen- und Grundzustand heißt *Umgebungstransition*.

Die *Länge* eines Intervalls I ist definiert durch $|I| := n$. Das Intervall I ist *leer*, falls $|I| = 0$, und *unendlich*, falls $|I| = \infty$. Die Menge aller Intervalle wird mit $\mathcal{I}$ bezeichnet. □

Abweichend von der üblichen Definition von Intervallen haben die Intervalle in ITL$^+$ dabei zusätzliche Zwischenzustände σ' zwischen den „normalen“ Zuständen (die in ITL$^+$ Grundzustände heißen). Diese werden benutzt, um zwischen Schritten des Systems und den Schritten der Umgebung eines Systems zu unterscheiden. Zum besseren Verständnis der Intervalle in ITL$^+$ mit System- und Umgebungsschritten wird eine informelle grafische Darstellung in Abbildung 2.1 gezeigt.

Die Unterscheidung zwischen Sytem- und Umgebungsschritten in Intervallen ist ähnlich zu *Traces* [CK95] bzw. zu *Aczel-traces* [dRdBH$^+$01]. Aczel-traces beginnen mit einem Umgebungsschritt, während die hier und in [CK95] verwendeten Intervalle (bzw. Traces) mit einem Systemschritt beginnen. In [CK95] und [dRdBH$^+$01] wird allerdings – im Unterschied zu ITL$^+$– nur ein generisches Framework ohne eine Logik oder einem Kalkül dafür vorgestellt.

Der erste Zustand σ_0 eines Intervalls enthält die initiale Belegung aller Variablen. Der zweite Zustand σ'_0 enthält die Variablenwerte direkt nach dem ersten Systemschritt, während der darauf folgende Zustand σ_1 die Variablenwerte nach dem ersten Umgebungsschritt enthält. Da sich die Werte der statischen Variablen auf dem gesamten Intervall nicht ändern, sind sie auch in allen drei Zuständen σ_0, σ'_0 und σ_1 gleich. Ein leeres Intervall (d. h. $|I| = 0$) besteht nur aus einem Zustand σ_0.

Die folgenden Funktionen werden benutzt, um Teilintervalle zu selektieren, beziehungsweise ein neues Intervall zu konstruieren.

Definition 7. Sei $I = (\sigma_0, \sigma'_0, \sigma_1, \ldots)$ ein endliches oder unendliches Intervall und $0 \leq i \leq j \leq |I|$. Dann ist

$$\begin{aligned} I|_i &:= (\sigma_i, \ldots) \\ I|^i &:= (\sigma_0, \ldots, \sigma_i) \\ I|_i^j &:= (I|^j)|_i \\ (\sigma, \sigma', I) &:= (\sigma, \sigma', \sigma_0, \sigma'_0, \sigma_1, \ldots) \, . \end{aligned}$$

□

Die Funktion $I|_i$ liefert den Postfix von I ab dem i-ten Zustand, während die Funktion $I|^i$ den Präfix von I bis (einschließlich) dem i-ten Zustand liefert. Mit der Funktion $I|_i^j$ wird der Infix des Intervalls I vom j-ten bis zum i-ten Zustand berechnet. Schließlich wird mit (σ, σ', I) das Intervall bezeichnet, das mit den beiden Zuständen σ und σ' beginnt und dann genauso wie I fortgesetzt wird.

Um auf die einzelnen Zustände eines Intervalls zugreifen zu können, werden die folgenden Funktionen benutzt.

Definition 8. Sei $I = (\sigma_0, \sigma_0', \sigma_1, \ldots)$ ein Intervall und $n = |I|$. Dann ist

$$\begin{aligned} I(i) &:= \begin{cases} \sigma_i & \text{falls } i \leq n \\ \sigma_n & \text{sonst} \end{cases} \\ I(i)' &:= \begin{cases} \sigma_i' & \text{falls } i < n \\ \sigma_n & \text{sonst} \end{cases} \\ I(i)'' &:= I(i+1)\ . \end{aligned}$$

□

Die Funktionen $I(i)$ und $I'(i)$ liefern den i-ten Grund- bzw. Zwischenzustand des Intervalls I. Die doppelt gestrichene Funktion $I''(i)$ gibt den nächsten (d. h. den $i+1$-ten) Grundzustand zurück. Beim Zugriff auf einen Zustand hinter dem letzten Zustand geben alle drei Funktionen den letzten Grundzustand zurück, das heißt die Werte aller Variablen werden als konstant nach dem letzten Zustand angenommen.

Mit der Schreibweise

$$I[\underline{x} \leftarrow \underline{a}]$$

wird das Intervall bezeichnet, das aus I entsteht, wenn der Wert der statischen Variablen $\underline{x}$ in allen Zuständen auf die Werte $\underline{a}$ abgebildet wird. Zusätzlich gilt

$$I_1 =_{\underline{V}} I_2,$$

wenn die Intervalle I_1 und I_2 die gleiche Länge haben und in allen Zuständen bis auf die Variablen $\underline{V}$ die gleichen Werte haben. Eine formale Definition von $I[\underline{x} \leftarrow \underline{a}]$ und $I_1 =_{\underline{V}} I_2$ ist in [Bal05] angegeben.

Definition 9 *(Semantik von Ausdrücken).* Sei $\mathcal{A}$ eine Algebra und I ein endliches oder unendliches Intervall. Die *Semantik* einer ITL$^+$ Formel wird durch die Funktion

$$[\![.]\!]_{\mathcal{A},I} : \mathbf{E}_t \to \mathbf{D}_t$$

definiert. □

Die Funktion $[\![.]\!]_{\mathcal{A},I}$ bildet Ausdrücke auf Werte der entsprechenden Domain ab. Diese Funktion wird benutzt, um die Semantik von Ausdrücken zu beschreiben.

Zusätzlich werden noch die folgenden abkürzenden Schreibweisen benutzt: Für eine Formel $\varphi \in \mathbf{E}_{Bool}$ wird auch

$$\mathcal{A}, I \models \varphi \quad \text{gdw.} \quad [\![\varphi]\!]_{\mathcal{A},I} = \mathrm{tt}$$

geschrieben. Im Folgenden wird eine feste Algebra $\mathcal{A}$ als gegeben angenommen (z. B. als Modell einer algebraischen Spezifikation). Daher wird die Algebra auch weggelassen und verkürzt $[\![\varphi]\!]_I = \mathrm{tt}$ bzw. $I \models \varphi$ geschrieben. Außerdem wird oft die Semantik eines Ausdrucks e_1 auf die Semantik eines anderen Ausdrucks e_2 zurückgeführt. Dafür wird die Schreibweise

$$e_1 \ :\equiv \ e_2$$

benutzt. Dies steht verkürzend für

$$[\![e_1]\!]_I := [\![e_2]\!]_I\ .$$

2.2.2 Prädikatenlogische Formeln

Die Semantik von prädikatenlogischen Formeln ist wie folgt definiert:

Definition 10. Sei **SIG** eine Signatur, $\mathcal{A}_{\mathbf{SIG}}$ eine Algebra und I ein Intervall. Dann gilt

$$\begin{array}{rcll}
[\![x]\!]_{\mathcal{A},I} & := & I(0)(x) & \text{für alle } x \in \mathbf{SV} \\
[\![V]\!]_{\mathcal{A},I} & := & I(0)(V) & \text{für alle } V \in \mathbf{DV} \\
[\![V']\!]_{\mathcal{A},I} & := & I(0)'(V) & \text{für alle } V \in \mathbf{DV} \\
[\![V'']\!]_{\mathcal{A},I} & := & I(0)''(V) & \text{für alle } V \in \mathbf{DV} \\
[\![f]\!]_{\mathcal{A},I} & := & f_{\mathcal{A}} & \text{für alle } f \in \mathbf{OP} \\
[\![e'(\underline{e})]\!]_{\mathcal{A},I} & := & [\![e']\!]_{\mathcal{A},I}([\![\underline{e}]\!]_{\mathcal{A},I}) & \text{für alle } e' \in \mathbf{E}_{\underline{t}\Rightarrow t'} \text{ und } \underline{e} \in \mathbf{E}_{\underline{t}}
\end{array}$$

Für alle $e \in \mathbf{E}_{Bool}$ und $\varphi, \psi \in \mathbf{E}_{\underline{t}}$ ist

$$[\![e \supset \varphi\,;\,\psi]\!]_{\mathcal{A},I} := \begin{cases} [\![\varphi]\!]_{\mathcal{A},I} & \text{falls } [\![e]\!]_{\mathcal{A},I} = \mathrm{tt} \\ [\![\psi]\!]_{\mathcal{A},I} & \text{falls } [\![e]\!]_{\mathcal{A},I} = \mathrm{ff} \end{cases}$$

Für alle $\underline{x} \in \mathbf{SV}_{\underline{t}}$ und $e \in \mathbf{E}_{t'}$ ist $[\![\lambda \underline{x}.e]\!]_{\mathcal{A},I}$ eine Funktion $f : \mathbf{D}_{\underline{t}} \to \mathbf{D}_{t'}$ mit $f(\underline{a}) := [\![e]\!]_{\mathcal{A},I[\underline{x}\leftarrow\underline{a}]}$.

□

Zu beachten ist, dass nach Definition 8 die Variablenwerte nach dem letzten Zustand als konstant angenommen werden; für $|I| = n$ gilt also z. B. $I(n)'(V) = I(n)(V)$ und $I(n+1)(V) = I(n)(V)$. Statische und ungestrichene dynamische Variablen werden über dem ersten Grundzustand σ_0 des Intervalls $(\sigma_0, \sigma_0', \sigma_1, \ldots)$ ausgewertet, während die gestrichenen Variablen über dem ersten Zwischenzustand σ_0' und die doppelt gestrichenen Variablen über dem zweiten Grundzustand σ_1 ausgewertet werden. Funktionstypen und Operationen werden einfach durch ihre Anwendung ausgewertet. Die Auswertung eines Lambda-Ausdrucks $\lambda \underline{x}.e$ ergibt eine neue Funktion, deren Funktionswert durch die Auswertung von e bestimmt wird.

Als nächstes wird die Semantik von Gleichungen und Existenzquantoren definiert:

Definition 11 *(Semantik von Gleichungen).* Sei **SIG** eine Signatur, $\mathcal{A}_{\mathbf{SIG}}$ eine Algebra, I ein Intervall und $e_1, e_2 \in \mathbf{E}_t$. Dann ist

$$\mathcal{A}, I \models (e_1 = e_2) \quad \text{gdw.} \quad [\![e_1]\!]_{\mathcal{A},I} = [\![e_2]\!]_{\mathcal{A},I}$$

□

Definition 12 *(Semantik des Existenzquantors).*
Sei **SIG** eine Signatur, $\mathcal{A}_{\mathbf{SIG}}$ eine Algebra und I ein Intervall. Dann gilt für alle $\underline{V} \in \mathbf{SV} \cup \mathbf{DV}$ und $\varphi \in \mathbf{E}_{Bool}$

$$\mathcal{A}, I \models \exists\, \underline{V}.\varphi \quad \text{gdw.} \quad \text{es gibt ein } I_0 \text{ mit } I_0 =_{\underline{V}} I \text{ und } \mathcal{A}, I_0 \models \varphi$$

Die Semantik des Allquantors wird durch Umschreibung in den Existenzquantor definiert:

$$\forall\, \underline{V}.\varphi \quad :\equiv \quad \neg\, \exists\, \underline{V}.\neg\, \varphi$$

□

2.2.3 Temporallogische Formeln

Die Semantik der temporalen Operatoren kann analog zu ihrer Semantik in *Interval Temporal Logic* [CMZ02] definiert werden, obwohl ITL$^+$ mit den einfach und doppeltgestrichenen Variablen eine andere Definition der Intervalle zugrunde liegt. Dies hat jedoch keinen Einfluss auf die Semantik der Temporaloperatoren. Zur formalen Definition der temporalen Operatoren reicht es aus, die Semantik der beiden Grundoperatoren „$\circ$“ (*Strong-Next*) und **until** zu definieren, die anderen Operatoren können von diesen abgeleitet werden.

Definition 13 *(Temporale Grundoperatoren).*
Seien $\varphi, \psi \in \mathbf{E}_{Bool}$ und I ein Intervall. Dann gilt:

$$I \models \varphi \textbf{ until } \psi \quad \text{gdw} \quad \text{es gibt ein } n \leq |I| \text{ mit} \qquad I|_n \models \psi$$
$$\text{und } I|_m \models \varphi \text{ für alle } 0 \leq m < n,$$
$$I \models \circ\, \varphi \quad \text{gdw} \quad |I| \geq 1 \text{ und } I|_1 \models \varphi\ .$$

□

Die Semantik von φ **until** ψ besagt, dass ψ irgendwann auf dem Intervall I erfüllt ist und dass φ in jedem Zustand davor gilt. Diese Definition entspricht dem Standard in temporalen Logiken (siehe z. B. [MP91a] oder [Eme90]). Die Semantik für den *Strong-Next*-Operator $\circ\, \varphi$ fordert, dass es in dem Intervall I einen nächsten Zustand gibt, in dem φ gilt. Da im Unterschied zu den meisten Temporallogiken, in ITL$^+$ auch endliche und sogar leere Intervalle erlaubt sind, fordert der *Strong-Next*-Operator zusätzlich zum Standard, dass das Intervall auch einen nächsten Zustand besitzt. Er entspricht damit dem gleichen Operator in ITL [Mos85].

Die anderen temporalen Operatoren können von diesen beiden Operatoren abgeleitet werden:

Definition 14 *(Abgeleitete temporale Operatoren).*

$$\begin{aligned}
\textbf{last} &:\equiv \neg \circ \text{true} \\
\Box\, \varphi &:\equiv \neg \Diamond \neg\, \varphi \\
\Diamond\, \varphi &:\equiv \text{true } \textbf{until } \varphi \\
\varphi \textbf{ unless } \psi &:\equiv \Box\, \varphi \vee \varphi \textbf{ until } \psi \\
\bullet\, \varphi &:\equiv \neg \circ \neg\, \varphi\ .
\end{aligned}$$

□

Die Formel **last** gilt genau dann, wenn das Intervall nur einen Zustand enthält.

Der *Always*-Operator $\Box\ \varphi$ besagt, dass φ von jetzt an in jedem Zustand gilt, während der *Eventually*-Operator $\Diamond\ \varphi$ genau dann wahr ist, wenn es einen Zustand im Intervall gibt, in dem φ gilt. Die Semantik des **unless**-Operators ist ähnlich zu der des **until**-Operators, nur dass φ **unless** ψ schon gilt, wenn φ auf dem gesamten Intervall erfüllt ist. Der schwache *Next*-Operator $\bullet\ \varphi$ fordert nicht, dass es einen nächsten Zustand gibt, sondern ist auf dem Intervall mit einem Zustand immer erfüllt.

Die Semantik der Pfadquantoren wird folgendermaßen definiert:

Definition 15 *(Pfadquantoren).*

$$I \models \mathbf{A}\ \varphi \quad \text{gdw} \quad \text{für alle } I_0 \text{ mit } I_0(0) = I(0) \text{ gilt } I_0 \models \varphi$$

Die Semantik des Exist-Pfadquantors wird von dem All-Pfadquantor abgeleitet:

$$\mathbf{E}\ \varphi \quad :\equiv \quad \neg\ \mathbf{A}\ \neg\ \varphi$$

□

2.2.4 Sequentielle Programme

Die Programmoperatoren können in ITL$^+$ ähnlich eingebettet werden wie in ITL [Mos85]. Allerdings gibt es auch einige Unterschiede in ITL$^+$ zu ITL [Mos85]. So beeinflussen Zuweisungen in letzteren nur die zugewiesenen Variablen, während Zuweisungen in wirklichen Programmiersprachen alle Variablen des Systems unverändert lassen. In [Bal05] wurde daher eine so genannte *Frame-Assumption* $\lceil V_1, \ldots, V_n \rceil$ benutzt, die alle Variablen außer V_1 bis V_n im nächsten Systemschritt unverändert lässt. Dieser Ansatz hat sich als zu restriktiv erwiesen. Das Hauptproblem dabei ist, dass durch diese Art der Frame-Assumption keine freien Variablen mehr existieren. Wenn zum Beispiel in einer Sequenz die Formel $\exists\ V.\varphi$ auf der linken Seite einer Sequenz vorkommt, kann hier nicht die im Sequenzenkalkül übliche Regel *exists left* angewandt werden, um die Variable V zu instanzieren. Dadurch ist ein Beweis über quantifizierte Formeln mit den üblichen Regeln unmöglich. Weitere Probleme ergeben sich zum Beispiel auch bei Verfeinerungsbeweisen.

Um diese Probleme zu lösen, wurde ein von der Grundidee her ähnlicher Ansatz wie in TLA [Lam94, Lam02] gewählt. Mit der neuen *Frame-Assumption* $[P]_{\underline{V}}$ beeinflusst das Programm P nur die Systemvariablen, die mit der Variablenmenge $\underline{V}$ angegeben werden; alle anderen Variablen bleiben von der Formel unbeeinflusst. Die Syntax (siehe Abschnitt 3) wurde so gewählt, dass Programme immer von einer Frame-Assumption umschlossen werden, die so den Variablenkontext des Programms angeben.

Im Vergleich zur Frame-Assumption hat der entsprechende Operator $[A]_{\underline{e}}$ in TLA (dort *Action Operator* genannt) eine unterschiedliche Semantik:[1]

$$[A]_{\underline{e}} \quad := \quad A \vee \left(\bigwedge_{e \in \underline{e}} e' = e \right)$$

[1] Dynamische Variablen wie e werden in TLA klein geschrieben.

Entweder die Aktionsformel[2] A wird ausgeführt, oder das System stottert und alle Variablen in $\underline{e}$ bleiben unverändert. Die Aktionsformel A ist dabei eine prädikatenlogische Formel, die mittels ungestrichener und gestrichener Variablen Zustandsübergänge beschreibt. Doppelt gestrichene Variablen wie in ITL$^+$ gibt es in TLA nicht. Stattdessen werden die Stotterschritte benutzt, um Umgebungsverhalten semantisch zuzulassen. Der Action Operator ermöglicht also das Stottern einer Formel, was in TLA eine gewollte Eigenschaft ist. Der Operator hat aber ebenfalls nur Auswirkung auf das Stotterverhalten einer Formel, nicht aber auf die in A beschriebenen Schritte. Daher muss z. B. in TLA bei einer Zuweisung – im Unterschied zu der hier vorgestellten ITL$^+$– separat in A spezifiziert werden, dass außer der zugewiesenen Variable keine andere Variable in $\underline{e}$ verändert wird, um das gewünschte Verhalten einer Zuweisung zu erhalten.

Da in ITL$^+$ Programme und Formeln gemischt vorkommen können, ist es möglich, dass eine prädikatenlogische oder temporallogische Formel innerhalb einer Frame-Assumption vorkommt. Auf diese Formeln hat die Frame-Assumption keine Auswirkung, das heißt, sie kann einfach weggelassen werden.

Definition 16 *(Semantik von Ausdrücken in Frame-Assumptions).*
Sei $e \in \mathbf{E}_{Bool}$. Dann ist

$$[e]_{\underline{V}} \quad :\equiv \quad e$$

□

Wie schon im vorherigen Abschnitt für die abgeleiteten temporalen Operatoren wird die Semantik der meisten Programmoperatoren dadurch definiert, indem sie auf die Semantik von schon definierten Operatoren zurückgeführt wird. Die Semantik von Zuweisungen kann damit wie folgt definiert werden:

Definition 17 *(Semantik von Zuweisungen).*

$$\begin{aligned}
[A_1 := e_1, \ldots, A_n := e_n]_{\underline{V}} \quad :\equiv \quad & A'_1 = e_1 \wedge \ \ldots \ \wedge A'_n = e_n \\
& \wedge \bigwedge_{X \in \{\underline{V}\} \setminus \{A_1, \ldots, A_n\}} X' = X \\
& \wedge \circ \textbf{last} \\
[A := ?]_{\underline{V}} \quad :\equiv \quad & \bigwedge_{X \in \{\underline{V}\} \setminus \{A\}} X' = X \\
& \wedge \circ \textbf{last}
\end{aligned}$$

□

Die Zuweisung $A := e$ setzt den Wert von A nach dem Systemschritt auf e. Alle anderen Variablen X aus $\underline{V}$ bleiben unverändert (d. h. $X' = X$). Das $\circ$ **last** legt fest, dass die Zuweisung genau einen Schritt dauert, das heißt, $I \models [A := e]_{\underline{V}}$ impliziert $|I| = 1$. Bei einem *Random Assignment* $A :=?$ wird der Wert von A nach dem Systemschritt einfach unspezifiziert belassen.

[2]Eine Aktionsformel in TLA ist eine prädikatenlogische Formel die Zustandsübergänge beschreibt.

Ähnlich wie die Semantik für Zuweisungen wird auch die Semantik für **skip** definiert:

Definition 18 *(Semantik von* **skip***).*

$$[\mathbf{skip}]_{\underline{V}} \quad :\equiv \quad \bigwedge_{X \in \underline{V}} X' = X \wedge \circ \mathbf{last}$$

□

Ein $[\mathbf{skip}]_{\underline{V}}$ entspricht also einem einzelnen Stotterschritt, in dem keine der Variablen $\underline{V}$ geändert werden.

Zur Programmkomposition werden der *Chop*-Operator „;“ und der *Stern*-Operator „*“ benutzt. Deren Semantik ist wie folgt definiert:

Definition 19 *(Semantik der Programmkomposition).* Sei $\alpha, \beta \in \mathbf{P}$ und $\underline{V} \in \mathbf{DV}$. Dann ist

$$\begin{aligned}
I \models [\alpha;\, \beta]_{\underline{V}} \quad \text{gdw} \quad & \text{es gibt ein } n \leq |I| \text{ mit } I|^n \models [\alpha]_{\underline{V}} \text{ und } I|_n \models [\beta]_{\underline{V}} \\
\text{oder } & |I| = \infty \text{ und } I \models [\alpha]_{\underline{V}} \\
I \models [\alpha^*]_{\underline{V}} \quad \text{gdw} \quad & |I| = 0 \\
\text{oder } & \text{es gibt } n_i \text{ mit } 0 = n_0 < n_1 < \cdots < n_m < |I|\,, \\
& \quad \text{so dass} \quad \text{für alle } i \text{ mit } 0 \leq i < m \text{ gilt } I|_{n_i}^{n_{i+1}} \models [\alpha]_{\underline{V}} \\
& \qquad \text{und } I|_{n_m} \models [\alpha]_{\underline{V}} \\
\text{oder } & \quad |I| = \infty \\
& \quad \text{und es gibt unendlich viele } n_i \text{ mit } 0 = n_0 < n_1 < \ldots\,, \\
& \qquad \text{so dass für alle } i \text{ gilt } I|_{n_i}^{n_{i+1}} \models [\alpha]_{\underline{V}}
\end{aligned}$$

□

Der *Chop*-Operator $[\alpha; \beta]_{\underline{V}}$ gilt, wenn entweder $[\alpha]_{\underline{V}}$ auf dem gesamten Intervall gilt oder wenn es möglich ist, das Intervall so in ein Präfix- und ein Suffix-Teilintervall zu trennen, dass auf dem Präfix-Teilintervall $[\alpha]_{\underline{V}}$ und auf dem Suffix-Teilintervall $[\beta]_{\underline{V}}$ gilt. Wenn α und β reine Programme sind, entspricht dieser Operator damit genau der sequentiellen Komposition von Programmen. So wie der *Chop*-Operator der sequentiellen Komposition entspricht, so entspricht der Stern-Operator einer Programmschleife ohne Bedingung. Die Formel $[\alpha^*]_{\underline{V}}$ ist erfüllt, wenn das Intervall in n Teilintervalle geteilt werden kann (n kann hier 0 oder auch unendlich sein) und $[\alpha]_{\underline{V}}$ auf jedem dieser Teilintervalle gilt. Anders gesprochen muss es möglich sein, $[\alpha]_{\underline{V}}$ n-mal auf dem Intervall zu iterieren, damit $[\alpha^*]_{\underline{V}}$ gilt. Dieser Operator ist – unabhängig von der Formel α – immer auf dem leeren Intervall (d. h. $|I| = 0$) erfüllt. Dies entspricht dann 0 Iterationen.

Die Kontrollstrukturen kommen in zwei Formen vor: Zum einen die $*$-Form (z. B. **if*** p **then** α), bei der der Test p keinen extra Schritt benötigt, und zum anderen die normale Form (wie z. B. **if** p **then** α), bei der die Auswertung von p einen Schritt dauert. Die $*$-Formen werden üblicherweise zur Spezifikation

atomarer Operationen, wie z. B. die *Compare-and-Swap*-Operation in Kapitel 6, benutzt. Die formale Semantik für Kontrollstrukturen ist:

Definition 20 *(Semantik der Kontrollstrukturen).* Sei $\underline{V} \in \mathbf{DV}$, $\alpha, \alpha_1, \alpha_2 \in \mathbf{P}$ und $p \in \mathbf{E}_{Bool}$, wobei p ein prädikatenlogischer Ausdruck ist. Dann sind

$$\begin{aligned}
[\mathbf{if}^*\ \ p\ \mathbf{then}\ \alpha_1\ \mathbf{else}\ \alpha_2]_{\underline{V}} &:\equiv (p\ \rightarrow [\alpha_1]_{\underline{V}}) \wedge (\neg\, p\ \rightarrow [\alpha_2]_{\underline{V}}) \\
[\mathbf{while}^*\ \ p\ \mathbf{do}\ \alpha]_{\underline{V}} &:\equiv (p \wedge [\alpha]_{\underline{V}})^*; (\mathbf{last} \wedge \neg\, p) \\
[\mathbf{if}\ p\ \mathbf{then}\ \alpha_1\ \mathbf{else}\ \alpha_2]_{\underline{V}} &:\equiv [\mathbf{if}^*\ p\ \mathbf{then}\ (\mathbf{skip};\ \alpha_1)\ \mathbf{else}\ (\mathbf{skip};\ \alpha_2)]_{\underline{V}} \\
[\mathbf{while}\ p\ \mathbf{do}\ \alpha]_{\underline{V}} &:\equiv (p \wedge [\mathbf{skip}; \alpha]_{\underline{V}})^*; ([\mathbf{skip}]_{\underline{V}} \wedge \neg\, p)
\end{aligned}$$

□

Die Fallunterscheidung **if*** kann durch prädikatenlogische Implikation und Konjunktion ausgedrückt werden. Die Semantikdefinition der **while***-Schleife wird auf die Semantik des Stern-Operators zurückgeführt, der einer Schleife ohne Bedingung entspricht. Auch die beiden Formen ohne Stern, **if** und **while**, werden jeweils auf die entsprechenden Stern-Formen zurückgeführt. Dabei wird vor die auszuführenden Programmblöcke α, α_1 und α_2 noch ein **skip** vorangestellt. Dadurch wird erreicht, dass die Auswertung des Tests p genau einen Schritt benötigt. Beim **while**-Operator wird zusätzlich noch ein Stotterschritt beim Schleifenabbruch eingefügt, damit auch hier die Auswertung des Tests einen Schritt dauert.

Die Semantik der lokalen Variablendeklaration eines **let**-Ausdrucks wird auf den Existenzquantor zurückgeführt:

Definition 21 *(Semantik von lokaler Variablendeklaration).*
Sei $\alpha \in \mathbf{P}$, $e \in \mathbf{E}_t$, $A, A_0 \in \mathbf{DV}_t$ mit $A_0 \notin \mathrm{free}(e, \alpha)$.

$$[\mathbf{let}\ A = e\ \mathbf{in}\ \alpha]_{\underline{V}} \quad :\equiv \quad \exists\, A_0.\ A_0 = e \wedge [\alpha_A^{A_0}]_{A_0,\underline{V}} \wedge \Box\ A_0'' = A_0'$$

□

Die lokal deklarierte Variable A wird durch die existenzquantifizierte Variable A_0 ersetzt. Deren Wert wird mit e initialisiert und sie darf von der Umgebung nicht geändert werden. Zusätzlich wird die Variable A_0 zu den Programmvariablen von α dazu genommen, damit diese Variablen bei Zuweisungen etc. nicht geändert werden.

Die Semantik des **choose**-Ausdrucks kann, analog zu der des **let**-Ausdrucks, ebenfalls auf den Existenzquantor zurückgeführt werden:

Definition 22 *(Semantik der **choose**-Deklaration).*
Sei $\alpha, \beta \in \mathbf{P}$, $\varphi \in \mathbf{E}_{Bool}$, $A, A_0 \in \mathbf{DV}_t$ mit $A_0 \notin \mathrm{free}(\varphi, \alpha)$. Dann ist

$$\begin{aligned}
[\mathbf{choose}\ A\ \mathbf{with}\ \varphi\ \mathbf{in}\ \alpha\ \mathbf{ifnone}\ \beta]_{\underline{V}} \quad :\equiv \quad & \exists\, A_0.\quad \varphi_A^{A_0} \wedge [\alpha_A^{A_0}]_{A_0,\underline{V}} \\
& \qquad \wedge \Box\ A_0'' = A_0' \\
& \vee (\neg\, \exists\, A.\ \varphi) \wedge [\beta]_{\underline{V}}
\end{aligned}$$

□

Im Unterschied zur **let**-Deklaration kann der neu deklarierte Variablenwert bei **choose** alle Werte annehmen, bei denen der Ausdruck φ wahr ist.

Die Semantik von Prozeduren wird wie folgt definiert:

Definition 23 *(Semantik von Prozeduren).*
Sei $proc \in \mathbf{PROC}^{t'}_{t_1,\ldots,t_i}$, $e_1 \in \mathbf{E}_{t_1}, \ldots, e_i \in \mathbf{E}_{t_i}$, $\underline{V'} \in \mathbf{DV}_{\underline{t'}}$, und $n = |I|$.

$$
\begin{aligned}
[\![[proc(e_1,\ldots,e_i;\underline{V'})]_{\underline{V}}]\!]_I = \mathbf{tt} \quad :\Leftrightarrow & \\
proc_{\mathcal{A}} \ni \quad & [\![e_1]\!]_I \times \ldots \times [\![e_i]\!]_I \\
& \times (I(0)(\underline{V'}), I(1)(\underline{V'}), \ldots, I(n)(\underline{V'})) \\
& \times (I'(0)(\underline{V'}), I'(1)(\underline{V'}), \ldots, I'(n)(\underline{V'})) \\
& \times (I(0)(blk), I(1)(blk), \ldots, I(n)(blk)) \\
& \times (I'(0)(blk), I'(1)(blk), \ldots, I'(n)(blk))
\end{aligned}
$$

□

Die Ausdrücke der *Value*-Parameter werden über dem ersten Zustand ausgewertet. Für jede Variable der *Value*-Parameter wird der Werteverlauf der jeweiligen Variable – in ungestrichener und gestrichener Form – im Intervall zur Auswertung benutzt. Zusätzlich wird auch der Werteverlauf der *blk*-Variable in ungestrichener und gestrichener Form berücksichtigt.

2.2.5 Programmsynchronisierung

In ITL$^+$ werden sowohl synchron- als auch interleaved-parallele Ausführungen von Programmen unterstützt. Dabei kommunizieren die parallelen Programme über gemeinsame Variablen (engl. *shared variables*) miteinander. Synchronisiert werden die Programme mittels des **await**-Operators: **await** φ bedeutet, dass das Programm so lange blockiert ist (d. h., es werden nur Stotterschritte ausgeführt), solange φ zu false ausgewertet wird. Dass ein Programm blockiert ist, wird durch den Ausdruck **blocked** signalisiert. Die Auswertung dieses Ausdrucks wird auf die booleschen Variable *blk* zurückgeführt (dadurch wird die Semantik und die Implementierung von **blocked** vereinfacht). Formal wird **blocked** wie folgt definiert:

Definition 24 *(Programmblocking).*
Sei I ein Intervall. Dann gilt:

$$I \models \mathbf{blocked} \quad \text{gdw} \quad I(0)(blk) \neq I'(0)(blk)$$

□

Damit kann **await** wie folgt definiert werden:

Definition 25 *(Programmsynchronisierung).*
Sei $\varphi \in \mathbf{E}_{Bool}$. Dann ist

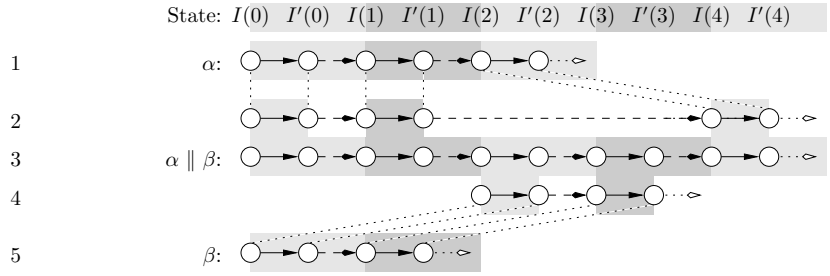


Abbildung 2.2: Interleaving von Intervallen

$$
\begin{aligned}
[\textbf{await}\ \ \varphi]_{\underline{V}} \quad :\equiv \quad & [\textbf{while}^* \neg \varphi \ \textbf{do} \quad \textbf{blocked} \\
& \qquad\qquad \wedge \textstyle\bigwedge_{X \in \underline{V}} X = X' \\
& \qquad\qquad \wedge \circ\, \textbf{last} \qquad\qquad]_{\underline{V}}
\end{aligned}
$$

□

Der **await**-Ausdruck blockiert, solange φ nicht wahr ist. Außerdem wird dabei keine Programmvariable geändert. Das heißt, der Prozess wartet aktiv darauf, dass die Umgebung die Bedingung erfüllt.

2.2.6 Interleaved-parallele Programme

Interleaved-parallele Programme werden durch den *Interleaving-Operator* „ $\parallel$ “ modelliert. Dabei können mit $\alpha \parallel \beta$ zwei beliebige Programme α und β parallel ausgeführt werden. Die Abbildung 2.2 gibt ein informelles Beispiel des Interleavings. In jedem Systemschritt wird entweder ein Systemschritt der Komponente α oder der Komponente β ausgeführt. Dabei werden die Umgebungsschritte einer Komponente durch die globalen Umgebungsschritte (d. h. Umgebungsschritte von $\alpha \parallel \beta$) und die Systemschritte der jeweils anderen Komponente gebildet. Bei einem geblockten Schritt einer Komponente wird jeweils der andere Prozess ausgeführt.

Formal wird die Semantik des Interleaving-Operators durch das Interleaving $I = I_1 \parallel I_2$ von zwei Intervallen I_1 und I_2 definiert. Dabei ergeben sich in jedem Schritt sechs mögliche Fälle aus I_1 und I_2 das neue Intervall I zu bilden:

1. Die erste (d. h. linke) Komponente terminiert im aktuellen Zustand: $I(0) = I_1(0)$ und die Ausführung wird mit dem zweiten Intervall fortgeführt, d. h. $I = I_2$.

2. Die erste Komponente macht einen nicht blockierten Schritt: $I(0) = I_1(0)$ und $I(0)' = I_1(0)'$ und das verbleibende Intervall $I|_1$ wird durch $I|_1 = I_1|_1 \parallel I_2$ gebildet.

3. Die erste Komponente ist im aktuellen Zustand $I(0)$ blockiert und ein Schritt der zweiten Komponente wird mit $I(0) = I_2(0)$ und $I(0)' = I_2(0)'$

ausgeführt. Die Systemschritte beider Komponenten werden dabei abgearbeitet und das Restintervall wird durch $I|_1 = I_1|_1 \parallel I_2|_1$ gebildet. Falls der Schritt der rechten Komponente ebenfalls blockiert ist, ergibt sich daraus ein blockierter Schritt des Gesamtintervalls I. Falls die zweite Komponente im aktuellen Zustand terminiert, macht die linke Komponente einen blockierten Schritt und bildet das Restintervall.

4. Die zweite Komponente terminiert.

5. Die zweite Komponente macht einen nichtblockierten Schritt.

6. Die zweite Komponente ist blockiert.

Zwei Intervalle können nur dann durch Interleaving miteinander kombiniert werden, wenn sie die gleichen statischen und dynamischen Variablen besitzen. Insbesondere müssen sich auch beide Unterintervalle die Variable *blk* teilen. Das Gesamtintervall ist nur dann geblockt, wenn beide Unterintervalle geblockt sind. Falls nur ein Unterintervall geblockt ist, wird der Systemschritt des anderen Unterintervalls ausgeführt.

Wie oben schon angedeutet, wird die Semantik des Interleaving-Operators durch das Interleaving von Intervallen $I_1 \parallel I_2$ definiert. Die Semantik des Interleavings zweier Formeln $\alpha \parallel \beta$ wird dann wie folgt definiert:

$$I \models \alpha \parallel \beta \quad \text{gdw.} \quad \text{es gibt } I_1, I_2 \text{ mit} \quad \begin{aligned} & I_1 \models \alpha \\ & \text{und } I_2 \models \beta \\ & \text{und } I \in I_1 \parallel I_2 \end{aligned}$$

Mit dieser Definition bezeichnet $I_1 \parallel I_2$ eine Menge von Intervallen. Um die Menge $I_1 \parallel I_2$ zu bestimmen, wird die Funktion $\|\!\lfloor_n$ mit

$$. \|\!\lfloor_n . : \mathbf{I} \times \mathbf{I} \to \mathcal{P}(\mathbf{I})$$

und $n \in \mathbb{N}_0$ benutzt. Informell bedeutet $I_1 \|\!\lfloor_n I_2$, dass I_1 in den nächsten n Schritten priorisiert wird. Das heißt, in den nächsten n Schritten macht I_1 immer dann einen Schritt, wenn es nicht blockiert ist. Danach wird die Priorisierung auf das andere Intervall I_2 umgeschaltet. Durch diesen Mechanismus der Priorisierung wird schwache Fairness des Interleavings erreicht.

Mit dem Operator $\|\!\lfloor_n$ kann der Interleaving Operator $\parallel$ wie folgt definiert werden:

$$I_1 \parallel I_2 :\equiv \bigcup_n (I_1 \|\!\lfloor_n I_2) \cup (I_2 \|\!\lfloor_n I_1) \ .$$

Informell entspricht $I_1 \|\!\lfloor_n I_2$ den Fällen *1. - 3.* der obigen Beschreibung des Interleavings und $I_2 \|\!\lfloor_n I_1$ entspricht den anderen drei Fällen *4. - 6.*.

Die Semantik der Funktion $\|\!\lfloor_n$ wird mit Hilfe von SOS-Regeln [Plo81] definiert. Dabei wird die Notation aus [AFV01] benutzt. Die SOS-Regeln sind in Tabelle 2.1 angegeben. Sie beschreiben alle möglichen Schritte von $I_1 \|\!\lfloor_n I_2$ oder beim Ausführen eines einzelnen Intervalls I (diese Regeln sind nötig, falls eine Komponente terminiert ist).

$$\frac{I_1 = (\sigma) \qquad I_2 = (\sigma, \sigma', I_2')}{(I_1 \parallel\!\lfloor_{n+1} I_2) \xrightarrow{\sigma,\sigma'} I_2'} \textit{ ilvl lst}$$

$$\frac{I_1 = (\sigma, \sigma', I_1') \qquad I_1 \models \neg\mathbf{blocked}}{(I_1 \parallel\!\lfloor_{n+1} I_2) \xrightarrow{\sigma,\sigma'} (I_1' \parallel\!\lfloor_{n} I_2)} \textit{ ilvl stp}$$

$$\frac{I_1 = (\sigma, \sigma_1', I_1') \qquad I_1 \models \mathbf{blocked} \qquad I_2 = (\sigma, \sigma_2', I_2')}{(I_1 \parallel\!\lfloor_{n+1} I_2) \xrightarrow{\sigma,\sigma_2'} (I_1' \parallel\!\lfloor_{n} I_2')} \textit{ ilvl blk}$$

$$\frac{I_1 = (\sigma, \sigma_1', I_1') \qquad I_1 \models \mathbf{blocked} \qquad I_2 = (\sigma)}{(I_1 \parallel\!\lfloor_{n+1} I_2) \xrightarrow{\sigma,\sigma_1'} I_1'} \textit{ ilvl blk lst}$$

$$\frac{(I_2 \parallel\!\lfloor_{n+1} I_1) \xrightarrow{\sigma,\sigma'} (I_2' \parallel\!\lfloor_{n} I_1')}{(I_1 \parallel\!\lfloor_{0} I_2) \xrightarrow{\sigma,\sigma'} (I_2' \parallel\!\lfloor_{n} I_1')} \textit{ ilvl swtch}_n\textit{, für } n \in \mathbb{N}_0$$

$$\frac{}{(\sigma, \sigma', I) \xrightarrow{\sigma,\sigma'} I} \textit{ stp}$$

$$\frac{}{(\sigma) \xrightarrow{\sigma} \emptyset} \textit{ lst}$$

Tabelle 2.1: Semantik des Interleaving-Operators

In der Regel *ilvl lst* wird der Fall behandelt, in dem das priorisierte Intervall terminiert (dies entspricht den Fällen *1.* bzw *4.* der informellen Beschreibung oben). Die Regel *ilvl stp* behandelt den Fall, in dem die priorisierte Komponente einen nichtblockierten Schritt macht (dies entspricht Fall *2.* bzw. *5.*). Mit der Regel *ilvl blk* wird ein blockierter Schritt der ersten Komponente behandelt, indem die zweite Komponente einen Schritt macht, während mit der Regel *ilvl blk lst* ein blockierter Schritt der ersten Komponente behandelt wird, wenn die zweite Komponente terminiert (beide Regeln entsprechen Fall *3.* bzw. *6.*). Die Regel *ilvl swtch*$_n$ wird benutzt, um die Priorisierung umzuschalten. Die letzten beiden Regeln *stp* und *lst* behandeln den Fall, in dem eine Komponente terminiert ist und das Restintervall durch die verbleibende Komponente bestimmt wird.

Die Intervalle $I \in (I_1 \mid\mid\!\lfloor_n I_2)$ werden entweder als unendliche Intervalle $I = (\sigma_0, \sigma_0', \sigma_1, \sigma_1', \dots)$ aus einer nichtterminierenden Sequenz

$$(I_1 \mid\mid\!\lfloor_n I_2) \xrightarrow{\sigma_0,\sigma_0'} (I_1' \mid\mid\!\lfloor_{n-1} I_2') \xrightarrow{\sigma_1,\sigma_1'} \dots$$

oder als endliche Intervalle $I = (\sigma_0, \sigma_0', \dots, \sigma_n)$ aus einer terminierenden Sequenz

$$(I_1 \mid\mid\!\lfloor_n I_2) \xrightarrow{\sigma_0,\sigma_0'} \dots \xrightarrow{\sigma_n} \emptyset\ .$$

gebildet.

2.3 Sequenzenkalkül und Rewriting

Die Beweismethode für ITL$^+$ basiert auf dem Sequenzenkalkül [Gen35, GTL89]. Regeln im Sequenzenkalkül haben die folgende Form:

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \dots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\ \mathit{name}\ .$$

Sie werden von unten nach oben angewandt. Die Regel *name* verfeinert eine gegebene Konklusion $\Gamma \vdash \Delta$ mit n Prämissen $\Gamma_n \vdash \Delta_n$. Dabei stehen Γ und Δ für (evtl. leere) Listen von Formeln. $\Gamma \vdash \Delta$ gilt genau dann, wenn die Konjunktion aller Formeln aus Γ die Disjunktion aller Formeln aus Δ impliziert. Zusätzlich werden im Folgenden häufig Umschreibungsregeln (engl. *rewrite rules*) benutzt, um Unterformeln umzuschreiben. Eine Umschreibungsregel wird durch

$$\mathit{name:} \quad \varphi \quad \leftrightarrow \quad \psi\ .$$

angegeben. Mit dieser Regel kann eine Formel φ durch eine äquivalente Formel ψ an jeder Stelle einer angegebenen Sequenz ausgetauscht werden.

2.4 Symbolische Ausführung

Die hier vorgestellte Beweisstrategie besteht darin, temporallogische Formeln symbolisch auszuführen. Parallele Programme werden als Spezialfall einer temporallogischen Formel angesehen und daher mit der gleichen Strategie symbolisch ausgeführt.

Die Grundidee dieser Beweisstrategie ist aus der *Dynamischen Logik* (DL) [Har84] übernommen. In DL wird eine Formel $\langle v := e; \alpha \rangle\ \varphi$ in zwei Schritten symbolisch ausgeführt. Zuerst wird die Formel normalisiert, das heißt der nächste Programmschritt – hier die führende Zuweisung – wird vom Rest des Programms getrennt. Dies wird mit der folgenden Regel bewerkstelligt:

$$\frac{\Gamma \vdash \langle v := e \rangle\ \langle \alpha \rangle\ \varphi}{\Gamma \vdash \langle v := e; \alpha \rangle\ \varphi}\ \textit{normalize}$$

Danach kann dieser Programmschritt symbolisch ausgeführt werden, wozu in DL die folgende Regel benutzt wird:

$$\frac{\Gamma_v^{v_0}, v = e_v^{v_0} \vdash \langle \alpha \rangle\ \varphi}{\Gamma \vdash \langle v := e \rangle\ \langle \alpha \rangle\ \varphi}\ \textit{asg } r$$

Diese Regel berechnet aus der Vorbedingung Γ die stärkste Nachbedingung $\Gamma_v^{v_0}, v = e_v^{v_0}$ der Zuweisung. Die Formel $e_v^{v_0}$ steht dabei für eine Umbenennung der Variable v zu v_0.

Die Beweisstrategie für den hier vorgestellten ITL$^+$-Kalkül ähnelt dem Vorgehen in DL. Das heißt, jede temporallogische Formel in einer Sequenz wird in einem ersten Schritt zu einer Normalform umgewandelt, die den nächsten Schritt von der restlichen Formel, die das System ab dem nächsten Zustand beschreibt, separiert. Danach wird ein Schritt für die gesamte Sequenz ausgeführt (siehe unten).

Somit wird eine temporallogische Formel (oder auch ein Programm) in die folgende Form umgeschrieben:

$$\tau \wedge \circ\, \varphi$$

Dabei ist die Transitionsformel τ eine prädikatenlogische Formel über ungestrichenen, gestrichenen und doppelt gestrichenen Variablen, die den nächsten System- und Umgebungsschritt beschreibt.

Allerdings muss dieses Schema für die Normalform noch erweitert werden, um die folgenden Sonderfälle zu berücksichtigen. Zum einen muss das Programm terminieren können. Das heißt, dass der aktuelle Zustand unter bestimmten Bedingungen der letzte Zustand sein kann. Diese Bedingung wird durch eine Formel τ_0 ausgedrückt (siehe unten). Außerdem kann der nächste Schritt nichtdeterministisch sein. Daher müssen verschiedene Transitionen τ_i mit jeweils entsprechenden Formeln φ_i berücksichtigt werden. Diese beschreiben die möglichen Transitionen und die jeweils dazugehörigen Nachfolgezustände. Schließlich kann es noch nötig sein, einen Zusammenhang zwischen einer Transition τ_i und dem Nachfolgezustand φ_i zu beschreiben, der nicht alleine durch die ungestrichenen, gestrichenen und doppelt gestrichenen Variablen ausgedrückt werden kann. Dies ist zum Beispiel der Fall, wenn etwa der erste Schritt nichtdeterministisch einen Wert wählt. Dieser Wert wird in der Semantik existenzquantifiziert. Siehe dazu z. B. die Semantik der **let**-Regel (Abschnitt 2.4.5). Um diesen Zusammenhang zu beschreiben, werden hier statische Variablen a_i benutzt, die sowohl in der Transition τ_i als auch in der Formel φ_i vorkommen. Damit ergibt sich das folgende allgemeine Schema, um den nächsten Schritt von

$$\frac{\varphi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\varphi \vee \psi, \Gamma \vdash \Delta}\ \mathit{dis\ l} \qquad \frac{\varphi_a^{a_0}, \Gamma \vdash \Delta}{\exists\ a.\ \varphi, \Gamma \vdash \Delta}\ \mathit{ex\ l}$$

mit neuer Variable a_0 bezüglich $(\mathrm{free}(\varphi) \setminus \{a\}) \cup \mathrm{free}(\Gamma, \Delta)$

$$\frac{\tau^{\underline{a},\underline{a},\underline{a}}_{\underline{A},\underline{A}',\underline{A}''} \vdash}{\tau, \mathbf{last} \vdash}\ \mathit{lst} \qquad \frac{\tau^{\underline{a_1},\underline{a_2},\underline{A}}_{\underline{A},\underline{A}',\underline{A}''},\ \varphi \vdash}{\tau, \circ\, \varphi \vdash}\ \mathit{stp}$$

mit $\{\underline{A}\} = \mathrm{free}(\tau, \varphi)$ und $\underline{a}, \underline{a_1}, \underline{a_2}$ enthalten zu $\underline{A}$ entsprechende neue statische Variablen bezüglich $\mathrm{free}(\tau, \varphi)$

Tabelle 2.2: Regeln zur Ausführung eines Schrittes

einer gegebenen temporallogischen Formel zu trennen:[3]

$$(\tau_0 \wedge \mathbf{last}) \vee \bigvee_{i=1}^{n} (\exists\ a_i.\ \tau_i \wedge \circ\, \varphi_i)\ .$$

Dieses allgemeine Schema wird im Folgenden als *Normalform* bezeichnet.

2.4.1 Ausführen eines Schrittes

Sobald eine Sequenz in Normalform ist, kann ein Schritt ausgeführt werden. Informell wird bei einem solchen symbolischen Ausführungsschritt der momentan betrachtete Zustand um einen Schritt im Intervall nach hinten verschoben. Dazu müssen alle Formeln im Antezedent in Normalform sein. Dies kann erreicht werden, indem jede einzelne Formel in die Normalform umgewandelt wird (die Regeln dafür werden in den folgenden Kapiteln dargestellt). Mehrere Formeln in Normalform können dann zu einer Formel in Normalform kombiniert werden. Da eine Formel im Sukzedent äquivalent zu ihrer negierten Formel im Antezedent ist, wird zur Vereinfachung im Folgenden angenommen, dass der Sukzedent leer ist. Die zur Ausführung eines Schrittes zu betrachtende Sequenz hat also die folgende Form:

$$(\tau_0 \wedge \mathbf{last}) \vee \bigvee_{i=1}^{n} (\exists\ a_i.\ \tau_i \wedge \circ\, \varphi_i) \vdash$$

Mit den beiden Regeln *dis l* und *ex l* aus Tabelle 2.2 können die Disjunktionen und die Quantoren eliminiert werden. Auf die verbleibenden Prämissen der Form

$$\tau_0 \wedge \mathbf{last} \vdash$$

und

$$\tau_i \wedge \circ\, \varphi_i \vdash$$

können die beiden Regeln *lst* und *stp* angewandt werden.

Im ersten Fall, in dem die Ausführung terminiert, werden alle freien dynamischen Variablen A – unabhängig davon, ob sie ungestrichen, gestrichen oder doppelt gestrichen vorkommen – durch neue statische Variablen a ersetzt. Dies

[3] Manche Normalformregeln erzeugen auch eine Formel der Form $\tau \wedge \bullet\, \varphi$. Diese können in die äquivalente Normalform $(\tau \wedge \mathbf{last}) \vee (\tau \wedge \circ\, \varphi)$ umgeformt werden.

ergibt eine rein prädikatenlogische Formel, die nur statische Variablen enthält. Diese kann mit den üblichen Regeln für Prädikatenlogik bewiesen werden.

Im anderen Fall wird der aktuelle Zustand um einen Schritt auf dem Intervall weiter verschoben. Im prädikatenlogischen Teil der Formel τ wird der Wert der dynamischen Variablen A und A' des alten Zustandes in neuen statischen Variablen a_1 und a_2 gespeichert (durch Zuweisungen $A = a_1$ bzw. $A' = a_2$). Die doppelt gestrichenen Variablen A'' des alten Zustandes werden zu ungestrichenen Variablen A des neuen Zustandes umbenannt. Zuletzt werden noch die führenden *Next*-Operatoren der temporallogischen Formel $\circ\,\varphi$ entfernt. In diesem Zustand kann nun die symbolische Ausführung mit der Formel φ fortgesetzt werden.

2.4.2 Ausführung temporallogischer Formeln

alw:	$\Box\,\varphi$	$\leftrightarrow$	$\varphi \wedge \bullet\,\Box\,\varphi$
ev:	$\Diamond\,\varphi$	$\leftrightarrow$	$\varphi \vee \circ\,\Diamond\,\varphi$
untl:	φ **until** ψ	$\leftrightarrow$	$\psi \vee (\varphi \wedge \circ\,(\varphi$ **until** $\psi))$
unls:	φ **unless** ψ	$\leftrightarrow$	$\psi \vee (\varphi \wedge \bullet\,(\varphi$ **unless** $\psi))$

Tabelle 2.3: Normalformregeln für temporallogische Operatoren

Das Prinzip der symbolischen Ausführung kann auf temporallogische Formeln angewandt werden. Zum Beispiel ähnelt der Operator $\Box\,\varphi$ einer Schleife in einer Programmiersprache, da die Formel φ in jedem Schritt ausgeführt wird. Die entsprechende Normalformregel *alw* ist in Tabelle 2.3 dargestellt. Diese Normalformregel entspricht auch der üblichen rekursiven Definition von $\Box\,\varphi$. Die Formel φ muss im aktuellen Zustand gelten und im nächsten Zustand muss wieder $\Box\,\varphi$ gelten. Falls φ weitere temporale Formeln enthält, muss diese Formel weiter in Normalform umgeschrieben werden, um die Gesamtformel in Normalform zu bringen. Für die anderen temporalen Operatoren existieren ähnliche Regeln zur Umwandlung in die Normalform.

2.4.3 Ausführung von Programmoperatoren

Sequentielle Programme können direkt in Normalform umgeschrieben werden. Die Umschreibungsregeln dafür werden in der Tabelle 2.4 angegeben. Für Zuweisungen und parallele Zuweisungen folgen die Umschreibungsregeln direkt aus deren Semantik.

Die Normalform von Fallunterscheidungen (d. h. **if**-Operatoren) und Schleifen kann ebenfalls einfach aus der Semantik abgeleitet werden. So ist zum Beispiel in der Formel

$$[\mathbf{if}^{*}\ \varphi\ \mathbf{then}\ \alpha_1\ \mathbf{else}\ \alpha_2]_{\underline{V}}$$

entweder φ wahr und α_1 wird ausgeführt oder φ ist falsch und α_2 wird weiter ausgeführt. Diese beiden Transitionen ergeben zusammen die folgende Normal-

asgp:	$[A_1 := e_1, \ldots, A_n := e_n]_{\underline{V}}$	$\leftrightarrow$	$A'_1 = e_1 \wedge \ldots \wedge A'_n = e_n$ $\wedge \bigwedge_{X \in \{\underline{V}\} \setminus \{A_1, \ldots, A_n\}} X' = X$ $\wedge \circ$ **last**
asgr:	$[A := ?]_{\underline{V}}$	$\leftrightarrow$	$\bigwedge_{X \in \{\underline{V}\} \setminus \{A\}} X' = X$ $\wedge \circ$ **last**
skip:	$[\textbf{skip}]_{\underline{V}}$	$\leftrightarrow$	$\bigwedge_{X \in \{\underline{V}\}} X' = X$ $\wedge \circ$ **last**
ite:*	$[\textbf{if}^* \ \psi \ \textbf{then} \ \varphi_1 \ \textbf{else} \ \varphi_2]_{\underline{V}}$	$\leftrightarrow$	$\psi \wedge [\varphi_1]_{\underline{V}}$ $\vee \neg \psi \wedge [\varphi_2]_{\underline{V}}$
ite:	$[\textbf{if} \ \psi \ \textbf{then} \ \varphi_1 \ \textbf{else} \ \varphi_2]_{\underline{V}}$	$\leftrightarrow$	$\psi \wedge [\textbf{skip}; \varphi_1]_{\underline{V}}$ $\vee \neg \psi \wedge [\textbf{skip}; \varphi_2]_{\underline{V}}$
star:	$[\alpha^*]_{\underline{V}}$	$\leftrightarrow$	**last** $\vee [((\, [\alpha]_{\underline{V}} \wedge \neg \, \textbf{last}\,); \alpha^*)]_{\underline{V}}$
whl:*	$[\textbf{while}^* \ \ \psi \ \textbf{do} \ \varphi]_{\underline{V}}$	$\leftrightarrow$	$[(\psi \wedge [\varphi]_{\underline{V}} \wedge \neg \, \textbf{last}); \ \textbf{while} \ \psi \ \textbf{do} \ \varphi]_{\underline{V}}$ $\vee \neg \psi \wedge \textbf{last}$
whl:	$[\textbf{while} \ \psi \ \textbf{do} \ \varphi]_{\underline{V}}$	$\leftrightarrow$	$[(\psi \wedge [\textbf{skip}; \varphi]_{\underline{V}} \wedge \neg \, \textbf{last}); \ \textbf{while} \ \psi \ \textbf{do} \ \varphi]_{\underline{V}}$ $\vee \neg \psi \wedge \textbf{skip}$
let:	$[\textbf{let} \ \underline{A} = \underline{e} \ \textbf{in} \ \alpha \ \in \varphi]_{\underline{V}}$	$\leftrightarrow$	$\exists \, \underline{A}. \quad \underline{A} = \underline{e} \ \wedge [\varphi]_{\underline{V},\underline{A}}$ $\wedge \, \Box \ \underline{A}'' = \underline{A}'$
choose:	$[\textbf{choose} \ \underline{A} \ \textbf{with} \ \psi$ $\textbf{in} \ \varphi_1$ $\textbf{ifnone} \ \varphi_2]_{\underline{V}}$	$\leftrightarrow$	$(\exists \, \underline{A}. \quad \psi \wedge [\varphi_1]_{\underline{V},\underline{A}}$ $\wedge \, \Box \ \underline{A}'' = \underline{A}')$ $\vee \, (\neg \, \exists \, \underline{A}. \, \psi) \wedge [\varphi_2]_{\underline{V}}$

Tabelle 2.4: Regeln zur Ausführung von sequentiellen Programmoperatoren

form:

$$\varphi \wedge [\alpha_1]_{\underline{V}} \vee \neg\, \varphi \wedge [\alpha_2]_{\underline{V}}$$

Natürlich müssen α_1 und α_2 ebenfalls in Normalform umgeschrieben werden.

Die Normalformregel für den *star*-Operator ist identisch mit der Eigenschaft „*ChopStarEqv*“ für den Sternoperator [CMZ02]. Die Umschreibungsregel der **while**- und **while***-Operatoren können von der Normalformregel des *star*-Operators abgeleitet werden.

Die Umschreibungsregeln für den **let**- und den **choose**-Operator sind von den quantifizierten dynamischen Variablen abgeleitet. Deren Semantik ist in Abschnitt 2.4.5 beschrieben.

2.4.4 Ausführung sequentieller Komposition

Die symbolische Ausführung der sequentiellen Komposition von Programmen ist komplizierter als bei den oben beschriebenen Operatoren, da in diesem Fall keine einfache Äquivalenz angegeben werden kann, die eine solche Formel in die Normalform umschreibt. Das Problem dabei ist, dass die Formel φ in der Formel $[\varphi;\, \psi]_{\underline{V}}$ eine unbekannte Anzahl von Schritten benötigt, bis sie terminiert und die Ausführung mit ψ fortgesetzt werden kann.

$$\frac{\vdash [\varphi_1]_{\underline{V}} \rightarrow [\varphi_2]_{\underline{V}}}{\vdash [\varphi_1;\, \psi]_{\underline{V}} \rightarrow [\varphi_2;\, \psi]_{\underline{V}}}\ \textit{chp lem}$$

chp dis:	$[(\varphi_1 \vee \varphi_2);\, \psi]_{\underline{V}}$	$\leftrightarrow$	$[\varphi_1;\, \psi]_{\underline{V}} \vee [\varphi_2;\, \psi]_{\underline{V}}$
chp ex:	$[(\exists\, a.\ \varphi);\, \psi]_{\underline{V}}$	$\leftrightarrow$	$\exists\, a.\ [\varphi;\, \psi]_{\underline{V}}$
	wobei o.B.d.A. $a \notin \mathrm{free}(\psi)$		
chp lst:	$[(\tau \wedge \mathbf{last}); \psi]_{\underline{V}}$	$\leftrightarrow$	$\tau^{A,A}_{A',A''} \wedge [\psi]_{\underline{V}}$
chp stp:	$[(\tau \wedge \circ\, \varphi); \psi]_{\underline{V}}$	$\leftrightarrow$	$\tau \wedge \circ\, [\varphi;\, \psi]_{\underline{V}}$

Tabelle 2.5: Normalformregeln für sequentielle Komposition

Die Regeln zur symbolischen Ausführung einer Formel $\varphi;\, \psi$ sind in der Tabelle 2.5 angegeben. Zuerst wird die Formel φ in Normalform umgeschrieben. Dies erlaubt die Regel *chp lem* aus Tabelle 2.5: Falls die Teilformel $[\varphi_1]_{\underline{V}}$ die Formel $[\varphi_2]_{\underline{V}}$ impliziert, so impliziert auch $[\varphi_1;\, \psi]_{\underline{V}}$ die Formel $[\varphi_2;\, \psi]_{\underline{V}}$. Diese Regel ist eine sogenannte Kongruenzregel. Nach Anwendung dieser Regel ergibt sich dann die neue Formel

$$\left[\left(\tau_0 \wedge \mathbf{last} \vee \bigvee\, (\exists\, a_i.\ \tau_i \wedge \circ\, \varphi_i)\right);\, \psi\right]_{\underline{V}}$$

Bevor nun der sequentielle Kompositionsoperator in Normalform umgeschrieben wird, muss diese Formel weiter vereinfacht werden. Dazu werden die Regeln *chp dis* und *chp ex* verwendet, die es erlauben den Kompositionsoperator über Disjunktion und Existenzquantoren zu distribuieren. Damit ergibt sich dann die folgende Formel:

$$[(\tau_0 \wedge \mathbf{last}); \psi]_{\underline{V}} \vee \bigvee\, \exists\, a_i.\ [(\tau_i \wedge \circ\, \varphi_i);\, \psi]_{\underline{V}}$$

Die erste Teilformel behandelt den Fall, in dem das Programm φ terminiert. In den anderen Fällen wird ein Schritt τ_i ausgeführt danach das Programm $\varphi_i; \psi$.

Nun kann der sequentielle Kompositionsoperator in den einzelnen Teilformeln in eine Normalform umgeschrieben werden. Dazu wird die Regel *chp lst* für die erste terminierende Teilformel und die Regel *chp stp* für die anderen Teilformeln benutzt. Da die dynamischen Variablen A im letzten Schritt stottern, werden die gestrichenen und doppelt gestrichenen Variablen A' und A'' in τ durch die entsprechenden ungestrichenen Variablen A ersetzt. Durch die Anwendung dieser beiden Regeln ergibt sich die folgende Formel:

$$\tau_0{}^{A,A}_{A',A''} \wedge [\psi]_{\underline{V}} \vee \bigvee \exists\, a_i.\ \tau_i \wedge \circ\, [\varphi_i; \psi]_{\underline{V}}$$

Im ersten Fall ist φ terminiert und die Teilformel $[\psi]_{\underline{V}}$ muss weiter umgeschrieben werden, um die Formel in Normalform zu bringen. In den anderen Fällen wurde eine Formel für den ersten Schritt τ_i erfolgreich von der restlichen Formel $[\varphi_i; \psi]_{\underline{V}}$ getrennt.

Beispiel 1 *(Sequentielle Programme).* Als Beispiel soll das Programm

$$[N := 0;\ N := 1]_{N,V},$$

in Normalform umgeschrieben und ein Schritt ausgeführt werden. Dazu wird in einem ersten Schritt die erste Zuweisung mit der Regel *asg* aus Tabelle 2.4 in Normalform umgeschrieben. Dies ergibt die folgende Formel:

$$[(N' = 0 \wedge \circ\, \mathbf{last} \wedge V' = V); N := 1]_{N,V}$$

Auf diese Formel kann nun die Regel *chp stp* aus Tabelle 2.5 angewendet werden. Damit wird der Kompositionsoperator unter den *Next*-Operator gezogen:

$$N' = 0 \wedge V' = V \wedge \circ\, [\mathbf{last}; N := 1]_{N,V}$$

Schließlich kann diese Formel mit der Äquivalenz $[\mathbf{last}; \varphi]_{N,V} \leftrightarrow [\varphi]_{N,V}$ zu

$$N' = 0 \wedge V' = V \wedge \circ\, [N := 1]_{N,V}$$

vereinfacht werden, was die obige Formel in Normalform darstellt.

Die symbolische Ausführung dieser Formel mittels der *stp*-Regel ergibt

$$n_2 = 0 \wedge v_2 = v_1 \wedge [N := 1]_{N,V}$$

□

2.4.5 Lokale Variablen und Quantoren

Formeln mit existenzquantifizierten dynamischen Variablen können ebenfalls symbolisch ausgeführt werden. Die Umschreibungsregeln hierzu sind in Tabelle 2.6 angegeben. Diese Regeln werden analog zu den Normalformregeln für sequentielle Komposition aus dem vorhergehenden Kapitel angewandt. Die

ex stp Regel ersetzt die ungestrichenen, gestrichenen und doppelt gestrichenen Variablen im momentanen Zustand durch neue statische Variablen a_0, a_1, a_2. Während die Variablen a_0 und a_1 nur im momentanen Zustand vorkommen, ist die Variable a_2 die Verbindung des momentanen Zustands mit dem nächsten Zustand. Daher muss diese Variable sowohl in der Transitionsformel τ als auch in der Formel φ, die das System ab dem nächsten Zustand beschreibt, quantifiziert werden.

Diese Strategie der symbolischen Ausführung erlaubt es, dass die Werte der Variable in allen Zuständen des Intervalls nicht am Anfang des Beweises festgelegt werden müssen (wie das zum Beispiel in TLA nötig ist), sondern sie erlaubt eine sukzessive Berechnung der quantifizierten Werte für jeden Schritt.

$$\frac{\vdash \forall\, X.\ (\varphi_1 \to \varphi_2)}{\vdash \exists\, X.\ \varphi_1 \to \exists\, X.\ \varphi_2}\ \textit{ex lem}$$

ex dis:	$\exists\, X.\ (\varphi_1 \vee \varphi_2)$	$\leftrightarrow$	$\exists\, X.\ \varphi_1 \vee \exists\, X.\ \varphi_2$
ex ex:	$\exists\, X.\ \exists\, a.\ \varphi$	$\leftrightarrow$	$\exists\, a.\ \exists\, X.\ \varphi$
ex lst:	$\exists\, X.\ \tau \wedge \mathbf{last}$	$\leftrightarrow$	$\exists\, a_0.\ \tau^{a_0,a_0,a_0}_{X,X',X''} \wedge \mathbf{last}$

wobei a_0 neu ist bezüglich free(τ)

ex stp:	$\exists\, X.\ \tau \wedge \circ\, \varphi$	$\leftrightarrow$	$\exists\, a_2.\ (\ \ (\exists\, a_0, a_1.\ \tau^{a_0,a_1,a_2}_{X,X',X''})$ $\wedge \circ\, \exists\, X.\ (X = a_2 \wedge \varphi)\ )$

wobei a_0, a_1, a_2 neu sind bezüglich free(τ, φ)

Tabelle 2.6: Normalformregeln für Existenzquantoren über dynamische Variablen

Die Normalformregeln für lokale Variablendeklarationen können aus den Normalformregeln für quantifizierte dynamische Variablen abgeleitet werden. Dazu wird die Äquivalenz (21) aus Abschnitt 2.2.4 benutzt.

2.4.6 Pfadquantoren

Bis auf den ersten Zustand werden Formeln mit einem führenden Pfadquantor über einem separaten Intervall ausgewertet. Daher werden diese Formeln nicht wie andere Formeln mit der in Abschnitt 2.4.1 beschriebenen *stp*-Regel symbolisch ausgeführt. Um die *stp*-Regel auf eine Sequenz mit einer pfadquantifizierten Formel anwenden zu können, müssen bei dieser zuerst alle freien dynamischen Variablen gebunden werden. Dies geschieht mit den in Tabelle 2.7 vorgestellten Regeln. Da eine Formel ohne freie dynamische Variablen invariant bezüglich des aktuellen Zustands ist (d. h., die Formel gilt entweder in allen oder in keinem Zustand eines Intervalls), werden die so umgeformten Formeln von einem symbolischen Ausführungsschritt nicht verändert. Formal wird dazu eine Formel τ_c, die keine freien dynamischen Variablen enthält, als Transitionsformel betrachtet. Durch die folgende Regel kann τ_c in Normalform gebracht werden:

$$\tau_c \ \leftrightarrow\ \tau_c \wedge \bullet\, \text{true}$$

pex clos: $\mathbf{E}\ \varphi \quad \leftrightarrow \quad \exists\ \underline{a}.\ \underline{X} = \underline{a} \wedge \mathbf{E}\ \exists\ \underline{X}.\ (\underline{X} = \underline{a} \wedge \varphi)$
wobei $\{\underline{X}\} = \text{free}(\varphi)$ und $\underline{a}$ enthält zu $\underline{X}$ entsprechende neue statische Variablen bezüglich $\text{free}(\varphi)$

pall clos: $\mathbf{A}\ \varphi \quad \leftrightarrow \quad \exists\ \underline{a}.\ \underline{X} = \underline{a} \wedge \mathbf{A}\ \exists\ \underline{X}.\ (\underline{X} = \underline{a} \wedge \varphi)$
wobei $\{\underline{X}\} = \text{free}(\varphi)$ und $\underline{a}$ enthält zu $\underline{X}$ entsprechende neue statische Variablen bezüglich $\text{free}(\varphi)$

Tabelle 2.7: Regeln zur Berechnung abgeschlossener Pfadquantorformeln

$$\frac{\vdash \varphi_1 \rightarrow \varphi_2}{\vdash \mathbf{E}\ \varphi_1 \rightarrow \mathbf{E}\ \varphi_2}\ \textit{pex lem}$$

pex dis: $\mathbf{E}\ (\varphi_1 \vee \varphi_2) \quad \leftrightarrow \quad \mathbf{E}\ \varphi_1 \vee \mathbf{E}\ \varphi_2$

pex ex: $\mathbf{E}\ \exists\ a.\ \varphi \quad \leftrightarrow \quad \exists\ a.\ \mathbf{E}\ \varphi$

pex lst: $\mathbf{E}\ (\tau \wedge \mathbf{last}) \quad \leftrightarrow \quad \tau^{\underline{X},\underline{X}}_{\underline{X}',\underline{X}''}$
wobei $\{\underline{X}\} = \text{free}(\tau)$

pex stp: $\mathbf{E}\ (\tau \wedge \circ\ \varphi) \quad \leftrightarrow \quad \exists\ \underline{a_2}.\ (\ \ (\exists\ \underline{a_1}.\ \tau^{\underline{a_1},\underline{a_2}}_{\underline{X}',\underline{X}''}) \wedge \mathbf{E}\ \exists\ \underline{X}.\ (\underline{X} = \underline{a_2} \wedge \varphi)\ )$
wobei $\{\underline{X}\} = \text{free}(\tau, \varphi)$ und $\underline{a_1}, \underline{a_2}$ enthalten zu $\underline{X}$ entsprechende neue statische Variablen bezüglich $\text{free}(\tau, \varphi)$

Tabelle 2.8: Ausführungsregeln für den Existiert-Pfadquantor

Damit bleibt τ_c von der *stp*-Regel unbeeinflusst, da sie keine freien dynamischen Variablen enthält.

Um zu zeigen, dass eine pfadquantifizierte Formel gilt, muss diese allerdings ebenfalls symbolisch ausgeführt werden. In dieser Arbeit wird nur die symbolische Ausführung von Existiert-Pfadquantoren beschrieben. Die entsprechenden Regeln für eine Formel $\mathbf{A}\ \varphi$ werden hier nicht benötigt, können aber mit den Regeln für $[\text{true}]\varphi$ hergeleitet werden [Bal05].

Die Regeln zur symbolischen Ausführung einer Formel $\mathbf{E}\ \psi$ sind in Tabelle 2.8 angegeben. Wie bei der normalen symbolischen Ausführung werden die Regeln *pex lem*, *pex dis* und *pex ex* benutzt, um die Formel ψ in die übliche Normalform zu bringen. Wenn diese Normalform von der Form $\tau \wedge \mathbf{last}$ ist, wird die Regel *pex lst* benutzt. Dabei wird gezeigt, dass τ in einem Endzustand gilt, d. h., dass alle dynamischen Variablen stottern dürfen. Falls die Normalform von der Form $\tau \wedge \circ\ \varphi$ ist, wird mit der Regel *pex stp* der Nachfolgezustand des quantifizierten Intervalls berechnet. Das Prinzip dieser Regel ist ähnlich wie bei der normalen symbolischen Ausführung. Die Transitionsformel τ wird aus der quantifizierten Formel „herausgezogen“. Dabei werden die gestrichenen und doppelt gestrichenen Variablen durch neue statische Variablen $\underline{a_1}$ und $\underline{a_2}$ ersetzt. Für die quantifizierte Formel muss nun noch gezeigt werden, dass es ein Intervall gibt, das mit $\underline{a_2}$ beginnt und die Formel φ erfüllt.

2.4.7 Interleaving

Wie bei der sequentiellen Komposition kann der Interleaving-Operator nicht direkt ausgeführt werden, sondern es ist zuerst nötig, die Teilformeln in Normalform umzuschreiben. Dabei ist es wichtig zu vermeiden, dass beide Formeln in Normalform gebracht werden müssen. Dies würde unnötige Fallunterscheidungen verursachen und die Sequenz schwer lesbar machen. Um dies zu erreichen, wird in einem ersten Schritt entschieden, welche der beiden parallelen Komponenten den nächsten Schritt ausführt. Dazu wird der Left-Merge-Operator $\varphi \mathbin{||\!\lfloor} \psi$ benutzt, der im nächsten Schritt die linke Komponente bevorzugt. Von der Intuition her entspricht dieser Operator dem Left-Merge-Operator von Bergstra [BPS01].[4] Die Semantik dieses Operators wird von der Semantik des $||\!\lfloor_n$ Operators (siehe Kapitel 2.2.6) abgeleitet:

$$\varphi \mathbin{||\!\lfloor} \psi \quad := \quad \bigcup_n \{I_1 \mathbin{||\!\lfloor_n} I_2 \mid I_1 \models \varphi \wedge I_2 \models \psi\}$$

Die Semantik von $||\!\lfloor_n$ wurde durch eine Menge von Intervallen definiert. Somit kann die Semantik von $||\!\lfloor$ als Vereinigung dieser Mengen über alle n definiert werden.

Die Entscheidung, welche der beiden Komponenten den nächsten Schritt ausführt, wird mit Hilfe dieses Operators und folgender Äquivalenz getroffen:

$$\varphi \parallel \psi \;\leftrightarrow\; (\varphi \mathbin{||\!\lfloor} \psi) \vee (\psi \mathbin{||\!\lfloor} \varphi)$$

Um die $||\!\lfloor$-Formel auszuführen, muss die linke Teilformel in Normalform gebracht werden. Danach kann der Left-Merge-Operator selbst ausgeführt werden.

$$\frac{\vdash \varphi_1 \to \varphi_2}{\vdash \varphi_1 \mathbin{||\!\lfloor} \psi \to \varphi_2 \mathbin{||\!\lfloor} \psi} \; \textit{ilvl lem}$$

ilvl dis:	$(\varphi_1 \vee \varphi_2) \mathbin{||\!\lfloor} \psi$	$\leftrightarrow$	$\varphi_1 \mathbin{||\!\lfloor} \psi \vee \varphi_2 \mathbin{||\!\lfloor} \psi$
ilvl ex:	$(\exists\, a.\, \varphi) \mathbin{||\!\lfloor} \psi$	$\leftrightarrow$	$\exists\, a_0.\, \varphi_a^{a_0} \mathbin{||\!\lfloor} \psi$
	wobei a_0 neu bezüglich $(\mathrm{free}(\varphi) \setminus \{a\}) \cup \mathrm{free}(\psi)$		
ilvl lst:	$(\tau \wedge \mathbf{last}) \mathbin{||\!\lfloor} \psi$	$\leftrightarrow$	$\tau^{A,A}_{A',A''} \wedge \psi$
ilvl stp:	$(\tau \wedge \neg\, \mathbf{blocked} \wedge \circ\, \varphi) \mathbin{||\!\lfloor} \psi$		
	$\leftrightarrow \exists\, a_2.\, (\tau^{a_2}_{A''} \wedge \neg\, \mathbf{blocked} \wedge \circ\, ((A = a_2 \wedge \varphi) \parallel \psi))$		
ilvl blk:	$(\tau \wedge \mathbf{blocked} \wedge \circ\, \varphi) \mathbin{||\!\lfloor} \psi$		
	$\leftrightarrow \exists\, a_2.\, (\exists\, a_1.\, \tau^{a_1,a_2}_{A',A''}) \wedge (A = a_2 \wedge \varphi) \mathbin{||\!\lfloor^b} \psi$		

Tabelle 2.9: Normalformregel für den Left-Merge-Operator

Die Normalformregeln für den Left-Merge-Operator sind in Tabelle 2.9 angegeben. Sie sind ähnlich aufgebaut wie die Normalformregeln der sequentiellen

[4] Ich danke A. Knapp für den hilfreichen Hinweis auf diesen Zusammenhang.

Komposition. Um die linke Teilformel in Normalform umzuschreiben, wird die Kongruenzregel *ilvl lem* benutzt. Genau wie bei der sequentiellen Komposition distribuiert der Left-Merge-Operator über Disjunktion (mit der Regel *ilvl dis*) und über existenzquantifizierte Formeln (mit der Regel *ilvl lst*).

Falls die linke Komponente terminiert, wird die Ausführung mit der rechten Komponente fortgesetzt (Regel *ilvl lst*), ähnlich wie bei der Regel *chp stp* für die sequentiellen Komposition. Falls die linke Komponente noch nicht terminiert, hängt die weitere Ausführung davon ab, ob diese blockiert ist oder nicht.

Falls sie nicht blockiert ist, wird der nächste Schritt durch die Regel *ilvl stp* ausgeführt und danach wird das Interleaving der verbleibenden Komponenten φ und ψ fortgesetzt. Dabei ist zu beachten, dass die zweifach gestrichenen Variablen in der Schrittformel τ durch neue statische Variablen a_2 ersetzt werden, die beim nächsten Schritt gleich den ungestrichenen Variablen der linken Komponente sind. Dadurch wird erreicht, dass der Umgebungsschritt der linken Komponente die globalen Umgebungsschritte (d. h. die Umgebungsschritte aus Sicht des Gesamtsystems) als auch die Schritte der anderen Komponente enthält (siehe dazu auch den Abschnitt 2.2.6 bzw. die Abbildung 2.2).

Sofern die linke Komponente blockiert ist, wird durch die Regel *ilvl blk* ein *blockierter Schritt* ausgeführt: Die Komponente wartet aktiv, während sie blockiert ist (d. h., sie führt einen blockierten Schritt aus). Allerdings werden die gestrichenen Variablen von τ durch neue statische Variablen a_1 ersetzt, da die blockierte Transition nichts zur Transition des gesamten parallelen Systems beiträgt. Zusätzlich wird ein Schritt der anderen Komponente ausgeführt.

$$\frac{\vdash \varphi_1 \to \varphi_2}{\vdash \psi \mathbin{||\!\lfloor^b} \varphi_1 \to \psi \mathbin{||\!\lfloor^b} \varphi_2} \; \textit{ilvlb lem}$$

ilvlb dis:	$\psi \mathbin{||\!\lfloor^b} (\varphi_1 \vee \varphi_2)$	$\leftrightarrow$	$\psi \mathbin{||\!\lfloor^b} \varphi_1 \vee \psi \mathbin{||\!\lfloor^b} \varphi_2$
ilvlb ex:	$\psi \mathbin{||\!\lfloor^b} (\exists\, a.\, \varphi)$	$\leftrightarrow$	$\exists\, a_0.\, \psi \mathbin{||\!\lfloor^b} \varphi_a^{a_0}$
	a_0 neu bezüglich $(\mathrm{free}(\varphi) \setminus \{a\}) \cup \mathrm{free}(\psi)$		
ilvlb lst:	$\psi \mathbin{||\!\lfloor^b} (\tau \wedge \mathbf{last})$	$\leftrightarrow$	false
ilvlb stp:	$\psi \mathbin{||\!\lfloor^b} (\tau \wedge \circ\, \varphi)$	$\leftrightarrow$	$\exists\, a_2.\, \tau_{A''}^{a_2} \wedge \circ\, (\psi \parallel (A = a_2 \wedge \varphi))$

Tabelle 2.10: Normalformregeln für den Left-Merge-Operator im blockierten Fall

In dieser Situation, in der der linke Prozess blockiert ist und der rechte Prozess einen Schritt ausführt, wird der abgeleitete Operator $\varphi \mathbin{||\!\lfloor^b} \psi$ benutzt:

$$\varphi \mathbin{||\!\lfloor^b} \psi \;\; :\equiv \;\; (\mathbf{blocked} \wedge \circ\, \varphi) \mathbin{||\!\lfloor} \psi$$

Auf diesen abgeleiteten Operator sind die Normalformregeln aus der Tabelle 2.10 anwendbar. Sie sind fast identisch zu den oben beschriebenen Normalformregeln für den Left-Merge-Operator. Die Kongruenzregel *ilvlb lem* stellt sicher, dass die rechte Teilformel in eine Normalform umgeschrieben werden kann. Die Regeln *ilvlb dis* und *ilvlb ex* sorgen dafür, dass der Operator über

Disjunktionen und existenzquantifizierte Formeln distribuiert. Falls der zweite Prozess terminiert, wird die Regel *ilvlb lst* angewandt, ansonsten wird die Regel *ilvlb stp* benutzt.

2.5 Induktion und Beweislemmas

Induktion wird in ITL$^+$ benutzt, um Eigenschaften für Programme mit Schleifen zu beweisen. Beweislemmas dienen dazu, um die Anzahl der zu betrachtenden Abläufe in parallelen Systemen zu reduzieren.

2.5.1 Induktion

Bei einem Beweis mit symbolischer Ausführung von nichtterminierenden Systemen, wie zum Beispiel nichtterminierende Schleifen, wird Induktion benötigt, um den Beweis zu schließen. Sei $<$ eine wohlfundierte Ordnung und e eine Term. Eine allgemeine Regel für Induktion ist

$$\frac{e = a, \bullet \,\Box\, (e < a \rightarrow \mathbf{ih}), \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ \mathit{ind}(e)$$

wobei **ih** durch

$$\mathbf{ih} :\equiv \mathbf{A}\bullet\, (\bigwedge \Gamma \rightarrow \bigvee \Delta)$$

definiert ist und a eine neue statische Variable bezüglich der freien Variablen $\mathrm{free}(e, \Gamma, \Delta)$. Die Induktionshypothese muss mit dem Allpfadquantor quantifiziert werden, damit sie unabhängig vom aktuellen Ablauf ist und sich z. B. nicht mit einem Step ändert. Die Regel *ind* kann auf jede Beweisverpflichtung $\Gamma \vdash \Delta$ angewendet werden, um eine Induktionshypothese als zusätzliche Vorbedingung zu generieren. Ab dem nächsten Zustand kann diese Induktionshypothese angewandt werden, falls der Term e bezügliche der Ordnung $<$ abgenommen hat. Der ursprüngliche Wert von e wird dabei in der neuen statischen Variable a gespeichert.

Es ist jedoch in einigen Fällen möglich, einen Induktionsterm automatisch aus bestimmten Formeln zu generieren. Falls eine Lebendigkeitseigenschaft $\Diamond\, \varphi$ gilt, ist es möglich, über die Anzahl der Schritte bis zu dem Zustand, in dem φ wahr wird, zu induzieren. Aus dieser Grundidee ist die folgende Regel abgeleitet:

$$\frac{(\bullet\ \mathbf{ih})\ \mathbf{until}\ \varphi, \Gamma \vdash \Delta}{\Diamond\, \varphi, \Gamma \vdash \Delta}\ \mathit{ind\ ev}$$

Die Korrektheit dieser Regel kann gezeigt werden, indem man die Äquivalenz

$$\Diamond\, \varphi \ \leftrightarrow\ \exists N.(N = N'' + 1\ \mathbf{until}\ \varphi)$$

benutzt, um die Regel *ind ev* auf Induktion über die Variable N zurückzuführen. Dabei ist $N = N'' + 1$ äquivalent zu $(N > 0 \wedge N'' = N - 1)$, d. h., solange N größer 0 ist, wird N in jedem Schritt um eins verkleinert.

Die Regel *ind ev* ist nicht nur dann anwendbar, falls eine Formel $\Diamond\ \varphi$ im Antezedent vorkommt. Vielmehr ist es möglich eine solche Lebendigkeitseigenschaft aus anderen temporallogischen Formeln zu generieren. So kann zum Beispiel die Äquivalenz

$$\Box\ \varphi \leftrightarrow \neg\ \Diamond\ \neg\ \varphi\ . \tag{2.1}$$

benutzt werden, um eine Sicherheitsformel im Sukzedenten in die benötigte Lebendigkeitseigenschaft umzuwandeln. Für einen **until**-Operator im Sukzedent kann die Äquivalenz

$$\varphi\ \textbf{until}\ \psi \leftrightarrow \forall\ B.\ (\Diamond\ B) \rightarrow (\varphi\ \textbf{unless}\ (\varphi \wedge B \vee \psi))$$

benutzt werden, um die benötigte Lebendigkeitseigenschaft zu generieren. Die Formel φ **until** ψ ist genau dann wahr, wenn sie auf jedem Präfix eines Intervalls erfüllt ist. Dabei wird das Ende des betrachteten Ablaufs durch B markiert. Für andere temporallogische Formeln wie zum Beispiel einem **until**-Operator im Antezedent gibt es ähnliche Regeln, um aus ihnen die für die Induktion benötigte Lebendigkeitseigenschaft zu extrahieren.

Die hier vorgestellte Strategie der symbolischen Ausführung stellt sicher, dass der Ablauf entweder nach einer endlichen Anzahl von Schritten terminiert oder ein Zustand erreicht wird, in dem Induktion angewandt werden kann. Allerdings ist es in manchen Fällen nötig, die prädikatenlogischen Formeln zu einer Invariante zu generalisieren, damit Induktion angewandt werden kann. Ein Vorteil der hier vorgestellten Beweisstrategie ist allerdings, dass nur der prädikatenlogische Teil einer Formel generalisiert werden muss. Außerdem wird mit dem hier vorgestellten Induktionsprinzip zum Beispiel für den Beweis einer Schleife nur eine Hoare-artige Invariante benötigt, die jeweils nur zu Beginn und am Ende der Schleifenausführung gelten muss. Eine Invariante, die in jedem Schritt innerhalb des Schleifenrumpfes gilt, ist nicht erforderlich.

2.5.2 Beweislemmas

Die Ausführung von interleaved-parallelen Programmen ist nichtdeterministisch. Daher kann die Ausführung eines interleaved-parallelen Systems zu einer großen Anzahl unterschiedlicher Abläufe führen, die beim Beweisen alle behandelt werden müssen. Dabei können unterschiedliche Abläufe häufig zum gleichen Zustand führen. Beispielsweise können zwei Transitionen, die in unterschiedlicher Reihenfolge ausgeführt werden, sich dabei aber gegenseitig nicht beeinflussen, zum gleichen Zustand führen.

Um zu verhindern, dass in einem solchen Fall gleichartige Beweisteile für jeden dieser Zustände mehrfach geführt werden müssen, werden normalerweise Lemmas benutzt, die einmal bewiesen werden und dann für jeden passenden Zustand anwendbar sind. Bei dieser Strategie muss sich der Benutzer allerdings schon während des Beweises im Klaren sein, welche Zustände mehrfach auftreten können. Außerdem kann diese Strategie bei großen Beweisen sehr unübersichtlich werden, da in manchen Fällen viele sehr ähnliche Lemmas zu verwalten sind.

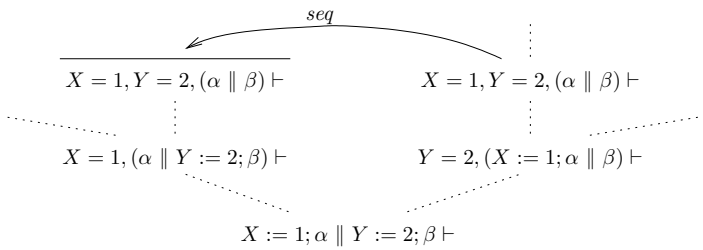


Abbildung 2.3: Beispiel für die Anwendung der *seq*-Regel

Um diese Probleme zu vermeiden, wurde mit Beweislemmas die Möglichkeit geschaffen, Knoten aus dem aktuellen Beweisbaum als Lemmas für den aktuellen Beweis anzunehmen. Dabei dürfen natürlich keine Zyklen auftreten, d. h., es darf weder einer der Vorgängerknoten im aktuellen Beweisast als Beweislemma angenommen werden, noch darf ein Knoten aus einem Ast benutzt werden, der bereits ein Beweislemma aus einem solchen Vorgängerknoten benutzt.

Ein Beispiel für die Anwendung der Beweislemma-Regel ist in Abbildung 2.3 dargestellt. In diesem Beispiel wird nach zwei Schritten in zwei verschiedenen Abläufen die gleiche Systemkonfiguration erreicht. Eine der beiden Prämissen kann durch Verwendung der anderen Prämisse als Beweislemma geschlossen werden.

2.6 Beispiel

Als Beispiel wird gezeigt, dass in der Formel

$$\begin{aligned} \Gamma \ :\equiv \quad & [\textbf{while } N < m \textbf{ do } N := N + 1]_N \\ & \wedge \ \Box \ N' = N'' \end{aligned}$$

die Variable N niemals einen größeren Wert als den der statischen Variable m annimmt. Formal wird also gezeigt, dass

$$N \leq m, \Gamma \vdash \Box \ N \leq m$$

gilt. Mit Hilfe der Äquivalenz (2.1) kann diese Formel zu

$$N \leq m, \Gamma, \Diamond \ N > m \ \vdash$$

umgeschrieben werden. Die prädikatenlogische Formel $N \leq m$ ist bereits eine passende Invariante, daher ist keine weitere Generalisierung nötig.

Bei der Anwendung der Regel *ind ev* wird eine Induktionshypothese generiert. Die neue Sequenz hat dann folgende Form:

$$N \leq m, \Gamma, (\bullet \ \textbf{ih}) \ \textbf{until} \ N > m \vdash \tag{2.2}$$

mit

$$\textbf{ih} :\equiv (N \leq m \wedge \Gamma) \rightarrow \text{false}$$

Die **until**-Formel in der Sequenz (2.2) kann mit den Regeln aus Tabelle 2.3 in Normalform gebracht werden. Da der rechte Teil der **until**-Formel $N > m$ in dieser Sequenz nicht gilt, hat die neue Sequenz die folgende Form:

$$N \leq m, \Gamma, \bullet\ \mathbf{ih}, \circ\ ((\bullet\ \mathbf{ih})\ \mathbf{until}\ N > m) \vdash \tag{2.3}$$

Auf die Formel $N \leq m$ kann nun eine Fallunterscheidung, ob $N = m$ oder $N < m$ gilt, angewandt werden.

Im Fall $N = 0$ kann die **while**-Schleife in der Formel Γ zu **last** umgeschrieben werden (mit der Regel *whl* aus Tabelle 2.4 und da $N = 0$ gilt). Dies widerspricht allerdings der Formel $\circ$ **ih until** $N > m$ aus der Sequenz (2.3), die fordert, dass es einen nächsten Zustand geben muss.

Im anderen Fall gilt $N < m$. Hier können die anderen Formeln in der Sequenz (2.3) in Normalform umgeschrieben werden. Nach der Ausführung eines Schrittes mit der *stp*-Regel aus Abschnitt 2.4.1 erhält man das folgende neue Beweisziel:

$$N \leq m, \Gamma, \mathbf{ih}, (\bullet\ \mathbf{ih})\ \mathbf{until}\ N > m \vdash$$

Dieses Beweisziel kann sofort mit **ih** geschlossen werden.

2.7 Andere Ansätze

Andere interaktive Ansätze wie z. B. der STeP-Beweiser [BBC+00], TLPVS [PA03] oder +Cal [Lam06a] benutzen ähnliche Spezifikationssprachen (obwohl die Semantik etwas verschieden ist, da keine dieser Sprachen doppelt gestrichene Variablen verwendet, und einige auf einer „Stotter"-Semantik basieren). Damit wird die Beschreibung eines Systems vor der Verifikation in ein passendes Transitions-System übersetzt (d. h. in die Normalform von TLA [Lam94] überführt). In der vorliegenden Arbeit wird dagegen die ursprüngliche Systembeschreibung beibehalten und die Transitionen mittels symbolischer Ausführung berechnet.

Der hier vorgestellte Ansatz unterstützt temporale Logik direkt, ähnlich wie der SteP Beweiser. Eine Alternative für die direkte Verwendung temporaler Logik bietet eine Codierung in einer Logik höherer Ordnung, wie z.B. TLPVS [PA03], Codierung von TLA in Isabelle [Mer95, Kal95] oder die Codierung von Rely-Guarantee Zusicherungen in Isabelle [Pre03]). Vorteilhaft ist dabei, dass die Korrektheit der temporalen Logik auf die der höheren Ordnung zurückgeführt werden kann. Allerdings erfordert dies auch die Codierung einer beträchtlichen Menge semantischer Ausdrücke. Es ist auch keine Codierung bekannt, die sämtliche Eigenschaften von ITL$^+$ unterstützt: Lokale Variablen, Rekursion, Blockierung und faires Interleaving.

2.8 Unterschiede zu früheren Versionen von ITL$^+$

Die Logik ITL$^+$ und der Kalkül für symbolische Ausführung wurde von Michael Balser entworfen und in seiner Dissertation [Bal05] beschrieben. Die hier

vorgestellte Logik und der Kalkül entsprechen im wesentlichen dieser Logik. Allerdings wurden einige Teile von ITL$^+$ im Rahmen dieser Arbeit an die hier verwendeten Beweistechniken angepasst bzw. verbessert. Diese Änderungen umfassen im wesentlichen die folgenden Punkte:

- Die Logik berücksichtigt nun auch Typen höherer Ordnung. Diese können sowohl von statischen als auch von dynamischen Variablen benutzt werden.

- Die Frame-Assumption wurde ersetzt. In [Bal05] hatten z. B. Programmzuweisungen keine freien Variablen, da sie explizit *alle* Variablen unverändert gelassen haben. Deshalb war es nicht möglich, neue Variablen für Allquantoren im Sukzedent bzw. Existenzquantoren im Antezedent zu finden. In der hier vorgestellten Logik wird für jedes Programm syntaktisch eine Menge von Variablen angegeben, die von diesem Programm beeinflusst werden, wie dies z. B. auch in TLA gehandhabt wird. Dadurch können quantifizierte Variablen wie im Sequenzenkalkül instanziert werden.

- In [Bal05] wurden drei Sorten von Variablen verwendet: Statische, dynamische und Programm-Variablen. In dieser Arbeit wurden die dynamischen und die Programm-Variablen vereinheitlicht, so dass in der Logik nur noch die Unterscheidung zwischen statischen und dynamischen Variablen existiert. Die **let**-Regel kann dadurch deutlich vereinfacht werden. Außerdem entspricht die Unterscheidung zwischen statischen und dynamischen Variablen der KIV-Implementierung.

- Die Quantoren für statische und dynamische Variablen wurden vereinheitlicht. Somit gibt es nur noch eine Art von Quantoren, mit denen sowohl statische als auch dynamische Variablen quantifiziert werden können.

- Die Systemoperatoren $\langle\varphi\rangle\psi$ wurde durch die Pfadquantoren $\mathbf{E}\varphi$ ersetzt. Dabei gilt:
$$\langle\varphi\rangle\psi \;\Leftrightarrow\; \langle\text{true}\rangle(\varphi \to \psi) \;\Leftrightarrow\; \mathbf{E}(\varphi \to \psi)$$

- Die Sequencing-Regel aus [Bal05] wurde durch Beweislemmas ersetzt. Beide Techniken sind funktional identisch, jedoch ist die Theorie für Beweislemmas deutlich einfacher.

- Die hier vorgestellte Logik stimmt mit der KIV-Implementierung überein. In [Bal05] gab es einige Unterschiede zwischen der Theorie und der KIV-Implementierung. Diese Unterschiede wurden korrigiert.

Da viele dieser Änderungen Auswirkungen auf große Teile der Logik haben, wurde aus Gründen der Übersichtlichkeit und Vollständigkeit die gesamte Logik in dieser Arbeit vorgestellt.

Kapitel 3

Modulares Beweisen

Im Folgenden werden grundlegenden Konzepte modularer Beweistechnik beschrieben. Insbesondere wird dabei die *Rely-Guarantee*-Methode erläutert, welche die Ausgangsbasis für die in den folgenden Kapiteln vorgestellten Techniken bildet.

3.1 Beispiel: Semaphor-Algorithmus

Um modulare Techniken zu erläutern, wird in diesem und im folgenden Kapitel der klassische Semaphor-Algorithmus, der 1965 von Edsger V. Dijkstra vorgestellt wurde [Dij02], als Beispiel verwendet. Dieser klassische parallele Algorithmus zur Prozesssynchronisation verhindert, dass mehrere Prozesse gleichzeitig auf eine kritische Ressource zugreifen können.

Der hier als Beispiel benutzte Semaphor-Algorithmus ist in Abbildung 3.1 dargestellt. Jede Komponente S hat drei Parameter: Die boolesche Semaphor *Sem*, einen *Program Counter PC*, der die Werte *critical* und *noncritical* annehmen kann und schließlich *State*, welches alle anderen Variablen repräsentiert, auf die S zugreifen kann. Die Variable PC wird benutzt, um festzustellen, ob sich ein Prozess im kritischen Abschnitt befindet, in dem auf die Ressource zugegriffen wird. In Zeile 3 wartet der Algorithmus, bis die Semaphore frei ist (d.h. Sem = true). Sobald dies eintritt, wird die Semaphore belegt (d.h. Sem = false) und der Programmzähler auf *critical* gesetzt (Zeile 4). Der Programmablauf in der kritischen Sektion wird durch den Prozeduraufruf von *Crit*

```
1  S(; PC, Sem, State){
2  while true do{
3      await Sem = true;
4      Sem := false, PC := critical;
5      Crit(State);
6      Sem := true, PC := noncritical;
7      Noncrit(State);
8  }
```

Abbildung 3.1: Spezifikation des Semaphor-Algorithmus

repräsentiert (Zeile 5). *Crit* bekommt als Parameter nur *State*, da hier weder der Programmzähler *PC* noch die Semaphor *Sem* verändert werden dürfen. Beim Verlassen des kritischen Abschnitts in Zeile 6 wird die Semaphor wieder freigegeben und der Programmzähler wieder auf *noncritical* gesetzt. Der weitere Ablauf in der nichtkritischen Sektion wird durch den Prozeduraufruf von *Noncrit* repräsentiert, der ebenfalls als Parameter nur die Variable *State* besitzt.

Als Korrektheitseigenschaft für diesen Algorithmus ist zu zeigen, dass zwei *Semaphor*-Prozesse niemals gemeinsam im kritischen Abschnitt sind. Mit dem in Kapitel 2 vorgestellten Kalkül wird diese Aussage durch folgende Formel ausgedrückt:

$$\begin{aligned} & S(PC_1, Sem, State) \ \| \ S(PC_2, Sem, State), \\ & PC_1 = noncritical, PC_2 = noncritical, \\ & Sem = \text{true} \\ \vdash \ & \Box \neg \ (PC_1 = critical \wedge PC_2 = critical) \end{aligned} \tag{3.1}$$

Im Anfangszustand sind beide Komponenten in ihrem nichtkritischen Bereich und die Semaphor *Sem* ist nicht belegt. Während des Systemablaufs mit zwei S-Komponenten dürfen sich beide Prozesse niemals zum gleichen Zeitpunkt im kritischen Bereich aufhalten. Somit soll gezeigt werden, dass PC_1 und PC_2 niemals gleichzeitig *critical* sind.

3.2 Motivation: Komplexität von direkten Beweisen mit parallelen Programmen

Die Sequenz (3.1) im vorhergehenden Abschnitt kann direkt bewiesen werden, indem man den Interleaving-Operator mit beiden Komponenten direkt symbolisch ausführt. Solange die abstrakten Prozeduraufrufe *Crit* und *Noncrit* sehr einfach sind, z. B. nur aus einem **skip**-Schritt bestehen, ist dieser Beweis auch nicht schwierig, wird aber schon unerwartet groß. Jede Komponente benötigt in diesem Fall mindestens fünf Schritte, bevor sie wieder am Schleifenanfang ankommt und durchläuft somit auch fünf verschiedene Zustände, die den Zeilen 2, 3, 5, 6 und 7 in Abbildung 3.1 entsprechen. Jeder Schritt der Komponenten selbst ist zwar deterministisch, allerdings führt das Interleaving in jedem Schritt zu zwei verschiedenen Möglichkeiten. Allgemein kann ein paralleles System bis zu

$$\textit{(Zustände pro Komponente)}^{\textit{(Anzahl der Komponenten)}}$$

mögliche Zustände annehmen, also theoretisch $5^2 = 25$ verschiedene Zustände beim Semaphor-Algorithmus. Zwar sind bei einem direkten Beweis vier dieser Zustände nicht erreichbar, da sich niemals beide Prozesse zugleich im kritischen Abschnitt befinden. Andererseits müssen aber auch einige Zustände ein zweites Mal betreten werden, um etwa Induktion oder Beweislemmas anzuwenden. Insgesamt müssen somit bei einem direkten Beweis des vereinfachten Semaphor

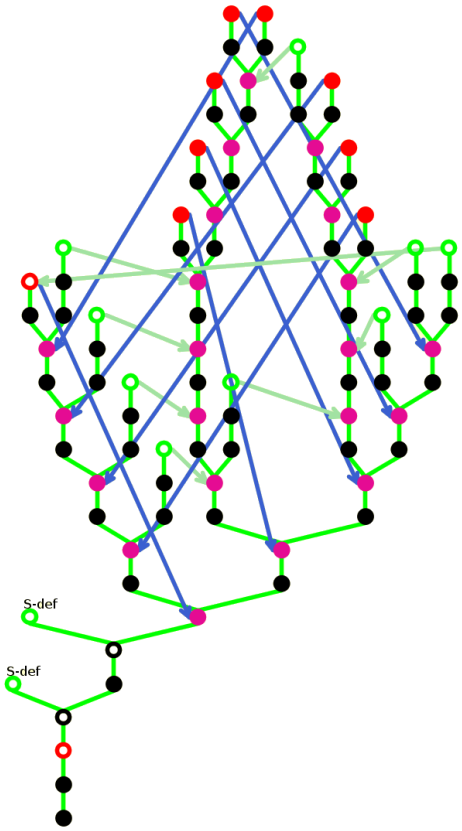


Abbildung 3.2: Beweisbaum für direkten Beweis des Semaphor Algorithmus

Algorithmus 39 Zustandsknoten betrachtet werden.[1]

Der Beweis wird noch aufwändiger, wenn die beiden Prozeduren *Noncrit* und *Crit* komplexer oder sogar komplett unspezifiziert sind. Auch in diesen Fällen müssen alle möglichen Interleavings berücksichtigt werden. Aber bei abstrakten Programmen hat jeder Interleavingschritt bei nur zwei Komponenten *immer* sechs verschiedene Möglichkeiten diesen Schritt auszuführen (diese sechs möglichen Schritte sind in Kapitel 2.2.6 beschrieben). Jeder Schritt mit zwei interleaved, unspezifizierten Operationen führt also zu sechs verschiedenen Nachfolgezuständen, von denen jeder wiederum bei weiterer symbolischer Ausführung mehrere Nachfolgezustände produziert. Mit zwei Komponenten, bei denen jede Komponenten sechs Zustände einnehmen kann, führt das symbolische Ausführen theoretisch zu $5^6 = 15625$ verschiedenen Zustandsknoten! Zwar wird diese theoretische Zahl in der Praxis normalerweise nicht erreicht, da ja nicht alle Schritte unspezifizierte Operation enthalten und die Zahl der erreichten Zustände durch Anwendung der Beweislemma-Regel weiter reduziert

[1] Mit Zustandsknoten werden die Knoten im Beweisbaum bezeichnet, die direkt aus einem symbolischen Ausführungsschritt folgen. Knoten die aus Simplifikation, prädikatenlogischen oder sonstigen temporallogischen Regeln (wie z. B. Generierung der Induktionshypothese oder Normalformberechnung) entstehen, werden nicht als eigener Zustansknoten gezählt.

werden kann. Die Erfahrung zeigt aber, dass schon Beweise von sehr einfachen Programmen so häufig zu einem nicht mehr akzeptablen Beweisaufwand führen.

Deutlich gravierender wird dieses Problem, wenn mehr als zwei nebenläufige Komponenten zu betrachten sind. Die Zahl der möglichen Zustände nimmt exponentiell mit der Zahl der parallelen Komponenten zu. Daher sind solche Fälle üblicherweise durch direkte symbolische Ausführung nicht mehr interaktiv zu beweisen.

3.3 Der Rely-Guarantee Ansatz

Die Grundidee des modularen Ansatzes ist, den Beweis einer Eigenschaft vollständig auf Beweise von lokalen Eigenschaften über die einzelnen Komponenten zu reduzieren. Paralleloperatoren, die ja — wie oben schon beschrieben — einen großen Teil des Beweisaufwandes ausmachen können, werden vollständig aus dem eigentlichen Beweis eliminiert.

Natürlich folgt die Eigenschaft für das Gesamtsystem nicht automatisch aus den gezeigten Komponenteneigenschaften. Um dies zu erreichen, wird ein so genanntes Modularisierungstheorem benutzt. Dieses gibt Bedingungen vor, welche die lokalen Eigenschaften erfüllen müssen, damit die Gesamteigenschaft gültig ist. Die Hauptschwierigkeit besteht darin, diese Beweisverpflichtungen zu finden und daraus ein entsprechendes Modularisierungstheorem zu entwickeln.

Das Hauptproblem dabei ist, dass sich die Komponenten in einem parallelen System gegenseitig in ihrem Ablauf beeinflussen. Daher erfüllt eine Komponente lokale Eigenschaften, die ohne Berücksichtigung der Umgebungsprozesse gezeigt wurden, eventuell nicht mehr, sobald diese Komponente Teil eines parallelen Systems ist.

Beispiel 2 *(Semaphor – Fortsetzung).*
Beim Semaphor-Beispiel erfüllt bereits die erste Komponente die Gesamteigenschaft wenn die zweite Komponente nicht berücksichtigt wird. Da die erste Komponente die Variable PC_2 nicht ändert, bleibt diese immer im Bereich *noncritical* und die Gesamteigenschaft ist somit trivialerweise erfüllt. Dieser Ablauf entspricht aber nicht den Abläufen, die aus Sicht der ersten Komponente in dem Gesamtsystem möglich wären, da sich hier ja PC_2 ändern kann. □

Daher ist es notwendig, dass die lokalen Eigenschaften der Komponenten so formuliert werden können, dass das Verhalten der Umgebung[2] berücksichtigt wird. Dabei ist nicht jedes Verhalten der Umgebung bedeutsam, sondern nur das für die lokale Eigenschaft relevante. Das Verhalten der Umgebung kann also abstrahiert werden.

Üblicherweise müssen die lokalen Eigenschaften und das (abstrahierte) Verhalten der Umgebung interaktiv spezifiziert werden. Daher ist es bei jeder modularen Technik wichtig, dass das Zusammenspiel zwischen der lokalen Eigenschaft und dem Verhalten der Umgebung möglichst intuitiv formuliert werden

[2]Mit der Umgebung eines Prozesses ist das Verhalten aller anderen Prozesse zusammen mit dem Verhalten einer eventuellen Umgebung des gesamten parallelen Systems gemeint.

kann. Dadurch kann die Anwendung dieser Technik wesentlich vereinfacht werden.

Um diese Probleme zu lösen, wurde die *Rely-Guarantee*-Methode (häufig auch *Assumption-Commitment*- oder auch *Assumption-Guarantee*-Methode genannt) entwickelt. Bei dieser Technik werden die lokalen Eigenschaften der Komponenten als Garantien (engl. *Guarantee*) formuliert, welche das von der Komponente garantierte Verhalten umfasst. Damit die Komponente diese Garantien erfüllt, können bestimmte Annahmen über die Umgebung mit der *Rely*-Bedingung getroffen werden. Informell muss dann für jede Komponente gezeigt werden, das ihre *Guarantee*-Bedingung erfüllt ist, wenn ihre *Rely*-Bedingung nicht verletzt wird. Die Garantie wird dann vom Modularisierungstheorem zum einen dazu benutzt, um zu zeigen, dass alle Komponenten die Umgebungsannahmen der anderen Komponenten nicht verletzen. Zum anderen werden die lokalen Garantien benutzt, um die Gesamteigenschaft für das parallele System abzuleiten.

Entwickelt wurde diese Methode zuerst von Cliff Jones [Jon83b] und J. Misra & K. M. Chandi [MC81]. Jones' *Rely-Guarantee*-Methode eignet sich für Systeme, deren Komponenten über gemeinsame Variablen (*shared-variables*) Daten austauschen. Etwa zur gleichen Zeit wie Jones' Methode wurde unter dem Namen *Assumption-Commitment* ein Formalismus von Misra & Chandi vorgestellt. Dieser Formalismus ist sehr ähnlich zur *Rely-Guarantee*-Methode, allerdings ist er auf verteilte Systeme spezialisiert, bei denen der Datenaustausch synchron über Nachrichten (*message passing*) erfolgt.

Beispiel 3 *(Semaphor – Fortsetzung).*
Damit die erste Komponente die Gesamtgarantie aufrecht erhalten kann (beide *PC*s sind niemals gemeinsam im kritischen Bereich), muss sie sich darauf verlassen können, unter welchen Umständen der andere Prozess den kritischen Bereich betritt, bzw. dass die Semaphor wieder zurückgesetzt wird. Der eigene Programmzähler darf außerdem von der Umgebung nicht verändert werden.

Informell ergeben sich damit die folgenden Umgebungsannahmen für die erste Komponente:

- Der eigene Programmzähler bleibt immer unverändert.
- Wenn die andere Komponente den kritischen Bereich in einem Schritt betritt (d. h., PC_2 wird von *critical* auf *noncritical* umgeschaltet), muss am Anfang dieses Schrittes die Semaphor noch frei sein (d. h. $Sem = \text{true}$) und nach diesem Schritt muss die Semaphor belegt sein (d. h., nach dem Umgebungsschritt muss $Sem = \text{false}$ gelten).
- Die Semaphor darf von der Umgebung nur dann zurückgesetzt werden, wenn die andere Komponente im kritischen Bereich ist (d. h. $PC_2 = critical$) und dieser im selben Schritt verlassen wird (d. h., nach dem Schritt gilt $PC_2 = noncritical$)
- In allen anderen Fällen sollen sowohl die Semaphor als auch der Programmzähler der anderen Komponente unverändert bleiben.

Da beide Komponenten in diesem Fall symmetrisch sind, muss jede Komponente diese Annahmen natürlich auch garantieren. Wie diese Eigenschaften formalisiert werden können, wird weiter unten in Abschnitt 3.5 gezeigt. □

3.4 Der Sustain-Operator

Ein wichtiger Aspekt ist, wie die Umgebungsannahmen und die Garantien zusammenspielen. Der naive Ansatz, für jede Komponente zu zeigen, dass sie ihre Garantie immer (d. h. auf dem gesamten Ablauf) erfüllt, wenn ihre Umgebungsannahme immer (d. h. ebenfalls auf dem gesamten Ablauf) erfüllt ist, reicht hier nicht aus. Der Versuch, auf diese Weise ein Modularisierungstheorem zu formulieren, führt sofort zu einem Zirkelschluss. Das folgende Beispiel verdeutlicht den Fehler: Eine Komponente kann z. B. im ersten Schritt ihre Garantie verletzen und damit eine Kette von Ereignissen auslösen, die einige Schritte später zur Verletzung ihrer Umgebungsannahme führen. Damit wäre die oben genannte Komponenteneigenschaft erfüllt, da ja die Umgebungsannahme nicht für den gesamten Ablauf gilt. Eine Komponente könnte bei diesem Ansatz also die zukünftige Verletzung ihrer Umgebungsannahme „ausnützen“, um ihre Garantie schon jetzt zu verletzen.

Um dies zu verhindern, ist es nötig, für alle Komponenten zu fordern, dass sie ihre Garantie erst verletzen dürfen, *nachdem* ihre Umgebungsannahme verletzt wurde. In [AL95] wurde dafür der $\overset{+}{\rightarrow}$ -Operator eingeführt (ein Name für diesen Operator wird in [AL95] nicht genannt). In dieser Arbeit wird der Operator auch *Sustain*-Operator genannt. Die Formel $R \overset{+}{\rightarrow} G$ besagt, dass G einen Schritt länger erfüllt sein muss als R. Andere *Rely-Guarantee*-Techniken verwenden meist einen ähnlichen Operator.

Mithilfe des Sustain-Operators kann jetzt für jede Komponente P gezeigt werden, dass sie ihre Garantie G erst verletzt, nachdem ihre Umgebungsannahme R verletzt wurde. Formal kann dies durch die folgende Formel ausgedrückt werden:

$$P \vdash R \overset{+}{\rightarrow} G$$

Mithilfe dieses Operators kann der oben beschriebene Zirkelschluss vermieden werden.

3.5 Einbettung von Rely-Guarantee Eigenschaften in ITL$^+$

Wie in Kapitel 2 bereits beschrieben erlaubt es ITL$^+$, Systemschritte als Relation zwischen ungestrichenen Variablen V und einfach gestrichenen Variablen V' darzustellen, während Umgebungsschritte als Relation zwischen einfach und doppelt gestrichenen Variablen V' und V'' ausgedrückt werden.[3] Dieser

[3] V steht hierbei für eine Menge von Variablen, V' bzw. V'' für die gleiche Menge, in der alle Variablen jeweils gestrichen bzw. doppelt gestrichen vorkommen.

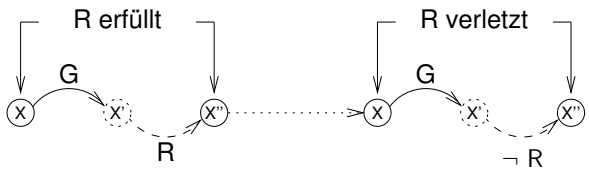


Abbildung 3.3: Beispiel für einen Ablauf von $R \xrightarrow{+} G$

Mechanismus erlaubt es, Bedingungen über System- bzw. Umgebungsschritte einfach durch Prädikate über ungestrichenen und gestrichenen Variablen bzw. einfach und doppelt gestrichenen Variablen auszudrücken. Bei der Formulierung von *Rely*- und *Guarantee*-Bedingungen kann man dies ausnutzen. Die *Rely*-Bedingung kann somit durch das Prädikat $R(V', V'')$ und die *Guarantee*-Bedingung durch $G(V, V')$ formuliert werden.

Beispiel 4 *(Semaphor - Fortsetzung).*
Die in Beispiel 3 informell beschriebenen *Rely*-Bedingung der ersten Komponente kann folgendermaßen in ITL+ formalisiert werden. Dabei werden die Variablen PC_1, PC_2 und Sem durch V abgekürzt:

$$\begin{aligned} R_1(V', V'') :\Leftrightarrow \quad & PC_1'' = PC_1' \\ & \wedge\ (PC_2' = \mathit{noncritical} \wedge PC_2'' = \mathit{critical} \rightarrow \mathit{Sem}' \wedge \neg\ \mathit{Sem}'') \\ & \wedge\ (PC_2' = \mathit{critical} \wedge PC_2'' = \mathit{noncritical} \rightarrow \neg\ \mathit{Sem}' \wedge \mathit{Sem}'') \\ & \wedge\ (PC_2' = PC_2'' \rightarrow \mathit{Sem}' = \mathit{Sem}'') \end{aligned}$$

Die Umgebungsannahme $R_2(V', V')$ für die zweite Komponente kann symmetrisch formuliert werden, d. h., die Vorkommen von PC_1 und PC_2 werden einfach vertauscht.

Da jede Komponente zum einen die Gesamteigenschaft erfüllen muss, zum anderen aber auch die Umgebungsannahme der anderen Komponenten nicht verletzen darf, kann die Garantie für die erste Komponente folgendermaßen formuliert werden:

$$\begin{aligned} G_1(V, V') :\Leftrightarrow \quad & \neg\ (PC_1' = \mathit{critical} \wedge PC_2' = \mathit{critical}) \\ & \wedge\ R_2(V, V') \end{aligned}$$

Die Garantie $G_2(V, V')$ der zweiten Komponente kann wiederum symmetrisch formuliert werden. □

3.6 Einbettung des Sustain-Operators in ITL$^+$

Da ITL+ bereits über temporallogische Operatoren verfügt, ist es möglich, den Sustain-Operator durch einen Standard TL-Operator auszudrücken. Die Formel

$$(G \wedge R)\ \textbf{unless}\ (G \wedge \neg\ R) \tag{3.2}$$

fällt genau mit der geforderten Semantik des Sustain-Operators zusammen. Die Abbildung 3.3 zeigt einen Beispielablauf mit dieser Formel. Die linke Seite der **unless**-Formel ist erfüllt, solange sowohl die Umgebungsannahme R als auch die Garantie G der Komponente erfüllt sind. Da der **unless**-Operator auch für Abläufe wahr ist, in denen die linke Teilformel in allen Zuständen gilt, fordert diese Formel nicht, dass die Garantie oder die Umgebungsannahme irgendwann verletzt werden müssen. Sollte die Umgebungsannahme aber doch in einem Ablauf verletzt werden, so behandelt die rechte Seite der **unless**-Formel den Schritt, in dem die Umgebungsannahme zum ersten Mal verletzt wird. Hier muss die Garantie auch im Systemschritt direkt vor der Verletzung der *Rely*-Bedingung noch gelten. Dadurch wird sichergestellt, dass die Garantie erst nach der Umgebungsannahme verletzt werden kann.

Die Formel (3.2) kann durch die folgende Äquivalenz noch weiter vereinfacht werden:

$$G \textbf{ unless } (G \wedge \neg\, R) \leftrightarrow (G \wedge R) \textbf{ unless } (G \wedge \neg\, R) \tag{3.3}$$

Dadurch kann der $\overset{+}{\rightarrow}$-Operator wie folgt definiert werden:

$$R \overset{+}{\rightarrow} G \quad :\equiv \quad G \textbf{ unless } (G \wedge \neg\, R) \tag{3.4}$$

Beispiel 5 *(Semaphor – Fortsetzung).*
Die oben entwickelte **unless**-Formel kann benutzt werden, um für eine einzelne Komponente nachzuweisen, dass sie ihre Garantie erfüllt solange ihre Umgebungsannahme erfüllt ist. Mit den in Beispiel 4 entwickelten *Rely*- und *Guarantee*-Bedingungen ergibt sich zum Beispiel für die erste Komponente S folgende ITL$^+$-Formel:

$$\begin{aligned} & S(;PC_1, Sem, State), PC_1 = noncritical \\ \vdash \quad & G_1(PC_1, PC_2, Sem, PC'_1, PC'_2, Sem') \\ \textbf{unless} \quad & G_1(PC_1, PC_2, Sem, PC'_1, PC'_2, Sem') \\ & \wedge \neg\, R_1(PC'_1, PC'_2, Sem', PC''_1, PC''_2, Sem'') \end{aligned}$$

□

Wie mit diesen Komponenteneigenschaften eine Gesamteigenschaft für das parallele System gezeigt werden kann, wird im nächsten Kapitel erläutert.

Kapitel 4

Interleaved parallele Systeme

Mithilfe des im vorherigen Kapitel definierten *Sustain*-Operators wird im Folgenden ein Modularisierungstheorem für den in Kapitel 2.2.6 vorgestellten Interleavingoperator entwickelt. Zur Spezifikation und zum Beweis dieses Theorems werden die Kompositionalität des Interleaving-Operators und die dynamische Prozesserzeugung mittels einer so genannten *Spawn-Prozedur* benötigt. Diese beiden Techniken werden im Abschnitt 4.1 vorgestellt. Im Abschnitt 4.2 wird das Modularisierungstheorem vorgestellt. Abschnitt 4.3 beschreibt den formal in KIV geführten Beweis. Im Abschnitt 4.5 wird die Anwendung des Theorems auf den im Kapitel 3 als laufendes Beispiel verwendeten Semaphor-Algorithmus vorgestellt. Im Anschluss daran wird in Abschnitt 4.6 mit dem *Producer-Channel-Consumer*-Beispiel die Anwendung des Theorems anhand einer Fallstudie gezeigt. Ein Teil der hier beschriebenen Ergebnisse wurde vom Autor bereits veröffentlicht [BBN+10, BNBR08].

4.1 Techniken zur Spezifikation und Verifikation des Modularisierungstheorems

In diesem Abschnitt werden zwei Techniken vorgestellt, die essentiell für die Spezifikation und den Beweis des entwickelten Modularisierungstheorems sind. Zum einen wird die Kompositionalität des Interleaving-Operators ausgenutzt, welche es erlaubt, nebenläufige Komponenten durch eine abstrakte Eigenschaft auszutauschen. Diese Eigenschaft wird im folgenden Kapitel 4.1.1 vorgestellt. Zusätzlich soll das Modularisierungstheorem für Systeme mit einer beliebigen, aber endlichen Anzahl von Komponenten spezifiziert werden. Dazu wird eine so genannte *Spawn-Prozedur* eingesetzt. Mit dieser können solche Systeme spezifiziert und mittels Induktion über die Anzahl der Komponenten auch verifiziert werden. Die *Spawn-Prozedur* und die dazugehörige Induktionstechnik werden in Abschnitt 4.1.2 erläutert.

4.1.1 Kompositionalität des Interleaving

Ein wesentliches Resultat von Balser[Bal05] ist ein kompositional definierter Interleavingoperator. Informell ist damit gemeint, dass sich in einem interleaved

parallelen System einzelne Komponenten durch abstraktere Komponenten ersetzen lassen. Formal wird die Kompositionalität durch das folgende Theorem begründet [Bal05]:

Satz 1 *(Kompositionale operationale Semantik).*
Sei $\varphi \in \mathbf{E}_{Bool}$ eine Formel und $\oplus$ ein Operator mit $\oplus(\varphi) \in \mathbf{E}_{Bool}$. Wenn die Semantik von $\oplus$ über Intervalle mit einer Funktion

$$\tilde{\oplus} \ : \ \mathcal{I} \to \mathcal{P}(\mathcal{I})$$

und der folgenden Eigenschaft

$$I \models \oplus(\varphi) \quad \textit{gdw.} \quad \textit{Es existiert ein } I_0 \textit{ mit } I \in \tilde{\oplus}(I_0) \textit{ und } I_0 \models \varphi$$

definiert ist, dann ist der Operator $\oplus$ *kompositional* und die folgende *Kompositionalitätseigenschaft* gilt:

$$\frac{\varphi \to \varphi'}{\oplus(\varphi) \to \oplus(\varphi')} \tag{4.1}$$

□

Der Beweis dieses Theorems ist in [Bal05] zu finden. Aus der Allgemeingültigkeit von $\varphi \to \varphi'$ kann aus der Formel (4.1) die Regel

$$\frac{\vdash \varphi \to \varphi' \quad \Gamma \vdash \oplus(\varphi), \Delta}{\Gamma \vdash \oplus(\varphi'), \Delta}$$

gefolgert werden.

Der in Kapitel 2.2.6 definierte Interleaving-Operator $\varphi_1 \parallel \psi$ ist semantisch über die Funktion

$$\|_n \ : \ \mathcal{I} \times \mathcal{I} \to \mathcal{P}(\mathcal{I})$$

definiert. Wenn das durch $I_\psi \models \psi$ definierte Intervall I_ψ als fest angenommen wird, ergibt sich mit

$$\tilde{\oplus}(I) \ := \ I \ \|_n \ I_\psi$$

genau die in Theorem 1 geforderte Funktion. Analog lässt sich dies natürlich auch für die rechte Komponente anwenden, um ψ durch eine abstraktere Komponente zu ersetzen.

Dadurch lassen sich die beiden folgenden Kalkülregeln für den Interleavingoperator ableiten:

$$\frac{\vdash \varphi_1 \to \varphi_2 \quad \varphi_2 \parallel \psi, \Gamma \vdash \Delta}{\varphi_1 \parallel \psi, \Gamma \vdash \Delta} \ \textit{ilv lem 1}$$

$$\frac{\vdash \varphi_1 \to \varphi_2 \quad \psi \parallel \varphi_2, \Gamma \vdash \Delta}{\psi \parallel \varphi_1, \Gamma \vdash \Delta} \ \textit{ilv lem 2}$$

Damit kann die nebenläufige Komponente φ_1 durch eine abstraktere Komponente φ_2 ersetzt werden, falls $\varphi_1 \to \varphi_2$ gilt.

4.1.2 Dynamische Prozesserzeugung

Um das Modularisierungstheorem flexibel einsetzen zu können, muss es möglichst generisch formuliert sein. Dazu ist es unter anderem wichtig, das Theorem nicht auf eine feste Anzahl von Prozessen zu beschränken, sondern eine beliebige Anzahl von nebenläufigen Komponenten zuzulassen. Um ein solches Theorem formal spezifizieren und beweisen zu können, wird eine geeignete Spezifikations- und Induktionstechnik benötigt.

Das dazu entwickelte Schema zur Verifikation von Systemen mit beliebig vielen Prozessen $Proc_i$ benutzt eine rekursiv definierte Spawn-Prozedur:

$$
\begin{array}{l}
\mathit{SpawnProc}(n;\, \mathit{Vars})\ \{ \\
\quad \textbf{if}^{*}\ n = 0\ \textbf{then} \\
\quad\quad P(0;\, \mathit{Vars}) \\
\quad \textbf{else} \\
\quad\quad P(n;\, \mathit{Vars}) \parallel \mathit{SpawnProc}(n-1;\, \mathit{Vars}) \\
\}
\end{array}
$$

Die Spawn-Prozedur $\mathit{SpawnProc}(n;\, \mathit{Vars})$ kann durch die rekursive Spezifikation für jedes n zu

$$\mathit{Proc}(n;\, \mathit{Vars}) \parallel \ \ldots\ \parallel \mathit{Proc}(0;\, \mathit{Vars})$$

expandiert werden. Der Value-Parameter n wird dabei als Prozess-Identifier für die nebenläufigen Prozesse *Proc* benutzt. Mit diesem Identifier können die verschiedenen *Proc*-Komponenten unterschieden werden. Die Referenz-Variable *Vars* ist hier nur ein Platzhalter. Für diese Variable können die von der Prozedur *Proc* benötigten Variablen eingesetzt werden.

Eigenschaften eines mit einer solchen Spawn-Prozedur spezifizierten Systems können mit Induktion über die Variable n bewiesen werden. Dabei sind die folgenden Beweisschritte notwendig: Um zu zeigen, dass eine Eigenschaft *Prop* für ein System mit einer beliebigen Anzahl von parallel laufenden Subprozessen *Proc* erfüllt ist, startet man mit der Sequenz

$$\mathit{SpawnProc}(n;\, \mathit{Vars}) \ \vdash\ \mathit{Prop}$$

Der Beweis wird durch strukturelle Induktion über die Variable n geführt. Als Induktionsanfang ist dann

$$\mathit{Proc}(0;\, \mathit{Vars}) \ \vdash\ \mathit{Prop}$$

zu beweisen. Der Induktionsschritt führt zu folgender Sequenz:

$$\mathit{SpawnProc}(n+1;\, \mathit{Vars}),\ \mathbf{A}\bullet\ \forall\ \mathit{Vars}.(\mathit{SpawnProc}(n;\, \mathit{Vars}) \rightarrow \mathit{Prop}) \ \vdash\ \mathit{Prop}$$

Dabei ist die Formel $\mathit{SpawnProc}(n;\, \mathit{Vars}) \rightarrow \mathit{Prop}$ die Induktionshypothese. Die Operatoren $\mathbf{A}\bullet$ müssen benutzt werden, da die Induktionshypothese unabhängig vom betrachteten Intervall ist. Wenn man $\mathit{SpawnProc}(n+1;\, \mathit{Vars})$ zu

$$\mathit{Proc}(n+1;\, \mathit{Vars}) \parallel \mathit{SpawnProc}(n;\, \mathit{Vars})$$

expandiert, kann die rechte Komponente dieser Formel mit der Induktionshypothese und mit Lemma *ilv lem 2* durch *Prop* ersetzt werden. Damit ergibt sich die folgende Formel:[1]

$$Proc(n; Vars) \parallel Prop \vdash Prop$$

Diese Formel lässt sich üblicherweise interaktiv durch symbolische Ausführung von *Proc* und *Prop* beweisen.

4.2 Modularisierungstheorem für interleaved parallele Systeme

In diesem Abschnitt wird das Rely-Guarantee-Theorem für den Interleaving-Operator vorgestellt. Dazu wird in Abschnitt 4.2.1 zuerst ein einfaches Modularisierungstheorem entwickelt, das den Zusammenhang zwischen den einzelnen Rely- und Guarantee-Bedingungen beschreibt. Da üblicherweise Beweise über Systeme auf Invarianten basieren, wird dieses einfache Theorem im darauf folgenden Abschnitt 4.2.2 um Invarianten erweitert.

4.2.1 Grundlegendes Rely-Guarantee Theorem

Ziel ist es, eine Gesamteigenschaft des Systems

$$P_1 \parallel \ldots \parallel P_n \quad \vdash \quad R(S', S'') \stackrel{+}{\rightarrow} G(S, S') \tag{4.2}$$

aus den Rely-Guarantee-Eigenschaften der Komponenten herzuleiten. Das heißt, für alle P_i kann angenommen werden, dass

$$P_i \vdash R_i(S', S'') \stackrel{+}{\rightarrow} G_i(S, S') \tag{4.3}$$

gilt. Damit aus dieser Eigenschaft das Gesamtziel gefolgert werden kann, müssen die einzelnen Rely- und Guarantee-Eigenschaften zusätzlich noch bestimmte Bedingungen erfüllen. Es gilt, diese Bedingungen so zu wählen, dass die Komponenten sich wechselseitig nicht stören und ihre *Rely*-Bedingungen nicht verletzen können. Im Folgenden werden diese zusätzlichen Bedingungen motiviert und erläutert.

Zum einen muss das Gesamtsystem die globale Garantie erfüllen, wenn alle Komponenten ihre lokale Garantien erfüllen.[2] Damit die globale Garantie erfüllt ist, muss jeder Systemschritt des Gesamtsystems $G(S, S')$ erfüllen. Da jeder Schritt des Gesamtsystems ein Schritt einer seiner Komponenten ist, kann dies durch folgende Formel ausgedrückt werden:

$$\forall i.\ G_i(S, S') \rightarrow G(S, S') \tag{4.4}$$

Die globale Garantie muss aus jeder lokalen Garantie folgen.

[1] Die Induktionshypothese wird hier nicht mehr benötigt und daher weggelassen.

[2] Als lokal werden Eigenschaften bezeichnet, die für einzelne Komponenten gelten. Dementsprechend gelten globale Eigenschaften für das Gesamtsystem.

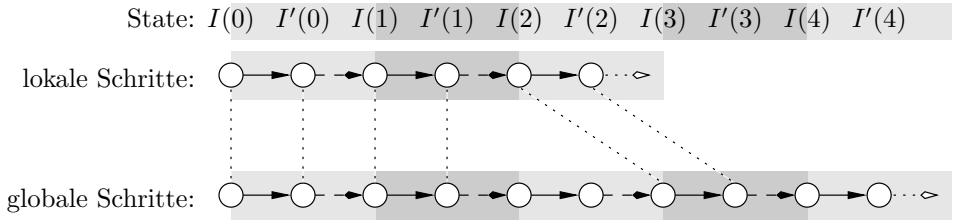


Abbildung 4.1: Zusammenhang zwischen lokalen und globalen Schritten

Die Komponenten erfüllen ihre lokalen Garantien aber nur so lange wie ihre Umgebungsannahmen (d. h. die lokalen *Rely*-Bedingungen) erfüllt sind. Im einfachsten Fall fällt der lokale mit dem globalen Umgebungsschritt zusammen. Das heißt, die lokale *Rely*-Bedingung R_i muss aus der globalen *Rely*-Bedingung folgen. Ein Beispiel hierfür wäre der erste Umgebungsschritt zwischen $I'(0)$ und $I(1)$ in Abbildung 4.1. Die folgende Bedingung erzwingt, dass die lokale Umgebungsannahme in diesem Fall erhalten bleibt:

$$\forall\, i.\ R(S, S') \rightarrow R_i(S, S') \tag{4.5}$$

Der lokale Umgebungsschritt kann sich jedoch auch aus mehreren globalen Schritten zusammensetzen. Globale Umgebungsschritte und Systemschritte anderer Komponenten bilden zusammen einen einzigen lokalen Umgebungsschritt. Ein Beispiel hierfür wäre der lokale Umgebungsschritt zwischen $I'(1)$ und $I(2)$ in Abbildung 4.1, der durch zwei globale Umgebungsschritte und einen Systemschritt einer anderen Komponente gebildet wird. Dass die Systemschritte G_j der anderen Komponenten in diesem Fall die lokale Umgebungsannahme R_i nicht zerstören, wird durch folgende Bedingung ausgedrückt:

$$\forall\, i, j.\ (i \neq j \wedge G_j(S, S')) \rightarrow R_i(S, S') \tag{4.6}$$

Eine Komponente muss ihre eigene *Rely*-Bedingung nicht erfüllen. Diese wichtige Eigenschaft ist auf den ersten Blick überraschend. So kommt es häufig vor, das eine Komponente eine lokale Variable zwar selbst ändert, sich aber darauf „verlässt“, dass diese nicht von der Umgebung geändert wird. Ein Beispiel dafür wäre ein Semaphor-Prozess. Die *Rely*-Bedingung eines Semaphor-Prozesses i verlangt, dass der Programmzähler PC_i nicht von der Umgebung verändert wird. Allerdings verändert der Prozess i selbst diese Variable und kann daher nicht seine *Rely*-Bedingung erhalten.

Zusätzlich wird die Transitivität der lokalen Umgebungsannahme verlangt:

$$\forall\, i.\ R_i(S, S') \wedge R_i(S', S'') \rightarrow R_i(S, S'') \tag{4.7}$$

Diese Bedingung ist nötig, damit ein lokaler Umgebungsschritt, der aus mehreren globalen Schritten zusammengesetzt wird, die lokale *Rely*-Bedingung erfüllt.

Die fünf Bedingungen (4.3) bis (4.7) reichen aus, um die gewünschte Gesamteigenschaft (4.2) zu zeigen:

Satz 2 *Modularisierungstheorem.*
Sei G, R, G_i, R_i und P_i so, dass (4.3) bis (4.7) gilt. Dann gilt auch:

$$P_1 \parallel \ldots \parallel P_n \ \vdash\ R(S', S'') \stackrel{+}{\rightarrow} G(S', S'')$$

Bei der Anwendung zeigt sich aber, dass mit diesen vier Bedingungen alleine die Formulierung der Rely- und Guarantee-Bedingungen mitunter sehr redundant wird. Im nächsten Abschnitt wird ein um Invarianten erweitertes Theorem vorgestellt, mit dem diese Redundanzen vermieden werden.

4.2.2 Kompositionstheorem mit Invarianten

Häufig enthält die Gesamtgarantie Variablen, die nicht in den Komponenten vorkommen. Wegen der Beweisverpflichtung (4.4) ist es nötig, dass die lokalen Garantien ebenfalls Wissen über diese Variablen enthalten. Ebenso müssen die lokalen Garantien Informationen über alle Variablen enthalten, die in den lokalen Umgebungsannahmen vorkommen, damit die Beweisverpflichtung (4.6) gezeigt werden kann. Meistens sind diese Informationen über die Variablen invariant, das heißt, über alle Zustände des Intervalls gleich. Solche Informationen können zum Beispiel sein, dass eine lokale Liste immer ein Präfix einer globalen Liste ist. Am Ende dieses Kapitels wird im ProChaCon-Beispiel eine solche Invariante benutzt.

Um Invarianten im vorgestellten Modularisierungstheorem (s. vorhergehender Abschnitt) zu benutzen, würde man diese zu allen Bedingungen über eine Implikation hinzufügen. So würde man zum Beispiel die lokalen Guarantee-Bedingungen wie folgt formulieren:

$$G_i(S, S') :\Leftrightarrow I(S) \rightarrow (Gstep_i(S, S') \wedge I(S'))$$

Dabei ist die Formel $I(S)$ die Invariante und $Gstep_i(S, S')$ beschreibt die erlaubten Schritte, die die lokale Komponente garantieren muss. Ähnlich müssen die lokalen Umgebungsannahmen, sowie die globalen Rely- und Guarantee-Bedingungen formuliert werden. Diese Formulierung führt aber zu häufiger Redundanz, wodurch der Beweis der Prämissen des Modularisierungstheorems unübersichtlich wird.

Um diesen Mehraufwand zu vermeiden, wird zusätzlich zu den Garantien und Umgebungsannahmen noch eine Invariante I definiert, die direkt in das Modularisierungstheorem integriert wird. Dadurch werden die Rely- und Guarantee-Bedingungen kompakter und übersichtlicher. Außerdem wird die explizite Invariante im Folgenden in verschiedenen Erweiterungen des Modularisierungstheorems benutzt. Diese Erweiterungen werden im Abschnitt 4.4 und bei der Beschreibung lock-freier Algorithmen (s. Kap. 7 und 8) behandelt.

Um eine explizite Invariante in das Modularisierungstheorem zu integrieren, muss zusätzlich zu den oben vorgestellten Beweisverpflichtungen gezeigt werden, dass die Invariante auf dem gesamten Intervall erhalten bleibt. Damit ergibt sich das folgende Modularisierungstheorem:

Satz 3 *(Modularisierungstheorem für Interleaving).*
Sei S eine Variable vom Typ *state* und seien P_i mit $i = 1, \ldots, n$ Prozeduren mit einem Parameter vom Typ *state*, I ein Prädikat für die Invariante über eine

Variable vom Typ *state*, sowie G, R, G_i und R_i zweistellige Prädikate vom Typ $state \times state$. Wenn für alle $i = 1, \ldots, n$ gilt:

1. $P_i(S), I(S) \vdash R_i(S', S'') \xrightarrow{+} G_i(S, S')$
2. $G_i(S, S') \wedge I(S) \rightarrow G(S, S') \wedge \bigwedge_{j \in \{1..n\} \wedge j \neq i} R_j(S, S')$
3. $R_i(S, S') \wedge R_i(S', S'') \wedge I(S) \wedge I(S') \rightarrow R_i(S, S'')$
4. $R(S, S') \wedge I(S) \rightarrow \bigwedge_{j \in \{1..n\}} R_j(S, S')$
5. $G_i(S, S') \wedge I(S) \rightarrow I(S')$
6. $R(S', S'') \wedge I(S') \rightarrow I(S'')$

Dann gilt auch:

$$P_1(S) \parallel \ldots \parallel P_n(S),\ I(S) \vdash R(S', S'') \xrightarrow{+} G(S, S') \qquad \square$$

Die Beweisverpflichtungen *1. - 4.* entsprechen im wesentlichen den Beweisverplichtungen (4.3) bis (4.7) aus Satz 2 aus Abschnitt 4.2.1. Die Invariante kann in diesen vier Beweisverpflichtungen jedoch als zusätzliche Voraussetzung benutzt werden.

In der Beweisverpflichtung *1.* wird für alle Komponenten gezeigt, dass die einzelnen Komponenten ihre Rely-/Guarantee-Bedingungen erfüllen (dies entspricht der Formel (4.3)). Die Formeln (4.4) und (4.6) wurden zur Beweisverpflichtung *2.* zusammengefasst. Hier wird gezeigt, dass die lokalen Garantien die globale Garantie und die lokalen Rely-Bedingungen der anderen Komponenten erfüllen. Die Transitivität der lokalen Rely-Bedingungen wird in der Beweisverpflichtung *3.* gezeigt (entspricht Formel (4.7)). In der Beweisverpflichtung *4.* wird (entsprechend der Formel (4.5)) gezeigt, dass die globale Rely-Bedingung alle lokalen Rely-Bedingungen impliziert.

Mit den beiden neuen Beweisverpflichtungen *5.* und *6.* in Satz 3 wird nachgewiesen, dass die Invariante auf dem ganzen Intervall erhalten bleibt. Da das Gesamtintervall aus lokalen Systemschritten und globalen Umgebungsschritten zusammengesetzt ist, genügt es zu zeigen, dass sowohl für die lokale Guarantee-Bedingung als auch die globale Rely-Bedingung die Invariante erhalten bleibt.

Die Einschränkung des Modularisierungstheorems auf eine Systemvariable S ist in der Praxis keine wesentliche Einschränkung. Falls mehrere Parameter benötigt werden, kann ein Produktdatentyp definiert werden, der alle gewünschten Typen enthält (in KIV werden dazu *Instantiated specs* benutzt).

Da die Conclusio des Theorems die gleiche Struktur hat wie die erste Beweisverpflichtung des Theorems, kann das Theorem bei großen Systemen auch transitiv angewendet werden. Das heißt, dass man z. B. ein nebenläufiges System in mehrere jeweils wiederum nebenläufige Teilkomponenten zerlegen kann. Für jede Teilkomponente kann man das Theorem anwenden, um für die einzelnen Komponenten jeweils eine Rely-/Guarantee-Eigenschaft zu zeigen. In einem zweiten Schritt zeigt man dann mit dem Theorem, dass aus den nachgewiesenen Rely-/Guarantee-Eigenschaften die Gesamteigenschaft folgt.

4.3 Spezifikation der Theorie und Beweis in KIV

Die oben beschriebenen Modularisierungstheoreme wurden in KIV spezifiziert und bewiesen. Im Folgenden wird die formale Spezifikation der Theorie für das Modularisierungstheorem aus Satz 3 und dessen Beweis vorgestellt. Das einfachere Theorem aus Satz 2 kann daraus gefolgert werden, indem Satz 3 mit einer leeren Invariante (d. h. $I(S) \leftrightarrow \text{true}$) benutzt wird.

Der Beweis ist wie folgt aufgebaut: Im folgenden Abschnitt 4.3.1 wird zunächst die Struktur der benutzten Spezifikationen erklärt. Dabei wird vor allem auch die Spawn-Prozedur definiert, mit der das parallele System erzeugt wird.

Danach wird im Abschnitt 4.3.2 mit dem Lifting-Lemma induktiv gezeigt, dass ein paralleles System eine Rely-/Guarantee-Bedingung erfüllt, wenn alle Komponenten ihre lokalen Rely-/Guarantee-Bedingungen erfüllen. Die Guarantee-Bedingung dieses Systems ist dabei die Disjunktion aller lokalen Komponenten-Guarantee-Bedingungen, die Rely-Bedingung des Systems dagegen die Konjunktion aller lokalen Komponenten-Rely-Bedingungen.

In Abschnitt 4.3.4 wird im Modularisierungslemma durch Anwendung des Lifting-Lemmas gezeigt, dass die in Abschnitt 4.3.1 definierte Spawn-Prozedur eine Disjunktion aller lokalen Komponenten-Guarantee-Bedingungen solange erfüllt, wie die Konjunktion aller lokalen Komponenten-Rely-Bedingungen erfüllt ist.

Schließlich wird das Modularisierungslemma in Abschnitt 4.3.5 benutzt, um das Modularisierungstheorem zu beweisen.

4.3.1 Struktur

Die Komponenten P des Modularisierungstheorems werden nur abstrakt spezifiziert, das heißt, es werden nur ihre Parameter spezifizert, aber nicht der Prozedurrumpf. Dieser kann dann in einer Fallstudie durch den entsprechenden Prozedurrumpf $P(n; S) \in \mathbf{PROC}$ instanziert werden.

Um die Spezifikation möglichst allgemein zu halten, wird dabei als Reference-Parameter nur der Wert $S \in \mathbf{State}$ mit der abstrakte Sorte **State** benutzt. Für diesen Parameter kann bei Instanzierung des Theorems ebenfalls ein zusammengesetzter Typ, der die benötigten Werte enthält, eingesetzt werden. Zusätzlich wird zur Unterscheidung der verschiedenen Prozesse ein Bezeichner benötigt. Dazu wird jedem Prozess zusätzlich der Value-Parameter $n \in \mathbf{Nat}$ übergeben, der als Prozessbezeichner benutzt wird. Damit ist $P(n; S) \in \mathbf{PROC}_{Nat}^{\mathbf{State}}$.

Die Rely- und Guarantee-Bedingungen der Komponenten werden als dreistellige Prädikate $\mathbf{Nat} \times \mathbf{State} \times \mathbf{State}$ spezifiziert. Der erste Parameter **Nat** steht wie oben für den Prozessbezeichner. Die beiden **State**-Parameter dienen, wie in Kapitel 3 beschrieben, für die ungestrichenen und gestrichenen Werte (bzw. gestrichenen und doppelt gestrichenen Werte bei den Rely-Bedingungen). Die beiden globalen Rely- und Guarantee-Bedingungen werden als als zweistellige Prädikate $\mathbf{State} \times \mathbf{State}$ definiert.

Um das Gesamtsystem mit n Komponenten zu modellieren, wird eine Spawn-

Prozedur benutzt, wie sie in Abschnitt 4.1.2 vorgestellt wurde.

$$
\begin{array}{l}
\mathit{SpawnP}(n; S)\ \{ \\
\quad \textbf{if}^{*}\ n = 0\ \textbf{then} \\
\quad\quad P(0; S) \\
\quad \textbf{else} \\
\quad\quad P(n; S) \parallel \mathit{SpawnP}(n-1; S) \\
\}
\end{array}
$$

Die Prozedur *SpawnP* erzeugt $n+1$ Prozesse P, denen jeweils ein Prozessbezeichner von 0 bis n zugewiesen wird. Dass jeder Prozess P seine Rely- und Guarantee-Bedingung erfüllt, wird als Axiom spezifiziert. Ebenso werden die anderen fünf prädikatenlogischen Beweisprämissen des Modularisierungstheorems als Axiome spezifiziert.

Für den Beweis des Gesamtsystems wird, wie in Abschnitt 4.1.2 bereits beschrieben, Induktion über die Anzahl der Komponenten benutzt. Informell zeigt man dabei im Induktionsschritt: Wenn n Komponenten ihre Rely- und Guarantee-Bedingungen erfüllen, dann gilt dies auch für $n+1$ Komponenten.

Um dies formalisieren zu können, werden die Prädikate

$$\mathit{Rto} : \mathbf{Nat} \times \mathbf{State} \times \mathbf{State}$$

und

$$\mathit{Gto} : \mathbf{Nat} \times \mathbf{State} \times \mathbf{State}$$

eingeführt. $\mathit{Rto}(n, s_1, s_2)$ besagt dabei, dass im Schritt von Zustand s_1 zu Zustand s_2 alle Rely-Bedingungen der Prozesse mit den Prozessbezeichnern von 0 bis n erfüllt sind. Formal wird dies durch das folgende Axiom ausgedrückt:

$$\vdash \mathit{Rto}(n, s_1, s_2) \leftrightarrow (\forall\ m.m \leq n \rightarrow R(m, s_1, s_2))$$

Das Prädikat $\mathit{Gto}(n, s_1, s_2)$ besagt, dass im Schritt von Zustand s_1 zu Zustand s_2 mindestens eine Guarantee-Bedingung G_m erfüllt ist. Dies wird durch das folgenden Axiom formalisiert:

$$\vdash \mathit{Gto}(n, s_1, s_2) \leftrightarrow (\exists\ m.m \leq n \rightarrow G(m, s_1, s_2))$$

Im Folgenden wird auch die kürzere Schreibweise Rto_n für $\mathit{Rto}(n, s_1, s_2)$ und Gto_n für $\mathit{Gto}(n, s_1, s_2)$ benutzt, wenn s_1 und s_2 aus dem Kontext ersichtlich sind.

Für diese Prädikate lassen sich Lemmas analog zu den Beweisprämissen aus Satz 3 formulieren. Für die Beweisprämisse 2. sind dafür zwei Lemmas notwendig:

$$\mathit{Gto}(n, S, S') \wedge I(S) \rightarrow G(S, S') \wedge R(n+1, S, S')$$

$$\mathit{Gto}(n+1, S, S') \wedge I(S) \rightarrow G(S, S') \wedge \mathit{Gto}(n, S, S')$$

Die Komponenten 0 bis n und die Komponente $n+1$ müssen gegenseitig ihre Umgebungssannahmen erhalten. Für die anderen Prämissen ist jeweils nur ein Lemma notwendig, also z. B.:

$$\mathit{Rto}(n, S, S') \wedge \mathit{Rto}(n, S', S'') \wedge I(S) \rightarrow \mathit{Rto}(n, S, S'')$$

Zusätzlich sind noch einige Lemmas über *Rto* und *Gto* hilfreich, z. B.

$$Gto(n, S, S') \rightarrow Gto(n+1, S, S')$$

oder

$$Rto(n+1, S, S') \rightarrow Rto(n, S, S')$$

4.3.2 Lifting Lemma

Zentral für den Beweis des Modularisierungstheorems ist das *Lifting-Lemma*. Ziel ist es, den Beweis des Modularisierungstheorems induktiv über die Anzahl der Komponenten zu führen. Mit diesem Lemma wird die Hauptbeweisverpflichtung des Induktionsschritts formalisiert und gezeigt.

Lemma 1 *(Lifting-Lemma).* Unter den Voraussetzung des Modularisierungstheorems (Satz 3) gilt für alle n:

$$\begin{aligned} & R(n+1, S', S'') \overset{+}{\rightarrow} G(n+1, S, S') \parallel Rto(n, S', S'') \overset{+}{\rightarrow} Gto(n, S, S') \ , \\ & I(S) \\ \vdash\ & Rto(n+1, S', S'') \overset{+}{\rightarrow} Gto(n+1, S, S') \end{aligned}$$

□

Informell wird im Lifting-Lemma induktiv gezeigt, dass die einzelnen Komponenten ihre Rely-Bedingungen untereinander jeweils durch ihre Guarantee-Bedingungen aufrechterhalten. Somit werden die lokalen Garantien so lange erfüllt, bis eine der lokalen Umgebungsannahmen von außen verletzt wird. Der linke Teil des im Lifting-Lemma beschriebenen nebenläufigen Systems im Antezedent enthält die Rely-/Guarantee-Bedingung des Prozesses $n+1$. Der rechte Teil steht für die Prozesse 0 bis n. Die Formel auf der rechten Seite des Interleaving-Operators besagt, dass die Systemschritte dieser Komponente immer *eine* der Garantien der Prozesse 0 bis n erfüllen, solange die Umgebungsschritte immer *alle* Umgebunsbedingungen der Prozesse 0 bis n erfüllen. Zusätzlich wird angenommen, dass die Invariante im ersten Zustand gilt. Gezeigt wird, dass alle Schritte dieses Systems in jedem Systemschritt die Garantie-Bedingung *eines* der Prozesses 0 bis $n+1$ erfüllen, solange die Rely-Bedingungen aller Prozesse 0 bis $n+1$ in jedem Umgebungsschritt erfüllt sind.

Beweis
Der Beweis dieses Lemmas erfolgt über symbolische Ausführung mit Induktion. Abbildung 4.2 zeigt den Beweisbaum für den Beweis. Prädikatenlogische Vereinfachungen, Axiome und Lemmas werden vom KIV-Simplifier automatisch angewendet. Sie sind im Beweisbaum aus Gründen der Übersichtlichkeit ausgelassen.

Da der $\overset{+}{\rightarrow}$-Operator durch den **unless** -Operator definiert ist (siehe Kapitel 3.5), kann dieser — wie in Kapitel 2.5.1 beschrieben — zur Induktion über Sicherheitsformeln benutzt werden. Der erste Schritt in dem Beweis ist ein symbolischer Ausführungsschritt. Dabei kann angenommen werden, dass die Formel $Rto(n+1, S', S'')$ während des Schrittes erfüllt ist (ansonsten „terminiert“ die

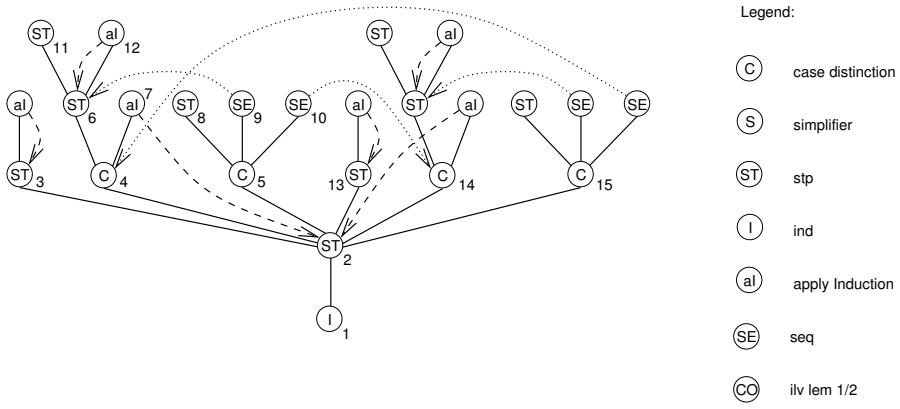


Abbildung 4.2: Beweisbaum für Lemma 1

Sustain-Formel im Sukzedens und der Beweis kann einfach geschlossen werden). Die sechs entstehenden Beweisäste entsprechen, wie in Kapitel 2.2.6 beschrieben, den sechs möglichen Schritten beim Interleaving. Im Folgenden werden zuerst die Knoten 3–5 beschrieben. Diese entsprechen einem Schritt des linken Programms.

Knoten 3 behandelt den Fall, in dem die erste Komponente terminiert ist. Hier muss gezeigt werden, dass die Formel

$$Rto(n, S', S'') \xrightarrow{+} Gto(n, S, S') \vdash Rto(n+1, S', S'') \xrightarrow{+} Gto(n+1, S, S')$$

gilt. Dies geschieht durch einen weiteren symbolischen Ausführungsschritt und Anwendung der Induktionshypothese auf Knoten 3, da sich die Formel nach einem Schritt wiederholt.

In Knoten 4 hat die erste Komponente einen normalen Schritt gemacht (das heißt, sie wurde weder terminiert noch war sie geblockt). Eine Fallunterscheidung ist nötig, um zu entscheiden, ob die Umgebungsannahme R_{n+1} der ersten Komponente im letzten Schritt verletzt wurde (Knoten 6) oder nicht (Knoten 7). Knoten 6 enthält die folgende Sequenz:

$$\begin{aligned} & \neg\ R(n+1, s_0, S) \parallel Rto(n, S', S'') \xrightarrow{+} Gto(n, S, S'), \\ & Rto(n+1, s_0, S) \\ \vdash\ & Rto(n+1, S', S'') \xrightarrow{+} Gto(n+1, S, S') \end{aligned}$$

Da die linke Komponente einen Schritt ausgeführt hat, wurde durch die symbolische Ausführung deren Variable S' durch die neue statische Variable s_0 und S'' durch S ersetzt. Das gleiche gilt für die globale Umgebungsannahme Rto. Mit der Definition von Rto kann die Formel $Rto(n+1, s_0, S)$ im Antecedens zu $R(n+1, s_0, S)$ vereinfacht werden. Die entstehende Sequenz stellt eine typische Situation bei abstrakten Rely-Guarantee-Beweisen dar. Daher wird der Beweis dieser Sequenz im nächsten Abschnitt 4.3.3 detailliert beschrieben. Knoten 7 enthält die gleiche Sequenz wie Knoten 2, daher kann dieser Knoten durch Induktion geschlossen werden.

Der Knoten 5 entspricht dem Fall, in dem die linke Komponente einen blockierten Schritt vollzogen hat. Hier sind ebenfalls drei Fälle möglich.

- Beide Annahmen R_{n+1} und Rto_n sind verletzt worden. Dies kann zu einem Widerspruch geführt werden, da Rto_{n+1} global gilt.
- Nur die Annahme R_1 ist verletzt (Knoten 9). Dies ist der gleiche Fall wie bei Knoten 6, daher kann dieser Ast mit *insert proof lemma* geschlossen werden.
- Die rechte Komponente hat einen nichtblockierten Schritt durchgeführt und beide Umgebungsannahmen R_{n+1} und Rto_n gelten weiterhin (Knoten 10). Dieser Fall ist in Knoten 13 behandelt, daher kann dieser Ast ebenfalls mit *insert proof lemma* geschlossen werden.

Die Knoten 13–15 entstehen durch einen Schritt der rechten Komponente $Rto(n, S', S'') \xrightarrow{+} Gto(n, S, S')$. Auch hier entstehen wieder drei Knoten, einer für Terminierung der rechten Komponente, der zweite für einen normalen Schritt und der dritte für einen blockierten Schritt. Alle drei Fälle sind symmetrisch zu den oben beschriebenen Knoten 3–5 und lassen sich daher analog beweisen. So führt zum Beispiel Knoten 14 analog zu Knoten 4 zur symmetrischen Fallunterscheidung, ob die Rely-Bedingung der rechten Komponente im letzten Schritt verletzt wurde oder nicht. Falls sie verletzt wurde, ist folgende Sequenz zu zeigen:

$$\begin{aligned} & R(n+1, S', S'') \xrightarrow{+} G(n+1, S, S') \parallel \neg\, Rto(n, s_0, S), \\ & Rto(n+1, s_0, S) \\ \vdash\ & Rto(n+1, S', S'') \xrightarrow{+} Gto(n+1, S, S') \end{aligned}$$

Auch hier kann die in Abschnitt 4.3.3 beschriebene Beweisstrategie verwendet werden, um diese Sequenz zu beweisen. Der andere Fall kann wieder mit Induktion bewiesen werden.

Ebenso kann Knoten 13 analog zu Knoten 3 bewiesen werden und Knoten 15 analog zu Knoten 5. □

4.3.3 Verletzte Umgebungsannahmen innerhalb der Komponenten

Eine Situation, die immer wieder in abstrakten Rely-Guarantee-Beweisen auftritt, wird durch die folgende Sequenz dargestellt:

$$\begin{aligned} & \neg\, R(n+1, s_0, S) \parallel Rto(n, S', S'') \xrightarrow{+} Gto(n, S, S'), \\ & R(n+1, s_0, S) \\ \vdash\ & Rto(n+1, S', S'') \xrightarrow{+} Gto(n+1, S, S') \end{aligned} \tag{4.8}$$

Diese Sequenz entspricht dem Knoten 4 des Lifting-Beweises. Im Umgebungsschritt der linken Komponente wurde die Rely-Bedingung $R(n+1, s_0, S)$ im letzten Schritt verletzt, während global die gleiche Formel $R(n+1, s_0, S)$ erfüllt

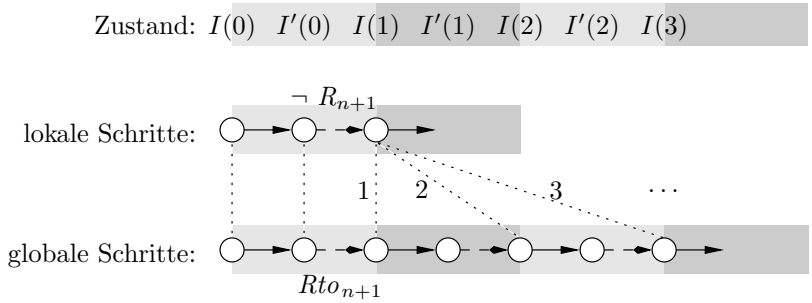


Abbildung 4.3: Verletzte Umgebungsannahme einer Komponente

ist. Die Situation der Sequenz entspricht dem Zustand $I(1)$ in Abbildung 4.3. Die Werte von S im Zwischenzustand $I'(0)$ wurden dabei durch die *stp*-Regel (Tabelle 2.2, Seite 26) im letzten Schritt durch statische Variablen s_0 ersetzt. Obwohl es einfach erscheint, hier einen Widerspruch zu zeigen, sind zum Beweis mehrere Schritte notwendig. Die Schwierigkeit besteht darin, dass die lokale und die globale Belegung von S nicht übereinstimmen muss, da noch nicht festgelegt ist, mit welchem globalen Zustand der lokale übereinstimmt. Falls im nächsten Schritt wieder die linke Komponente ausgeführt wird, gilt der Zusammenhang zwischen globalem und lokalem Zustand, der in Abbildung 4.3 durch die gepunktete und mit 1 gekennzeichnete Linie markiert ist. Falls die linke Komponente aber erst wieder im übernächsten Schritt ausgeführt wird, gilt die durch die Linie 2 gekennzeichnete Beziehung. In diesem Fall setzt sich der lokale Umgebungsschritt $R(n+1, s_0, S)$ aus einem globalen Umgebungsschritt, einem Systemschritt der rechten Komponente und einem erneuten globalen Umgebungsschritt zusammen. Falls die linke Komponente erst im dritten Schritt aufgeführt wird, gilt die mit Linie 3 markierte Beziehung, usw.

Beweis
Um also den Widerspruch zwischen der lokalen und der globalen Formel zu zeigen, muss die linke Komponente einen Schritt machen, damit sich beide S auf den gleichen Zustand beziehen. Dazu wird ein weiterer symbolischer Schritt ausgeführt. Wie bei dem Lifting-Lemma kann dabei angenommen werden, dass die Formel $Rto(n+1, S', S'')$ erfüllt ist (ansonsten „terminiert“ die Sustain-Formel im Sukzedens und der Beweis kann einfach geschlossen werden). Daraus ergeben sich drei Fälle:

- Die linke Komponente macht einen Schritt, d.h. es gilt die mit Linie 1 markierte Beziehung. Da nun die lokale und die globale Belegung von S übereinstimmen, kann der Widerspruch zwischen den beiden Formeln gezeigt werden.

- Die rechte Komponente macht einen Schritt und deren Rely-Bedingung ist nicht verletzt. Mit der Transitivität der Rely-Bedingungen kann gezeigt werden, dass global immer noch $R(n+1, s_0, S)$ gilt (dazu muss mit der Guarantee der rechten Komponente und der globalen Rely $Rto(n+1, S', S'')$ gezeigt werden, dass im letzten System- und Umgebungsschritt

die Annahme R_{n+1} erfüllt war). Damit kann diese Prämisse durch die Induktionshypothese geschlossen werden.

- Die rechte Komponente vollzieht einen Schritt, in dem ihre Rely-Bedingung ebenfalls verletzt wird (d. h., lokal gilt $\neg\ Rto(n, s_1, S)$). Dabei kann gezeigt werden, dass global sowohl $R(n+1, s_0, S)$ als auch noch $Rto(n, s_1, S)$ gilt. Durch einen weiteren symbolischen Ausführungsschritt kann gezeigt werden, dass dies zu einem Widerspruch führt, da bei einem Schritt das globale S mit einer der beiden lokalen Belegungen von S übereinstimmen muss.

□

4.3.4 Beweis des Modularisierungstheorems für Interleaving

Zum Beweis des Modularisierungstheorems für Interleaving (Satz 3) wird zunächst das folgende Lemma bewiesen:

Lemma 2 *(Modularisierungslemma).* Unter der Voraussetzung des Modularisierungstheorems (Satz 3) gilt für alle n:

$$[SpawnP(n; S)]_S,\ I(S)$$
$$\vdash\ Rto(n, S', S'') \xrightarrow{+}\ Gto(n, S, S')$$

□

Die Prozedur $SpawnP(n; S)$ wird benutzt, um das interleaved-parallele System $P(0; S) \parallel \ldots \parallel P(n; S)$ zu erzeugen. Außer der Variable S werden von diesem System keine anderen Variablen geändert. Zusätzlich wird angenommen, dass die Invariante $I(S)$ am Anfang des Ablaufs erfüllt ist. Gezeigt wird, dass jeder Systemschritt von $SpawnP(n; S)$ die Garantie-Bedingung *eines* der Prozesse 0 bis n erfüllt, solange die Rely-Bedingungen all dieser Prozesse 0 bis n in jedem Umgebungsschritt erfüllt sind.

Beweis
Der Beweis von Lemma 2 wird durch strukturelle Induktion über die Variable n geführt. Der Induktionsanfang besteht nach Expansion der Spawn-Prozedur $SpawnP(n; S)$ aus der folgenden Sequenz:

$$[P(0; S)]_S,\ I(S)$$
$$\vdash\ Rto(0, S', S'') \xrightarrow{+}\ Gto(0, S, S')$$

Diese Sequenz kann einfach bewiesen werden, indem man mit Prämisse 1. aus Satz 3 die Rely-/Guarantee-Bedingung $R(0, S', S'') \xrightarrow{+} G(0, S, S')$ für die Prozedur $P(0; S)$ einsetzt. Der Sukzedens dieser Sequenz lässt sich ebenfalls durch Einsetzen der Definitionen von Rto und Gto zu $R(0, S', S'') \xrightarrow{+} G(0, S, S')$ umschreiben, wodurch der Induktionsanfang bewiesen ist.

Im Induktionsschritt ist die folgende Sequenz zu beweisen:

$$[SpawnP(n+1; S)]_S,\ I(S),\ \mathbf{IH}$$
$$\vdash\ Rto(n+1, S', S'') \xrightarrow{+}\ Gto(n+1, S, S')$$

wobei **IH** die Induktionshypothese ist:

$$\mathbf{IH} \;:\equiv\; \forall\, S.([SpawnP(n;S)]_S \wedge I(S) \rightarrow Rto(n,S',S'') \stackrel{+}{\rightarrow} Gto(n,S,S'))$$

Zunächst wird der Spawn-Prozess $SpawnP(n+1;S)$ zu

$$P(n+1;S) \parallel SpawnP(n;S)$$

expandiert. Hier kann der rechte Prozess mit Hilfe der Induktionshypothese **IH** zu

$$I(S) \rightarrow Rto(n,S',S'') \stackrel{+}{\rightarrow} Gto(n,S,S')$$

umgeschrieben werden. Ebenso kann der linke Prozess mit der Prämisse 1. aus Satz 3 zu

$$I(S) \rightarrow R(n+1,S',S'') \stackrel{+}{\rightarrow} G(n+1,S,S')$$

umgeschrieben werden. Insgesamt ist die noch zu beweisende Sequenz dann:

$$\begin{aligned} [\quad & I(S) \rightarrow R(n+1,S',S'') \stackrel{+}{\rightarrow} G(n+1,S,S') \\ \parallel\ & I(S) \rightarrow Rto(n,S',S'') \stackrel{+}{\rightarrow} Gto(n,S,S')\ ]_S, \\ & I(S) \\ \vdash\ & Rto(n+1,S',S'') \stackrel{+}{\rightarrow} Gto(n+1,S,S') \end{aligned} \tag{4.9}$$

Die beiden Implikationen unter dem Interleaving-Operator können durch jeweils eine Fallunterscheidung, ob die Invariante im ersten Schritt der linken (bzw. rechten) Komponente gilt, eliminiert werden. Dazu wird zuerst mit Hilfe der $\stackrel{+}{\rightarrow}$-Formel im Sukzedens eine, in Kapitel 2.5.1 beschriebene, temporallogische Induktionshypothese **IH2** generiert. Die Fallunterscheidung führt dann zu den folgenden vier Fällen:

- Die Invariante ist im ersten lokalen Zustand bei beiden Komponenten verletzt. Da die Invariante aber im ersten globalen Zustand gilt (siehe Sequenz (4.9)), führt dies zu einem Widerspruch. Dieser kann durch einen einfachen symbolischen Ausführungsschritt gezeigt werden.

- Nur bei der linken Komponente ist die Invariante im ersten lokalen Zustand verletzt. In diesem Fall ist die folgende Sequenz zu beweisen:

  $$\begin{aligned} & [\neg\, I(S) \;\parallel\; Rto(n,S',S'') \stackrel{+}{\rightarrow} Gto(n,S,S')]_S,\; I(S), \mathbf{IH2} \\ \vdash\ & Rto(n+1,S',S'') \stackrel{+}{\rightarrow} Gto(n+1,S,S') \end{aligned}$$

  In einem symbolischen Ausführungsschritt vollzieht nun entweder die linke Komponente einen Schritt, wobei dieser Fall wieder zu einem Widerspruch führt, da die Invariante global gilt, oder die rechte Komponente macht den nächsten Schritt. In diesem Fall kann der Ast durch Anwendung der Induktionshypothese geschlossen werden, da die Sequenz unverändert bleibt.

- Die Invariante ist im ersten lokalen Zustand der rechten Komponente verletzt. Der Beweis dieses Falls verläuft analog zum vorhergehenden Fall.

- In beiden Komponenten gilt die Invariante im ersten Zustand:

$$\begin{array}{l} [\quad R(n+1,S',S'') \stackrel{+}{\rightarrow} G(n+1,S,S') \\ \quad \| \; Rto(n,S',S'') \stackrel{+}{\rightarrow} Gto(n,S,S') \;]_S, \\ I(S) \\ \vdash \; Rto(n+1,S',S'') \stackrel{+}{\rightarrow} Gto(n+1,S,S') \end{array}$$

  Diese Sequenz wurde im Lifting-Lemma (Lemma 2) bewiesen.

□

4.3.5 Beweis des Modularisierungstheorems

Die eigentliche Aussage von Satz 3 ist in KIV durch die folgende Sequenz modelliert:

$$[SpawnP(n;S)]_S,\ I(S) \ \vdash \ R(S',S'') \stackrel{+}{\rightarrow} G(S,S') \tag{4.10}$$

Durch Benutzung von Lemma 2 kann der Beweis auf den Beweis der folgenden Sequenz reduziert werden:

$$Rto(n,S',S'') \stackrel{+}{\rightarrow} Gto(n,S,S'),\ I(S) \ \vdash \ R(S',S'') \stackrel{+}{\rightarrow} G(S,S')$$

Diese Sequenz kann mit Hilfe der Prämissen 2. und 4. durch einen symbolischen Ausführungsschritt und Induktion bewiesen werden.

4.4 Folgerung aus dem Modularisierungstheorem

Oft will man keine globale Garantie wie im Modularisierungstheorem zeigen, sondern nur nachweisen, dass eine Eigenschaft invariant ist. Im folgenden Semaphor- und ProChaCon-Beispiel am Ende dieses Kapitels wird eine derartige invariante Eigenschaft bewiesen. Für diesen Fall kann Satz 3 angepasst werden:

Korollar 1.
Seien P_i, S, I, R, G_i und R_i wie in Theorem 3. Wenn für alle $i = 1, \ldots, n$ gilt:

1. $P_i(S), I(S) \vdash R_i(S',S'') \stackrel{+}{\rightarrow} G_i(S,S')$
2. $G_i(S,S') \wedge I(S) \rightarrow \bigwedge_{j\in\{1..n\} \wedge j\neq i} R_j(S,S')$
3. $R_i(S,S') \wedge R_i(S',S'') \wedge I(S') \wedge I(S') \rightarrow R_i(S,S'')$
4. $R(S,S') \wedge I(S) \rightarrow \bigwedge_{j\in\{1..n\}} R_j(S,S')$
5. $G_i(S,S') \wedge I(S) \rightarrow I(S')$
6. $R(S',S'') \wedge I(S') \rightarrow I(S'')$

dann gilt auch:

$$P_1(S) \parallel \ldots \parallel P_n(S),\ I(S),\ \Box\ R(S', S'') \ \vdash\ \Box\ (I(S) \wedge I(S'))$$

Die Beweisverpflichtungen in diesem Korollar sind bis auf 2 identisch mit denen aus Theorem 3. In der zweiten Beweisverpflichtung muss nur noch gezeigt werden, dass die lokalen Garantien die lokalen Umgebungsannahmen der anderen Komponenten erfüllen. Dass die lokalen Garantien auch die globalen Garantien erfüllen, entfällt hier. Als Folgerung wird jetzt gezeigt, dass die Invariante I in allen Zuständen und Zwischenzuständen gilt. Dazu muss jedoch vorausgesetzt werden, dass die globale *Rely*-Bedingung in allen globalen Umgebungsschritten gilt.

Beweis
Zum Beweis von Korrolar 1 wird das Modularisierungstheorem aus Satz 3 benutzt. Dabei wird für die globale Garantie G die Invariante I eingesetzt:

$$G(S, S') \ :\Leftrightarrow\ I(S) \wedge I(S') \tag{4.11}$$

Um Satz 3 anzuwenden, muss zunächst gezeigt werden, dass die Beweisverpflichtung 2 aus Satz 3 aus den Beweisverpflichtungen 2 und 5 des Korollars 1 folgt. Dies kann unter Verwendung von Formel (4.11) einfach bewiesen werden. Damit kann Satz 3 angewandt und der Beweis von Korollar 1 auf die folgende Sequenz reduziert werden:

$$\Box\ R(S', S''),\ R(S', S'') \xrightarrow{+} (I(S) \wedge I(S')) \vdash \Box\ (I(S) \wedge I(S'))$$

Der Beweis dieser Sequenz ist einfach und wird durch einen symbolisches Ausführungsschritt und Induktionsanwendung geführt. □

4.5 Beispiel Semaphor

Um die Anwendung des oben vorgestellten Theorems zu veranschaulichen, wird das Semaphor-Beispiel aus Kapitel 3 benutzt. Der Semaphor-Prozess und die lokalen Rely- und Guarantee-Bedingungen können genau wie in Kapitel 3 beschrieben spezifiziert werden. Zusätzlich ist jedoch noch eine Invariante für den Beweis der Rely-Guarantee Eigenschaft nötig: Jede Komponente garantiert, dass sie den kritischen Bereich nicht betritt, wenn die andere Komponente schon im kritischen Bereich ist. Um dies zu beweisen, ist als Voraussetzung das Wissen über den Zustand der beiden Programmzähler und der Semaphore wichtig. Zusätzlich wird die Gesamteigenschaft, dass beide Prozesse sich niemals zur gleichen Zeit im kritischen Bereich aufhalten, zur Invariante hinzugefügt. Zu diesem Zweck kann das Korollar 1 benutzt werden, um diese Gesamteigenschaft zu zeigen.

Daraus ergibt sich die folgende Invariante:

$$\begin{aligned} I(S) \ :\Leftrightarrow\quad & (PC_1 = \mathit{critical} \vee PC_2 = \mathit{critical} \rightarrow \neg\ \mathit{Sem}) \\ & \wedge\ \neg\ (PC_1 = \mathit{critical} \wedge PC_2 = \mathit{critical}) \end{aligned}$$

Das erste Konjunkt formalisiert, dass die Semaphore nicht frei ist, wenn sich eine der beiden Komponenten im kritischen Bereich befindet. Das zweiten Konjunkt enthält die mittels Korrolar 1 zu zeigende Gesamteigenschaft: Beide Komponenten befinden sich niemals gleichzeitig im kritischen Bereich.

Mit diesen Eigenschaften ist der Beweis der prädikatenlogischen Beweisverpflichtungen 2 – 6 des Modularisierungstheorems sehr einfach, da sie direkt aus der Spezifikation folgen. Vom KIV-System können sie alle vollautomatisch durch den Simplifier bewiesen werden.

Für die temporallogische Beweisverpflichtung 1 ist etwas mehr Beweisaufwand notwendig. Hier wird nur der Beweis für die erste Komponente beschrieben, da der Beweis für die zweite Komponente, bis auf die symmetrische Vertauschung der beiden Programmzähler identisch ist. Im wesentlichen besteht der Beweis aus dem schrittweisen symbolischen Ausführen der Semaphor-Komponente bis die Programmschleife durchlaufen ist und dem Anwenden von Induktion. Lediglich für die abstrakt spezifizierten kritischen und nicht-kritischen Abschnitte ist eine zusätzliche Generalisierung notwendig.

Zuerst werden alle Rely- und Guarantee-Eigenschaften durch ihre Spezifikation expandiert. Danach wird die **unless**-Formel im Sukzedens benutzt, um eine Induktionshypothese zu generieren (siehe dazu Kapitel 2.5.1). Da die Invariante nicht verbietet, dass die erste Komponente schon zu Beginn im kritischen Bereich ist (d. h. $PC_1 = \mathit{critical}$, in diesem Fall fordert die Invariante nur, dass $PC_2 = \mathit{noncritical}$ und $\neg\ \mathit{Sem}$), wird eine Fallunterscheidung gemacht, ob $PC_1 = \mathit{critical}$ ist oder nicht.

Im ersten Fall kann der Beweis nach zwei symbolischen Ausführungsschritten geschlossen werden, da die Komponente an der **await**-Bedingung in Zeile 3 blockiert wird. Auch die Umgebung kann hier nur Stotterschritte durchführen, da die andere Komponente sich im nicht-kritischen Bereich befindet, die Semaphore aber blockiert ist.

Im anderen Fall kann die Invariante zu

$$PC_1 = \mathit{noncritical} \wedge (PC_2 = \mathit{critical} \rightarrow \neg\ \mathit{Sem}) \tag{4.12}$$

vereinfacht werden. Diese Eigenschaft wird im Folgenden als Beweisinvariante für alle Zustände, in denen die erste Komponente nicht im kritischen Bereich ist, benutzt. Der erste Schritt expandiert die **while**-Schleife. Die Umgebung kann hier drei mögliche Schritte machen: Den kritischen Bereich betreten oder verlassen oder einen Stotterschritt ausführen. Alle drei Fälle können wieder zu der Beweisinvariante (4.12) generalisiert werden und durch Anwenden der Beweislemma-Regel zu einer Prämisse vereinigt werden. Im nächsten Schritt versucht die erste Komponente den kritischen Bereich zu betreten. Wenn dies fehlschlägt, kann der Ast durch Anwendung der Induktion auf den vorherigen Knoten geschlossen werden. Um zu zeigen, dass die Prozedur *Crit* in Zeile 5 nicht die Rely-Guarantee Eigenschaft verletzt, kann diese Prozedur zu

$$\Box\ (PC_1 = PC_1' \wedge PC_2 = PC_2' \wedge \mathit{Sem} = \mathit{Sem}') \tag{4.13}$$

generalisiert werden. Das heißt, PC_1, PC_2 und *Sem* werden von *Crit* nicht verändert. Ein erneuter symbolischer Ausführungsschritt führt zu einer Fallunterscheidung: Entweder es ist der letzte Schritt der Formel (4.13) erreicht und

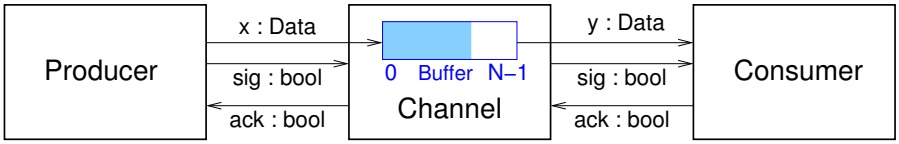


Abbildung 4.4: Das Producer-Channel-Consumer (ProChaCon) Beispiel

durch Ausführen der Zeile 6 des Programms wird der kritische Bereich verlassen, oder die Formel (4.13) macht einen Stotterschritt. Der zweite Fall kann sofort durch Anwendung der Induktion auf den vorherigen Knoten geschlossen werden. Im ersten Fall führen die verschiedenen möglichen Umgebungsschritte wieder zu unterschiedlichen Zuständen. Diese können aber durch Generalisierung zu (4.12) wieder vereinheitlicht werden. Der Prozeduraufruf von *Noncrit* in Zeile 7 kann genau wie die Prozedur *Crit* zu (4.13) generalisiert werden. Die Ausführung dieser Formel ergibt die gleiche Fallunterscheidung wie oben: Der eine Fall führt zu einem Stotterschritt, der sofort durch Induktionsanwendung auf den vorherigen Zustand geschlossen werden kann. Im anderen Fall ist die abstrahierte *Noncrit*-Prozedur abgearbeitet und die Komponente steht wieder am Anfang der Schleife. Dieser Ast kann durch Induktionsanwendung auf den ersten Zustand geschlossen werden.

4.6 Fallstudie: Producer-Channel-Consumer System

Als Anwendungsbeispiel für das oben vorgestellte Modularisierungstheorem soll in diesem Abschnitt das *Producer-Channel-Consumer*-Beispiel (kurz *ProChaCon*) vorgestellt werden. Dieses Beispiel ist eine Standardfallstudie, die zum Beispiel schon in [AL95] oder [BP99] verwendet wurde.

4.6.1 Überblick

Wie schon der Name impliziert, besteht das *ProChaCon*-Beispiel aus drei Komponenten: Daten werden über einen Kanal von einer *Producer*-Komponente an eine *Consumer*-Komponente übergeben. Abbildung 4.4 zeigt die drei Komponenten. Die *Producer*-Komponente generiert neue Werte, die über ein *Handshake*-Protokoll an die *Channel*-Komponente übergeben werden. Diese puffert die Daten in einer Schlange und übergibt sie dann weiter an die *Consumer*-Komponente.

Die Daten werden zwischen jeweils zwei Komponenten über ein klassisches zwei-Wege *Handshake*-Protokoll ausgetauscht. Neben einem Datenkanal signalisieren zwei boolesche Signale *sig* (für *Signal*) und *ack* (für *Acknowledgement* – Quittierung), wer von den beiden Komponenten auf den Datenkanal zugreifen darf: Sind *sig* und *ack* ungleich, darf der Sender einen neuen Wert in den Datenkanal schreiben und dies danach durch das Umschalten von *sig* signalisieren. Bei gleichem *sig* und *ack* Signal kann der Empfänger lesend auf den Datenkanal zugreifen. Durch das Umschalten des *ack*-Signals wird dem Sender wieder signalisiert, dass die Leseoperation abgeschlossen ist.

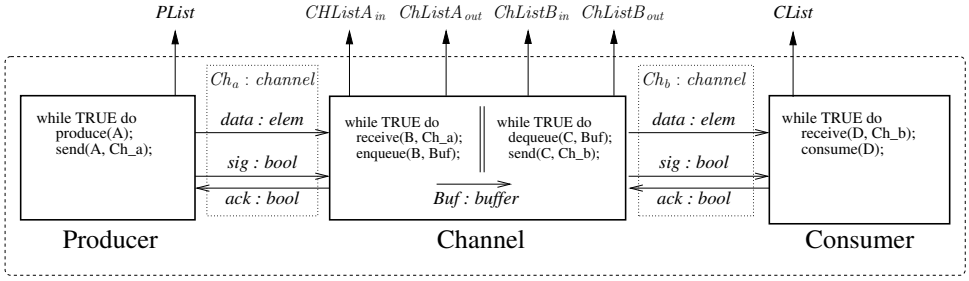


Abbildung 4.5: Struktur der ProChaCon-Spezifikation

Ziel der Verifikation soll es sein, zu zeigen, dass alle Werte, die vom *Producer* generiert werden, auch beim *Consumer* ankommen. Auch soll die Reihenfolge der Daten erhalten bleiben, und es dürfen auch keine Daten dupliziert werden, bzw. keine neuen, nicht vom *Producer* generierte Daten hinzugefügt werden.

4.6.2 Spezifikation

Insgesamt ist die Spezifikation des ProChaCon-Systems in vier Prozesse aufgeteilt. Neben den beiden oben beschriebenen Komponenten *Producer* und *Consumer* wird die *Channel*-Komponente durch zwei Prozesse spezifiziert. Der erste Prozess nimmt die Daten durch das Handshake-Protoll vom *Producer* entgegen und hängt diese an die Queue an, während der zweite Prozess die Daten aus der Queue nimmt und über das Handshake-Protokoll an den Consumer weiter gibt. Wie diese vier Prozesse zusammenhängen, ist in Abbildung 4.5 dargestellt.

Der Handshake-Kanal wird durch den *channel* spezifiziert. Für den Datenwert *data* wird dabei die abstrakte Spezifikation *elem* benutzt. Die beiden Handshake-Kanäle werden durch zwei Variablen vom Typ *channel* dargestellt. Jeder Prozess, der auf einen dieser Kanäle zugreift, bekommt diesen als *Reference*-Parameter. Mit den Postfix-Funktionen *.data*, *.sig* und *.ack* kann auf die Steuerbits *sig* und *ack* des Kanals zugegriffen werden. Zusätzlich werden die Funktionen

$$\begin{aligned} setData &: channel \times elem \mapsto channel \\ toggleSig &: channel \mapsto channel \\ toggleAck &: channel \mapsto channel \end{aligned}$$

benutzt. Die Funktion *setData* aktualisiert den Datenwert des Kanals. Die beiden Funktionen *toggleSig* und *toggleAck* kippen das *sig* bzw. *ack* Steuerbit des Kanals, um der jeweils anderen Komponente zu signalisieren, dass sie an der Reihe ist.

Der hier benutzte Verifikationsansatz versucht, den internen Zustand beziehungsweise das interne Verhalten der Komponenten weitgehend zu abstrahieren und die Rely- und Guarantee-Bedingungen über das Ein- und Ausgabeverhalten der Komponenten zu formulieren. Dazu ist es jedoch nötig, alle eingelesenen und ausgegebenen Werte der Komponenten mitzuprotokollieren. Zu diesem Zweck werden dem *Producer* und dem *Consumer* jeweils eine Liste (*PList* bzw. *CList*)

```
1  Prod(; Ch, PList) {
2      while true do {
3          choose A with true in {
4              await Ch.sig = Ch.ack;
5              Ch := setData(Ch, A);
6              Ch := toggleSig(Ch), PList := PList + A
7          }
8      }
9  }
```

Abbildung 4.6: Deklaration des Producers in KIV

```
1  Chan_enq(; Ch, Buffer, ChListA_in, ChListA_out){
2      let A in {
3          while true do {
4              await Ch.sig ≠ Ch.ack;
5              A := Ch.data;
6              Ch := toggleAck(Ch), ChListA_in := ChListA_in + A;
7              Buffer := Buffer + A, ChListA_out := ChListA_out + A;
8          }
9      }
10 }
```

Abbildung 4.7: Deklaration der Channel-Enqueue Komponente in KIV

und den beiden *Channel*-Prozessen je zwei Listen ($ChListA_{in}$, $ChListA_{out}$, $ChListB_{in}$ und $ChListB_{out}$) als *Reference*-Parameter übergeben. Jeder eingelesene oder ausgegebene Wert wird an diese Listen angehängt. Über diese Listen werden dann die Rely- und Guarantee-Bedingungen der Komponenten formalisiert. Diese sind im folgenden Kapitel 4.6.3 beschrieben.

Abbildung 4.6 zeigt die KIV-Spezifikation der Producer-Komponente. In Zeile 3 wird ein neuer Wert generiert, der dann an die Channel-Komponente übergeben wird. Danach wartet der Algorithmus in Zeile 4, bis er auf den Handshake-Kanal zugreifen kann. Dies wird durch $Ch.sig = Ch.ack$ signalisiert. In Zeile 5 wird dann der Datenwert in die Handshake-Variable geschrieben. Schließlich wird in Zeile 6 durch das Umschalten der *sig*-Variable der Channel-Komponente signalisiert, dass sie nun auf den Datenkanal der Handshake-Variable zugreifen kann. Dabei wird gleichzeitig durch eine parallele Zuweisung der ausgegebene Wert an die Liste *PList* angehängt.

Die Spezifikation der Channel-Komponente ist in zwei parallele Teilkomponenten aufgeteilt. Die erste Komponente $Chan_{enq}$ nimmt Werte vom Producer entgegen und schreibt sie in den Puffer. Die andere Komponente liest Werte aus dem Puffer (falls dieser nicht leer ist) und übergibt diese an den *Consumer*.

Die Spezifikation von $Chan_{enq}$ ist in Abbildung 4.7 dargestellt. In den Zeilen 4 – 6 wird ein Wert über das Handshake-Protokoll empfangen. Wiederum wartet der Algorithmus in Zeile 4 zuerst, bis er an der Reihe ist (d. h. $Ch.sig \neq Ch.ack$). Danach wird der Wert gelesen (Zeile 5) und durch das Umschalten der *ack* Va-

```
1   Chan_deq(; Ch, Buffer, ChListB_in, ChListB_out) {
2     let A in {
3       while true do {
4         await Buffer ≠ ∅;
5         A := Buffer.first;
6         Buffer := Buffer.rest, ChListB_in := ChListB_in + A;
7         await Ch.sig = Ch.ack;
8         Ch := setData(Ch, A);
9         Ch := toggleSig(Ch), ChListB_out := ChListB_out + A
10      }
11    }
12  }
```

Abbildung 4.8: Deklaration der Channel-Dequeue-Komponente in KIV

```
1   Cons(; Ch, CList) {
2     let A in {
3       while true do {
4         await Ch.sig ≠ Ch.ack;
5         A := Ch.data;
6         Ch := toggleAck(Ch), CList := CList + A
7       }
8     }
9   }
```

Abbildung 4.9: Deklaration der Consumer-Komponente in KIV

riable dem Sender signalisiert, dass der Empfang des Datenwertes abgeschlossen ist (Zeile 6). Danach wird der neue Wert an den Puffer angehängt (Zeile 7). In den beiden letzten Schritten wird zusätzlich jeweils die Liste $ChListA_{in}$ bzw. $ChListA_{out}$ entsprechend um den eingelesenen oder ausgegebenen Wert aktualisiert.

Abbildung 4.8 zeigt die Spezifikation von $Chan_{deq}$. In den Zeilen 3 – 5 werden neue Werte aus dem Puffer genommen. Dazu wartet die Komponente in Zeile 3 bis der Puffer Daten enthält. Im nächsten Schritt wird der erste Wert gelesen (Zeile 4) und danach aus dem Puffer genommen (Zeile 5). Dabei wird der entnommene Wert an die Liste $ChListB_{in}$ angehängt. Das Schreiben des neuen Wertes in den Handshake-Kanal zwischen der Channel- und der Consumer- Komponente geschieht in den Zeilen 6 – 8. Dies ist identisch mit dem Handshake-Ablauf der Producer-Komponente.

Schließlich liest die Consumer-Komponente *Cons* den Wert aus dem Handshake Kanal. Die Spezifikation ist in Abbildung 4.9 dargestellt. Die Zeilen 3 – 5 sind identisch zu den Zeilen 3 – 5 von $Chan_{enq}$.

4.6.3 Rely-Guarantee Eigenschaften

Da viele der Umgebungsannahmen bzw. Garantien als Aussage enthalten, dass Variablen aus einer Variablenmenge L nicht geändert, bzw. nur die Variablen aus L geändert werden, gelten die folgende Abkürzungen:

$$Unchg_{env}(L) := \bigwedge_{V \in L} V' = V''$$

$$Unchg_{sys}(L) := \bigwedge_{V \in L} V = V'$$

Als Systemvariablen werden für den linken bzw. rechten Handshake-Kanal die globalen Variablen Ch_a und Ch_b benutzt. Für den Datenpuffer wird die globale Variable Buf benutzt. Für die History-Listen der Producer-, Consumer- und Channel-Komponenten werden die Variablen $PList$, $CList$, $ChListA_{in}$, $ChListA_{out}$, $ChListB_{in}$ und $ChListB_{out}$ benutzt. Bei der Instanzierung des Modularisierungstheorems wird die generische Variable S durch einen Produktdatentyp aus diesen Variablen ersetzt. Als abkürzende Schreibweise wird $\underline{Vars}$ für alle Systemvariablen benutzt:

$$\begin{aligned}\underline{Vars} := \langle & Ch_a, Ch_b, Buf, PList, ChListA_{in}, ChListA_{out}, \\ & ChListB_{in}, ChListB_{out}, CList \rangle\end{aligned}$$

Als Gesamteigenschaft soll gezeigt werden, dass alle vom Producer gesendeten Werte auch vom Consumer in der gleichen Reihenfolge empfangen werden. Dazu werden History-Listen benutzt, welche die vom Producer gesendeten und die vom Consumer empfangenen Werte mitprotokollieren. Als zu verifizierende Gesamteigenschaft wird dann gezeigt, dass die Liste der vom Consumer empfangenen Werte in jedem Schritt ein Präfix der vom Producer gesendeten Werte ist. Dies kann durch die folgende Formel ausgedrückt werden

$$\Box\ CList \sqsubseteq PList \wedge CList' \sqsubseteq PList' \tag{4.14}$$

wobei $\sqsubseteq$ der Präfix-Operator ist. Zum Nachweis dieser Eigenschaft wird eine Invariante gebildet, aus der sich diese Eigenschaft folgern lässt. Zum Beweis von (4.14) wird dann Korollar 1 benutzt, um nachzuweisen, dass die Invariante immer erfüllt ist.

Als globale Umgebungsannahme wird dabei angenommen, dass keine Variable des ProChaCon-Systems von der Umgebung verändert wird:

$$R(\underline{Vars}', \underline{Vars}'') :\Leftrightarrow Unchg_{env}(\{\underline{Vars}\})$$

Damit ergibt sich die folgende Beweisverpflichtung für das Gesamtsystem:

$$\begin{aligned}& [SpawnP(3; \underline{Vars})]_{\underline{Vars}}, I(\underline{Vars}), \Box\ R(\underline{Vars}', \underline{Vars}'') \\ \vdash\ & \Box\ I(\underline{Vars}) \wedge I(\underline{Vars}')\end{aligned}$$

Die Prozedur $SpawnP$ erzeugt das ProChaCon-System. Dabei werden die von $SpawnP$ erzeugten Prozesse P so definiert, dass P mit der Prozessnummer 0

den Producer-Prozess erzeugt. P mit den Prozessnummern 1 und 2 erzeugen die linke, bzw. rechte Channel-Komponente und P mit der Prozessnummer 3 erzeugt schließlich die Consumer-Komponente. Zum Beweis dieser Sequenz wird Korollar 1 verwendet.

Im Folgenden werden die Teilgarantien für den Handshake-Kanal, die eine Komponente erfüllen muss, angegeben. Diese Teilgarantien beschreiben, wie und wann eine Komponente auf den Handshake-Kanal zugreifen darf und dass die History-Liste entsprechend aktualisiert wird. Daher werden diese Teilgarantien über eine Channel-Variable *Ch* und eine Liste *List* spezifiziert. Mit diesen Teilgarantien können die Guarantee-Bedingungen für die Producer- und die Consumer-Komponente einfach spezifiziert werden. Diese Teilgarantien werden aber auch in den Guarantee-Bedingungen der beiden Channel-Prozesse benutzt.

Die Garantien, die ein Sender-Prozess dem Handshake-Kanal zusichern muss, können wie folgt spezifiziert werden:

$$
\begin{aligned}
&\mathit{Handshake}_{\mathit{send}}(\mathit{Ch}, \mathit{List}, \mathit{Ch}', \mathit{List}') \leftrightarrow \\
&\qquad (\mathit{Ch.sig} \neq \mathit{Ch.ack} \rightarrow (\mathit{Ch}' = \mathit{Ch} \wedge \mathit{List}' = \mathit{List})) \\
&\wedge\ ((\mathit{Ch.sig} = \mathit{Ch.ack} \wedge \mathit{Ch}'.\mathit{sig} = \mathit{Ch}'.\mathit{ack}) \rightarrow \mathit{List}' = \mathit{List}) \\
&\wedge\ ((\mathit{Ch.sig} = \mathit{Ch.ack} \wedge \mathit{Ch}'.\mathit{sig} \neq \mathit{Ch}'.\mathit{ack}) \rightarrow \mathit{List}' = \mathit{List} + \mathit{Ch.data})
\end{aligned}
$$

Wenn die Signalbits *sig* und *ack* ungleich sind, wartet der Sender darauf, dass der Empfänger den Datenwert aus dem Kanal liest. In diesem Fall darf weder der Channel noch die History-Liste verändert werden. Falls *sig* und *ack* gleich sind, darf der Channel verändert werden. Hier muss unterschieden werden, ob im aktuellen Schritt der Handshake-Kanal für den Empfänger freigegeben wird oder nicht. Die History-Liste darf erst aktualisiert werden, wenn der Channel wieder für den Empfänger freigegeben wird. In diesem Fall muss der Wert im Datenfeld des Handshake-Kanals an die History-Liste angehängt werden.

Eine Empfänger-Komponente eines Handshake-Kanals muss die folgenden Eigenschaften garantieren:

$$
\begin{aligned}
&\mathit{Handshake}_{\mathit{rec}}(\mathit{Ch}, \mathit{List}, \mathit{Ch}', \mathit{List}') \leftrightarrow \\
&\qquad \mathit{Ch.data} = \mathit{Ch}'.\mathit{data} \\
&\wedge\ (\mathit{Ch.sig} = \mathit{Ch.ack} \rightarrow (\mathit{Ch}' = \mathit{Ch} \wedge \mathit{List}' = \mathit{List})) \\
&\wedge\ ((\mathit{Ch.sig} \neq \mathit{Ch.ack} \wedge \mathit{Ch}'.\mathit{sig} \neq \mathit{Ch}'.\mathit{ack}) \rightarrow \mathit{List}' = \mathit{List}) \\
&\wedge\ ((\mathit{Ch.sig} \neq \mathit{Ch.ack} \wedge \mathit{Ch}'.\mathit{sig} = \mathit{Ch}'.\mathit{ack}) \rightarrow \mathit{List}' = \mathit{List} + \mathit{Ch.data})
\end{aligned}
$$

Zum einen muss eine Empfänger-Komponente garantieren, dass sie das Datenfeld des Handshake-Kanals niemals ändert. Zusätzlich darf weder der Channel noch die History-Liste vom Empfänger verändert werden, wenn gerade die Sender-Komponente an der Reihe ist (d. h., wenn $\mathit{sig} = \mathit{ack}$). Wie bei den Sendergarantien darf die History-Liste erst verändert werden, wenn die Kontrolle wieder von der Empfänger- an die Sender-Komponente übergeben wird. Auch in diesem Fall muss der richtige Datenwert an die Liste angehängt werden.

Mit diesen Prädikaten für den Handshake-Kanal können die Rely- und Guarantee-Bedingungen für die Producer- und Consumer-Komponente definiert werden.

Für die Producer-Komponente ergibt sich als Garantiebedingung:

$$\begin{aligned} G_{Prod}(\underline{Vars}, \underline{Vars}') \leftrightarrow \quad & Unchg_{sys}(\{\underline{Vars}\} \setminus \{Ch_a, PList\}) \\ & \wedge\ Handshake_{send}(Ch_a, PList) \end{aligned}$$

Außer der Channel-Variable Ch_a und der History-Variable $PList$ werden von der Producer-Komponente keine Variablen geändert. Zusätzlich werden vom Producer die Garantien für den Sender auf den ersten Handshake-Kanal eingehalten. Als Umgebungsannahme für die Producer-Komponente ergibt sich:

$$\begin{aligned} R_{Prod}(\underline{Vars}', \underline{Vars}'') \leftrightarrow \quad & Unchg_{env}(\{PList\}) \\ & \wedge\ (Ch'_a.sig = Ch'_a.ack \rightarrow Ch''_a = Ch'_a) \end{aligned}$$

Die Umgebung garantiert dem Producer, dass die Variable $PList$ niemals von ihr geändert wird. Zusätzlich bleibt die Handshake-Variable Ch_a immer dann von der Umgebung unverändert, wenn die Producer-Komponente an der Reihe ist, das heißt, wenn sig gleich ack.

Die Rely- und Guarantee-Bedingungen für den Consumer sind sehr ähnlich. Die Garantie-Eigenschaft hat folgende Form:

$$\begin{aligned} G_{Cons}(\underline{Vars}, \underline{Vars}') \leftrightarrow \quad & Unchg_{sys}(\{\underline{Vars}\} \setminus \{Ch_b, CList\}) \\ & \wedge\ Handshake_{rec}(Ch_b, CList) \end{aligned}$$

Außer der Channel-Variable Ch_b und der History-Variable $CList$ werden vom Consumer keine anderen Variablen geändert. Zusätzlich erfüllt die Consumer-Komponente alle Garantien aus $Handshake_{rec}$ über den rechten Handshake-Kanal Ch_b. Die Umgebungsannahme für den Consumer ist wie folgt definiert:

$$\begin{aligned} R_{Cons}(\underline{Vars}', \underline{Vars}'') \leftrightarrow \quad & Unchg_{env}(\{CList\}) \\ & \wedge\ (Ch'_b.sig = Ch'_b.ack \rightarrow Ch''_b = Ch'_b) \end{aligned}$$

Wie beim Producer darf die History-Variable $CList$ nicht geändert werden und die Handshake-Variable Ch_b bleibt unverändert, solange der Consumer an der Reihe ist (d. h. sig ungleich ack).

Zusätzlich zu den Garantiebedingungen muss für jeden Handshake-Kanal in der Invariante der Zusammenhang der beiden History-Listen des Senders und des Empfängers gespeichert werden. Will man für einen Handshake-Kanal Ch zeigen, dass die History-Liste des Empfängers $List_{send}$ ein Präfix der History-Liste des Empfängers $List_{rec}$, so gilt:

$$\begin{aligned} & Handshake_{inv}(Ch, List_{send}, List_{rec}) \leftrightarrow \\ & \qquad (Ch.sig = Ch.ack \rightarrow List_{send} = List_{rec}) \\ & \wedge\ (Ch.sig \neq Ch.ack \rightarrow List_{send} = List_{rec} + Ch.data) \end{aligned}$$

Wenn der Sender an der Reihe ist, müssen alle gesendeten Werte beim Empfänger angekommen sein, das heißt, beide History-Listen müssen gleich sein. Falls der Empfänger an der Reihe ist, müssen bis auf den Wert, der sich gerade im Handshake-Kanal befindet, alle Werte beim Empfänger angekommen sein, das heißt $List_{send} = List_{rec} + Ch.data$.

Für die beiden Channel-Komponenten kann ein ähnliches Schema verwendet werden. Allerdings muss hier zusätzlich zu den Handshake-Garantiebedingungen noch zugesichert werden, dass die beiden Komponenten korrekt auf den Puffer zugreifen und deren History-Listen richtig aktualisiert werden. Für die in den Puffer schreibende Komponente wird dies mit der folgenden Eigenschaft garantiert:

$$\begin{aligned} Buffer_{in}(Buf, List) \leftrightarrow \quad & (Buf' = Buf \wedge List' = List) \\ & \vee \exists\, val.(Buf' = Buf + val \wedge List' = List + val) \end{aligned}$$

Entweder bleiben sowohl der Puffer *Buf* als auch die History-Liste *List* unverändert. Oder die schreibende Komponente fügt einen neuen Wert *val* am Ende des Puffers ein. In diesem Fall muss dieser Wert auch an die History-Liste angehängt werden. Für die aus dem Puffer lesende Komponente sind die Garantien ähnlich:

$$\begin{aligned} Buffer_{out}(Buf, List) \leftrightarrow \quad & (Buf' = Buf \wedge List' = List) \\ & \vee \exists\, val.(val + Buf' = Buf \wedge List' = List + val) \end{aligned}$$

Wieder bleiben im ersten Fall beide Variablen unverändert. Im zweiten Fall muss die History-Liste aktualisiert werden, wenn ein Wert aus dem Puffer entfernt wird, das heißt, falls der Wert *val* vom *Anfang* des Puffers *Buf* entfernt wird, muss dieser an die History-Liste *List* angehängt werden.

Mit diesen Prädikaten kann nun die Garantie für die erste Channel-Komponente $Chan_{enq}$ formuliert werden:

$$\begin{aligned} G_{Chan_{enq}}(\underline{Vars}, \underline{Vars}') \leftrightarrow \\ & Unchg(\{\underline{Vars}\} \setminus \{Ch_a, ChListA_{in}, ChListA_{out}, Buf\}) \\ \wedge\ & Handshake_{rec}(Ch_a, ChListA_{in}) \\ \wedge\ & Buffer_{enq}(Buf, ChListA_{out}) \\ \wedge\ & (ChListA_{out} \sqsubseteq ChListA_{in} \rightarrow ChListA'_{out} \sqsubseteq ChListA'_{in}) \end{aligned}$$

Wie schon beim Consumer garantiert die erste Channel-Komponente, dass die Variablen der anderen Prozesse nicht geändert werden und dass das Handshake-Protokoll korrekt implementiert ist. Zusätzlich garantiert die Komponente noch, dass der Puffer und die zweite History-Liste korrekt aktualisiert werden und die Ausgabe-History-Liste immer ein Präfix der Eingabe-History-Liste bleibt. Die Rely-Bedingung für die linke Channel-Komponente kann fast identisch zur Rely-Bedingung des Consumers definiert werden:

$$\begin{aligned} R_{Chan_{enq}}(\underline{Vars}, \underline{Vars}') \leftrightarrow \quad & Unchg(\{Ch_a, ChListA_{in}, ChListA_{out}\}) \\ & \wedge\ (Ch'_a.sig = Ch'_a.ack \rightarrow Ch''_a = Ch'_a) \end{aligned}$$

Weitere Umgebungsannahmen sind für den Channel nicht notwendig.

Die Rely- und Guarantee-Bedingungen für die zweite Channel-Komponente $Chan_{deq}$ können fast analog zur ersten Komponente definiert werden:

$$\begin{aligned} G_{Chan_{deq}}(\underline{Vars}, \underline{Vars}') \leftrightarrow \\ & Unchg(\{\underline{Vars}\} \setminus \{Ch_b, ChListB_{in}, ChListB_{out}, Buf\}) \\ \wedge\ & Handshake_{send}(Ch_b, ChListB_{out}) \\ \wedge\ & Buffer_{deq}(Buf, ChListB_{in}) \\ \wedge\ & (ChListB_{out} \sqsubseteq ChListB_{in} \rightarrow ChListB'_{out} \sqsubseteq ChListB'_{in}) \end{aligned}$$

$$\begin{aligned} R_{Chan_{deq}}(\underline{Vars}', \underline{Vars}'') \leftrightarrow \quad & Unchg(\{Ch_b, ChListB_{in}, ChListB_{out}\}) \\ \wedge\ & (Ch'_b.sig = Ch'_b.ack \rightarrow Ch''_b = Ch'_b) \\ \wedge\ & Buf' \sqsubseteq Buf'' \end{aligned}$$

Die einzige neue Bedingung in diesen beiden Eigenschaften ist die Formel $Buf' \sqsubseteq Buf''$ am Ende der Umgebungsannahme. Diese Annahme ist notwendig, damit $Chan_{deq}$ in Zeile 6 garantieren kann, dass auch der richtige Wert an die Liste $ChListB_{in}$ angefügt wird.

Die Invariante für den Puffer ist relativ einfach:

$$Buffer_{inf}(Buf, List_{send}, List_{rec}) \quad \leftrightarrow \quad List_{send} = List_{rec} + Buf$$

Alle Werte, die von der schreibenden Komponente in den Puffer geschrieben worden sind ($List_{send}$), sind entweder schon von der lesenden Komponente aus dem Puffer genommen ($List_{rec}$) oder noch im Puffer (Buf). Da zusätzliche Werte immer an die Enden der Listen angehängt werden, kann dieser Zusammenhang durch eine einfache Listenkonkatenation (mit dem überladenen Symbol „+“) ausgedrückt werden.

Damit ergibt sich die Invariante für das gesamte System:

$$\begin{aligned} I(\underline{Vars}) \leftrightarrow \quad & Handshake_{inf}(Ch_a, PList, ChListA_{in}) \\ \wedge\ & ChListA_{out} \sqsubseteq ChListA_{in} \\ \wedge\ & Buffer_{inf}(Buf, ChListA_{out}, ChListB_{in}) \\ \wedge\ & ChListB_{out} \sqsubseteq ChListB_{in} \\ \wedge\ & Handshake_{inf}(Ch_b, ChListB_{out}, CList) \end{aligned}$$

Jeder dieser fünf Konjunkte stellt sicher, dass zu jedem Zeitpunkt eine Teilmengenbeziehung zwischen je zwei aufeinanderfolgenden Listen besteht. Damit gilt für diese Invariante die folgende Implikation:

$$I(\underline{Vars}) \quad \rightarrow \quad CList \sqsubseteq PList$$

Zum Beweis der Gesamteigenschaft (4.14) kann nun das Korollar 1 aus Kapitel 4.4 benutzt werden, um nachzuweisen, dass die Invariante in jedem Grund- und Zwischenzustand erfüllt ist.

4.6.4 Verifikation

Mit den oben geschilderten Rely- und Guarantee-Bedingungen der Komponenten konnten in KIV alle Beweisverpflichtungen des Modularisierungstheorems für Interleaving (Satz 3) formal bewiesen werden. Um einen Eindruck vom erforderlichen Beweisaufwand zu gewinnen, wird im Folgenden der Beweis der temporallogischen Beweisverpflichtung (Beweisverpflichtung 1 des Theorems) für die $Chan_{enq}$ Komponente kurz beschrieben. Die dabei zu beweisende Formel lautet:

$$[Chan_{enq}(1; \underline{Vars})]_{\underline{Vars}},\ I(\underline{Vars})$$
$$\vdash\ R_{Chan_{enq}}(\underline{Vars}', \underline{Vars}'') \overset{+}{\rightarrow} G_{Chan_{enq}}(\underline{Vars}, \underline{Vars}')$$

Der Beweis wird mittels Induktion und symbolischer Ausführung geführt. Im ersten Schritt wird eine Induktionshypothese erzeugt. Dies ist möglich, da der Sustain-Operator auf der rechten Seite äquivalent ist zu einer **unless**-Formel, aus der eine Induktionshypothese generiert werden kann (s. Kapitel 2.5.1). Das Programm startet in der ersten Zeile (die Zeilennummern entsprechen den Nummern aus Abbildung 4.7) vor der **while**-Schleife. Zunächst wird die Schleife mittels eines symbolischen Ausführungsschrittes expandiert. Danach befindet sich das Programm in Zeile 4. Im nächsten symbolischen Ausführungsschritt gibt es eine Fallunterscheidung, ob die **await**-Bedingung zu wahr oder falsch ausgewertet wird. Falls die **await**-Bedingung zu falsch ausgewertet wird, führt das Programm einen Stotterschritt aus (d. h., sowohl das Programm als auch seine Variablen bleiben unverändert) und der Ast kann mithilfe der Induktionshypothese geschlossen werden. Im anderen Fall führt das Programm eine Zuweisung an die Variable A aus und befindet sich danach in Zeile 6. Die übrigen Schritte erfolgen nach genau dem gleichen Schema wie die ersten beiden Schritte, bis sich das Programm wieder in der ersten Zeile befindet. Sobald das Programm wieder den Schleifenanfang erreicht hat, kann der gesamte Beweis durch Anwendung der Induktionshypothese geschlossen werden.

Die Beweise für die prädikatenlogischen Beweisverpflichtungen sind deutlich einfacher. Üblicherweise beginnen sie mit einer Fallunterscheidung, welche die Konjunktion auf der rechten Seite auflöst. Der Rest des Beweises ist dann einfache Simplifikation. Diese Beweise werden vom Simplifier in KIV automatisch geschlossen.

4.6.5 Fazit

Zusammenfassend kann gesagt werden, dass die Wiederverwendbarkeit der Rely- und Guarantee-Bedingungen sehr hoch ist. So können zum Beispiel die Garantien für das Handshake-Protokoll für alle Komponenten wiederverwendet werden. Der Beweisaufwand für die temporallogischen Beweise ist minimal, da sowohl die prädikatenlogischen Simplifikationen als auch die symbolischen Ausführungsschritte von KIV automatisch ausgeführt werden können. Lediglich beim Erzeugen und Anwenden der Induktionshypothese ist in einigen Fällen ein interaktiver Eingriff von Seiten des Benutzers notwendig. Da der Parallelоperator durch das Modularisierungstheorem eliminiert wurde, sind die temporallogi-

schen Beweise sehr linear aufgebaut. Deshalb lässt sich auch der Beweis relativ einfach nachvollziehen und die eventuell nötigen Interaktionen können leicht gefunden werden.

4.7 Andere Arbeiten zur Rely-Guarantee-Methode

Die Verifikation paralleler Algorithmen ist ein sehr umfangreiches Wissensgebiet. Die wichtigsten Teilgebiete betreffen unter anderem die Verwendung von Invarianten [Ash75], Reduktion [Lip75, CL98, Mis03], Temporallogik [Lam94], Verifikationsdiagramme [Pnu85] und Linearisierbarkeit [HW90].

Einen weiteren wichtigen Beitrag zur Verifikation paralleler Algorithmen liefert der so genannte Rely-Guarantee-Ansatz (auch Assumption-Commitment oder Assumption-Guarantee genannt). Dieser Ansatz geht auf Arbeiten von Jones [Jon83b] und Misra & Chandi [MC81] zurück. Jones prägte den Begriff Rely-Guarantee. Sein Ansatz betrachtet nebenläufige Systeme mit gemeinsamen Variablen (shared-variables) [Jon83b, Jon83a, Jon81]. Misra & Chandi benutzen dagegen einen Ansatz, der ein nebenläufiges System mit synchronem Nachrichtenaustausch betrachtet [MC81]. Dieser Ansatz wurde als Assumption-Commitment bezeichnet. Beide Ansätze wurden von vielen Forschern aufgegriffen und verändert bzw. erweitert. Im Folgenden werden die wichtigsten Aspekte dieser Arbeiten vorgestellt.

De Roever et al. [dRdBH$^+$01, XdRH97] unterscheiden mit den so genannten Aczel-traces ähnlich zu ITL$^+$ zwischen System- und Umgebungsschritt. Aufbauend darauf wurde in dieser Arbeit eine Rely-Guarantee-Methode entwickelt, deren Beweisprämissen denen des Modularisierusgstheorems ohne Invariante (siehe Kapitel 4.2.1) entsprechen. Zusätzlich wird die Vollständigkeit dieser Technik nachgewiesen. Im Unterschied zur hier beschriebenen Technik wird dort allerdings nur ein generisches Framework vorgestellt, ohne eine Logik oder einen Kalkül.

Eine weitere Arbeit auf dem Gebiet Rely-Guarantee stammt von Cau & Colette [XCC94, CC96]. Auch hier wird ein generisches Framework ohne Logik vorgestellt. Dieser Ansatz stellt ein Modularisierungstheorem vor, das sowohl für zustands- als auch für nachrichtenbasierte Systeme geeignet ist. Die Beweisverpflichtungen entsprechen dem Ansatz von de Roever et al. und dem hier verfolgten Verfahren ohne Invarianten (Kapitel 4.2.1).

Von Colette & Knapp wurde ein weiterer Ansatz beschrieben [CK95], der ebenfalls ein Framework ohne Kalkül oder Logik beschreibt. Auch bei diesem Ansatz wird zwischen System- und Umgebungsschritt unterschieden. Im Unterschied zu ITL$^+$ oder Aczel-Traces alternieren aber System- und Umgebungsschritte nicht; diese können vielmehr in beliebiger Reihenfolge auftreten. Die Modularisierungstechnik ist ähnlich wie in [XCC94, CC96] beschrieben, berücksichtigt aber auch Invarianten und erlaubt daher auch den Beweis von *leadsto*-Eigenschaften. Diese Lebendigkeitseigenschaften sind wesentlich einfacher als die hier in Kapitel 9 untersuchte Lock-Freedom-Eigenschaft. Die in Kapitel 9 beschriebene Beweismethode zur Einbettung des Lock-Freedom-Theorems kann auf leadsto-Eigenschaften übertragen werden. Dabei muss im We-

sentlichen anstatt des Lifting-Lemmas für Lock-Freedom (siehe Kapitel 8.4.4, Lemma 14) eine Formel mit den beiden leadsto-Eigenschaften bewiesen werden. Dieser Beweis ist wesentlich einfacher als der Beweis für Lock-Freedom.

Ein anderer Ansatz auf diesem Gebiet wurde von Abadi & Lamport [AL95] vorgestellt. Diese Arbeit erweitert und ersetzt die in [AL89] vorgestellte Theorie. Dort wird eine Rely-Guarantee-Technik (unter dem Namen „Assumption-Commitment“) für die Temporallogik TLA [Lam94, Lam02] vorgestellt. Eine Werkzeugunterstüzung zu diesem Ansatz ist in [KL93] beschrieben. TLA benutzt Konjunktion als Paralleloperator und hat damit auf natürliche Weise einen kompositionalen Paralleloperator. Im Unterschied zum hier vorgestellten Ansatz müssen die Komponenten in einer stotteräquivalenten Normalform spezifiziert werden. Zusätzliche technische Aspekte wie Fairness oder die genaue Spezifikation der Art der Paralellität (interleaved, synchron etc.) müssen ebenfalls in dieser Normalform spezifiziert werden. Durch den expliziten Umgebungsschritt kann in ITL$^+$ auf die Stotteräquivalenz der Komponenten verzichtet werden. Fairness ist in den ITL$^+$-Interleaving-Operator schon integriert und muss nicht separat spezifiziert werden. Der $\overset{+}{\rightarrow}$-Operator wird in [AL95] als neuer semantischer Operator definiert. Am Ende der Arbeit gehen die Autoren darauf ein, wie die vorgestellte Methode auch zum Beweis von Verfeinerungseigenschaften genutzt werden kann. Eine in der vorliegenden Arbeit entwickelte Methode zum Beweis von Verfeinerungseigenschaften wird im Zusammenhang mit der Verifikation von lock-freien Algorithmen in Kapitel 7 beschrieben. Diese ist aber im Unterschied zur Arbeit von Abadi & Lamport [AL95] auf die Existenz einer Abstraktionsfunktion angewiesen. Für die untersuchten lock-freien Algorithmen ist dies allerdings kein großer Nachteil, da bisher nur ein einziger lock-freier Algorithmus bekannt ist, für den keine derartige Abstraktionsfunktion angegeben werden kann und dieser auch keine praktische Relevanz hat. Die in dieser Arbeit vorgestellte Verfeinerungsmethode basiert auf der Verfeinerung von ganzen Abläufen (traces), während in TLA die Verfeinerung auf einzelnen Transitionen basiert. In Kapitel 9 wird dargestellt, wie mit der Verfeinerung von Abläufen für einige komplexe Algorithmen auf spezielle Techniken wie z. B. Rückwärtssimulation (backward-simulation, siehe z. B. [LV95]) verzichtet werden kann, die klassische auf Transitionsverfeinerung basierende Ansätze für solche Fälle benötigen.

Eine weitere Arbeit wurde von Jonsson & Tsay [JT96] vorgestellt. Diese ist bezüglich der Mächtigkeit vergleichbar mit der Rely-Guarantee-Methode von Abadi & Lamport. Dort werden allerdings Vergangenheitsoperatoren zur Formulierung der Rely- und Guarantee-Bedingungen benutzt. Dadurch ist es möglich, Teile der Theorie syntaktisch aus LTL mit Past-Operatoren abzuleiten, anstatt eine rein semantische Theorie zu entwickeln. Allerdings ist auch diese Theorie auf semantische Konstrukte wie Maschinenabgeschlossenheit und Stotteräquivalenz angewiesen. Auch dieser Ansatz erlaubt den Nachweis von Verfeinerung. Für die Verfeinerung gilt im Wesentlichen das Gleiche wie beim Ansatz von Abadi & Lamport. Für die symbolische Ausführung ist der Ansatz von Jonsson & Tsay nicht geeignet, da er Vergangenheitsoperatoren verwendet.

Eine Arbeit, die sich ausschließlich mit synchronen Systemen beschäftigt, ist von Clarke et al. [CGP00]. Werkzeugunterstützung für diese Technik ist z. B. in

[GL94] dargestellt. Diese Technik wurde speziell für Modelchecking entwickelt und beruht auf ACTL-Formeln (das ist eine Untermenge der CTL-Formeln, in denen nur All-Pfadquantoren vorkommen) und fairen Kripke-Strukturen. Im Gegensatz zu dem in Kapitel 5 vorgestellten Ansatz, darf bei dieser Technik auf eine Variable jeweils nur eine Komponente schreibend zugreifen. Für den synchron parallelen Operator wird eine der Kompositionalität der Parallelopeoratoren (siehe Kapitel 4.1.1 bzw. 5.1.1) entsprechende Eigenschaft nachgewiesen. Zur Formulierung der Rely- und Guarantee-Bedingungen können ACTL-Formeln (inkl. Lebendigkeit) angenommen werden. Das vorgestellte Modularisierungstheorem ähnelt der in Kapitel 5.3.5 bzw. in [Sch08] beschriebenen Technik der Beweisdekomposition. Dabei werden einzelne Komponenten unterhalb des Paralleloperators schrittweise durch abstraktere Formeln ersetzt. Zum Nachweis der Gesamteigenschaft muss aber ein paralleles System berücksichtigt werden. Dies ist komplexer als die Beweisverpflichtungen im hier vorgestellten Modularisierungstheorem für synchrone Systeme (siehe Kapitel 5).

4.8 Zusammenfassung

Durch die explizite Unterscheidung von System- und Umgebungsschritt ist es in ITL^+ möglich, den Sustain-Operator direkt als Formel auszudrücken. Auf dieser Basis konnte gezeigt werden, dass sich die Rely-Guarantee-Methode in ITL^+ einbetten lässt. Die Anwendung dieser Methode wurde anhand des ProChaCon-Beispiels gezeigt.

Eine Besonderheit dieses Ansatzes ist, dass die Theorie bzw. das Modularisierungstheorem komplett mit dem ITL^+-Kalkül bewiesen werden kann. Dies hat den Vorteil, dass die Korrektheit des Modularisierungstheorems maschinengestützt beweisbar ist. Diese Theorie kann dann für Fallstudien instanziert werden. Somit bietet dieser Ansatz eine große Flexibilität, da je nach Verwendung die Theorie entsprechend angepasst und erweitert werden kann, ohne das zugrundeliegende Werkzeug verändern zu müssen. In diesem Kapitel wurde dies an einer Erweiterung des Theorems mit Invarianten demonstriert.

Im nächsten Kapitel wird anhand synchroner Parallelität gezeigt, dass dieser Ansatz nicht nur auf interleaved-parallele Systeme anwendbar ist. In den Kapiteln 6 bis 9 werden lock-freie Algorithmen untersucht. Dabei wird beschrieben, wie sich der Ansatz zum Nachweis von Verfeinerungs- und Lebendigkeitseigenschaften erweitern lässt. Die Anwendbarkeit in der Praxis wird anhand von lock-freien Algorithmen demonstriert, die nicht trivial sind.

Kapitel 5

Synchron parallele Systeme

Im vorhergehenden Kapitel wurden modulare Beweistechniken von interleaved parallelen Systemen untersucht. Eine weitere Systemklasse sind synchron parallele Systeme. Dabei führen alle Komponenten gleichzeitig einen Schritt aus. So werden zum Beispiel Hardwareschaltungen oder auch STATEMATE-Statecharts synchron ausgeführt. Eine weitere – eher ungewöhnliche – Art von synchronen Systemen sind Asbru-Pläne. Diese wurden zur Modellierung von natürlichen Abläufen, wie zum Beispiel medizinischer Behandlungsprotokolle, entwickelt. Im Folgenden wird eine modulare Beweistechnik für synchron parallele Systeme vorgestellt. Dazu wird in Abschnitt 5.1 zunächst ein Synchron-Parallel-Operator vorgestellt und in Abschnitt 5.2 ein Modularisierungstheorem für diesen Operator. Die Anwendung des Theorems wird in Abschnitt 5.3 anhand von Asbru-Plänen skizziert.

5.1 Der Synchron-Parallel-Operator

In diesem Abschnitt wird die Einbettung eines *Synchron-Parallel-Operators* (SPO) in KIV beschrieben. Ein wichtiger Aspekte bei der Definition eines SPO $||_s$ ist die Behandlung von gleichzeitigen Änderungen einer Variablen durch mehrere Komponenten, da in einem synchronen System $P_1 \,||_s\, P_2$ die beiden Komponenten P_1 und P_2 gleichzeitig einen Systemschritt ausführen und eine Variable verändern können. Für eine Variable S, für die von P_1 und P_2 unterschiedliche lokale Werte S_1' und S_2' berechnet werden, muss dieser Schreibkonflikt aufgelöst werden und ein Ergebnis des globalen Schritts S' definiert werden. Das Verhalten bei Schreibkonflikten hängt dabei von der betrachteten Fallstudie ab. Beispielsweise könnte bei einem Schreibkonflikt ein Wert nichtdeterministisch ausgewählt werden. Eine andere Möglichkeit wäre, im System sicher zu stellen, dass Schreibkonflikte gar nicht erst auftreten, das heißt, zwei Komponenten können nicht zwei Operationen synchron ausführen, die zu einem Konflikt führen. Als Variante davon kann man schreibende Zugriffe zulassen, wenn sie auf unterschiedliche Felder eines komplexen Datentyps – z.B. eines Arrays – zugreifen. Diese Variante wird zum Beispiel in Asbru benutzt.

Um all diese unterschiedlichen Möglichkeiten zu berücksichtigen, wurde für die Integration eines SPOs $||_s$ in ITL$^+$ ein flexibler Ansatz gewählt, bei dem

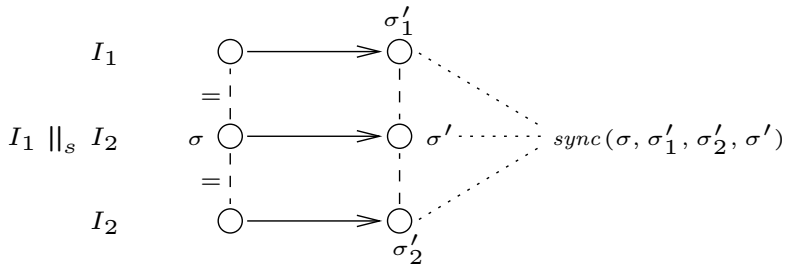


Abbildung 5.1: Darstellung eines synchron parallelen Systemschritts

die Art der Synchronisation in der Anwendung spezifiziert werden kann. Dazu wird für jeden Datentyp t das überladene Prädikat *sync*

$$sync : t \times t \times t \times t$$

definiert. Dieses Prädikat wird zum Auflösen von Schreibkonflikten benutzt. Im obigen Beispiel, in dem die beiden Komponenten P_1 und P_2 für die Variable S die beiden Werte S'_1 und S'_2 in ihrem lokalen Systemschritt berechnen, können mit $sync(S, S'_1, S'_2, S')$ alle gültigen Werte S' für den globalen Systemschritt angegeben werden.

Da alle Werte synchronisiert sein müssen, wird *sync* auch über Zustände definiert:

Definition 26 *(Sync Prädikat).*
Sei **SIG** eine Signatur mit $sync_t \in \mathbf{OP}_{t\times t\times t\times t}$ für alle $t \in \mathbf{T}$, sei $\mathcal{A}_{\mathbf{SIG}}$ eine Algebra und seien $\sigma, \sigma'_1, \sigma'_2$ und σ' Zustände. Dann ist

$$sync(\sigma, \sigma'_1, \sigma'_2, \sigma') = \text{tt} \quad \text{gdw.} \quad \begin{array}{l}\text{für alle } t \in \mathbf{T} \text{ und für alle } V \in \mathbf{DV}_t \text{ gilt} \\ (sync_t)_{\mathcal{A}}(\sigma(V), \sigma'_1(V), \sigma'_2(V), \sigma'(V)) = \text{tt}\end{array}$$

□

Eine graphische Darstellung, welche Zustände der lokalen bzw. parallelen Abläufe synchronisiert werden, ist in Abbildung 5.1 angegeben.

Mit einem *sync*-Prädikat, das über Zustände definiert ist, lässt sich die Semantik des SPO formal definieren. Ähnlich wie beim Interleaving-Operator wird die Semantik durch SOS-Regeln definiert, die in Tabelle 5.1 abgebildet sind. Die Regel *sync stp* behandelt den Fall, in dem beide Komponenten synchron einen Schritt ausführen. Die beiden Regeln *sync right lst* und *sync left lst* spezifizieren die Semantik, falls die rechte bzw. linke Komponente terminiert.

Die Regeln zur symbolischen Ausführung sind in Tabelle 5.2 dargestellt. Die drei Regeln *sync lem*, *sync dis* und *sync ex* entsprechen dem Standard. Die beiden Regeln *sync right lst* und *sync left lst* werden benutzt, falls eine der beiden parallelen Komponenten terminiert. Dabei muss für die terminierende Komponente gezeigt werden, dass der letzte Schritt ein Stotterschritt ist. Der weitere Ablauf wird dann durch die andere Komponente fortgesetzt. Die letzte Regel *sync stp* behandelt den Fall, in dem beide Komponenten einen synchronen

$$\frac{I_1 = (\sigma, \sigma_1', I_1') \qquad I_2 = (\sigma, \sigma_2', I_2') \qquad sync(\sigma, \sigma_1', \sigma_2', \sigma')}{(I_1 \parallel_s I_2) \xrightarrow{\sigma,\sigma'} (I_1' \parallel_s I_2')} \; sync\ stp$$

$$\frac{I_1 = (\sigma, \sigma_1', I_1') \qquad I_2 = (\sigma)}{(I_1 \parallel_s I_2) \xrightarrow{\sigma,\sigma_1'} I_1'} \; sync\ right\ lst$$

$$\frac{I_1 = (\sigma) \qquad I_2 = (\sigma, \sigma_2', I_2')}{(I_1 \parallel_s I_2) \xrightarrow{\sigma,\sigma_2'} I_2'} \; sync\ left\ lst$$

Tabelle 5.1: Semantik des Synchron-Operators

Schritt durchführen. Dabei werden die gestrichenen Variablen A' in den Transitionsformeln τ_1 bzw. τ_2 durch neue statische Variablen ersetzt, die dann durch das *sync*-Prädikat mit der globalen Variable A' synchronisiert werden.

$$\frac{\vdash \varphi_1 \rightarrow \varphi_2}{\vdash \varphi_1 \parallel_s \psi \rightarrow \varphi_2 \parallel_s \psi} \; sync\ lem$$

sync dis:	$(\varphi_1 \vee \varphi_2) \parallel_s \psi$	$\leftrightarrow$	$\varphi_1 \parallel_s \psi \vee \varphi_2 \parallel_s \psi$
sync ex:	$(\exists\, a.\ \varphi) \parallel_s \psi$	$\leftrightarrow$	$\exists\, a_0.\ \varphi_a^{a_0} \parallel_s \psi$

wobei a_0 neu bezüglich $(\text{free}(\varphi) \setminus \{a\}) \cup \text{free}(\psi)$

sync right lst:	$\varphi \parallel_s \tau \wedge \mathbf{last}$	$\leftrightarrow$	$\tau^{A,A}_{A',A''} \wedge \varphi$
sync left lst:	$\tau \wedge \mathbf{last} \parallel_s \psi$	$\leftrightarrow$	$\tau^{A,A}_{A',A''} \wedge \psi$
sync stp:	$\tau_1 \wedge \circ\, \varphi \parallel_s \tau_2 \wedge \circ\, \psi$		
	$\leftrightarrow$	$(\exists\, a_1, a_2.\ \tau_1{}^{a_1}_{A'} \wedge \tau_2{}^{a_2}_{A'} \wedge sync(A, a_1, a_2, A'))$	
		$\wedge \circ\, (\varphi \parallel_s \psi)$	

wobei a_1, a_2 neu bezüglich $\text{free}(\tau_1) \cup \text{free}(\tau_2)$

Tabelle 5.2: Normalformregel für den Synchron-Parallel-Operator

5.1.1 Eigenschaften des Synchron -Parallel-Operators

Die Kommutativität und Assoziativität des $\parallel_s$ Operators hängt von der Definition des *sync*-Prädikats ab. Die folgenden beiden Eigenschaften können benutzt werden, um zu testen, ob das *sync*-Prädikat kommutativ und assoziativ ist:

$$sync(a, b, c, d) \;\rightarrow\; sync(a, c, b, d)$$

$$\exists\, c.\ (sync(a, b_1, b_2, c) \wedge sync(a, c, b_3, d))$$
$$\leftrightarrow \exists\, c.\ (sync(a, b_2, b_3, c) \wedge sync(a, b_1, c, d))$$

Falls die erste Eigenschaft für *sync* gilt, ist der $||_s$ Operator kommutativ. Für Assoziativität muss die zweite Eigenschaft erfüllt sein.

Genau wie beim Interleaving Operator kann für den Synchron-Parallel-Operator mit Satz 1 aus Kapitel 4.1.1 gezeigt werden, dass dieser kompositional ist. Damit ist für $||_s$ die folgende Regel gültig:

$$\frac{\vdash\ \varphi_1 \rightarrow \varphi_2 \qquad \varphi_2\ ||_s\ \psi,\ \Gamma\ \vdash\ \Delta}{\varphi_1\ ||_s\ \psi,\ \Gamma\ \vdash\ \Delta}\text{sync lem}$$

Mit dieser Regel kann eine Komponente φ_1 unterhalb des Synchron-Parallel-Operators durch eine abstraktere Komponente φ_2 ersetzt werden.

5.2 Modularisierungstheorem für synchrone Systeme

Im Folgenden wird ein Modularisierungstheorem für den $||_s$ Operator vorgestellt. Hier wird das Theorem für zwei Komponenten erläutert. Um das Theorem auf n Komponenten zu erweitern, können ähnlich wie in Kapitel 4.3 Prädikate *Rto* und *Gto* verwendet werden, die eine Aggregation der lokalen Annahmen und Garantien darstellen.

Wie bei dem Modularisierungstheorem für Interleaving soll das Ziel erreicht werden, eine Gesamteigenschaft

$$P_1(S)\ ||_s\ P_2(S) \vdash R(S', S'') \xrightarrow{+} G(S, S') \tag{5.1}$$

aus den Rely-Guarantee-Eigenschaften der beiden Komponenten P_1 und P_2 herzuleiten. Dazu wird für die beiden P_i angenommen, dass folgendes gilt:

$$P_i(S) \vdash R_i(S', S'') \xrightarrow{+} G_i(S, S') \tag{5.2}$$

Nun müssen noch die prädikatenlogischen Bedingungen für die Rely- und Guarantee-Eigenschaften bestimmt werden. Da alle Prozesse gemeinsam einen Schritt machen, müssen alle Guarantee-Bedingungen miteinander durch das *sync*-Prädikat synchronisiert werden. Für die Guarantee-Bedingungen muss gezeigt werden, dass die Synchronisation der beiden Garantien G_1 und G_2 die Eigenschaft G impliziert:

$$sync(S, S_1', S_2', S') \wedge G_1(S, S_1') \wedge G_2(S, S_2') \rightarrow G(S, S') \tag{5.3}$$

Diese Formel entspricht dem in Abbildung 5.1 dargestellten Zusammenhang des globalen und der beiden lokalen Systemschritte. Falls eine der beiden Komponenten terminiert, muss die andere Komponente G alleine erfüllen:

$$G_1(S, S') \vee G_2(S, S') \rightarrow G(S, S') \tag{5.4}$$

Bei den Umgebungsannahmen muss sichergestellt werden, dass die lokalen Um-

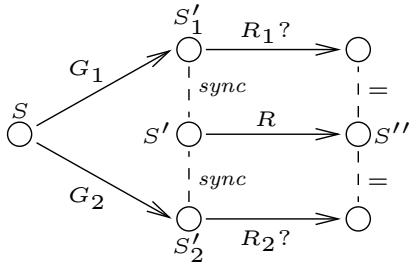


Abbildung 5.2: Darstellung der Umgebungsschritte

gebungsannahmen von der globalen Umgebungsannahme erfüllt werden. Dies wird durch die folgende Formel bewerkstelligt:

$$\begin{aligned} & sync(S, S'_1, S'_2, S') \wedge G_1(S, S') \wedge G_2(S, S') \wedge R(S', S'') \\ \rightarrow\ & R_1(S'_1, S'') \wedge R_2(S'_2, S'') \end{aligned} \tag{5.5}$$

Bei dieser Formel muss ebenfalls die Synchronisierung berücksichtigt werden. Dabei beginnen die lokalen Umgebungsschritte auf ihren lokalen Zwischenzuständen S'_1 und S'_2. Als Systemschritt können dazu die lokalen Garantien angenommen werden und als Umgebungsschritt R. Der Zusammenhang zwischen den Bedingungen ist in Abbildung 5.2 graphisch dargestellt. Falls eine Komponente terminiert, muss die lokale Umgebungsannahme der verbleibenden Komponente von der globalen Umgebung erhalten werden. Da die linke oder die rechte Komponente terminieren können, muss die globale Umgebungsannahme beide lokalen Umgebungsannahmen erfüllen:

$$R(S', S'') \rightarrow R_1(S', S'') \wedge R_2(S', S'') \tag{5.6}$$

Diese fünf Bedingungen reichen aus, um die gewünschte Gesamteigenschaft (5.1) zu zeigen:

Satz 4 *Modularisierungstheorem.*
Seien G, R, $G_{1/2}$, $R_{1/2}$ und $P_{1/2}$ so, dass (5.2) bis (5.6) gilt. Dann gilt auch:

$$P_1(S) \parallel_s P_2(S) \ \vdash\ R(S', S'') \xrightarrow{+} G(S', S'')$$

□

Der Beweis ist ähnlich aufgebaut wie für das Modularisierungstheorem für Interleaving. Bis auf das Lifting-Lemma können alle Beweise analog geführt werden. Beim Beweis des Lifting-Lemmas werden durch synchrone Ausführung von abstrakten Formeln weniger Fälle als beim Interleaving generiert, da keine Unterscheidung notwendig ist, welche Komponente einen Schritt ausführt, und blockierte Komponenten nicht separat behandelt werden.

Das Theorem kann auf n Komponenten erweitert werden, indem wie in Kapitel 4.3.1 entsprechende Prädikate Rto_i und Gto_i eingeführt werden, welche die lokalen Garantien und Annahmen mehrerer Komponenten zusammenfassen.

In diesem Theorem kann dann eine der beiden lokalen Eigenschaft durch Rto_{i-1} bzw. Gto_{i-1} und die globale Eigenschaft durch Rto_n bzw. Gto_n ersetzt werden.

Wie in Kapitel 4.2.2 kann das Theorem auch um Invarianten erweitert werden. Dabei müssen die beiden zusätzlichen Prämissen

$$G(S, S') \wedge I(S) \rightarrow I(S') \tag{5.7}$$

und

$$R(S', S'') \wedge I(S') \rightarrow I(S'') \tag{5.8}$$

gezeigt werden. Dies erlaubt es, zusätzlich in allen anderen Beweisverpflichtungen die Invariante $I(S)$ und $I(S')$ als zusätzliche Voraussetzung anzunehmen.

5.3 Beispiel: Medizinische Leitlinien

Die praktische Anwendung des Modularisierungstheorems für synchron parallele Systeme wird im Folgenden anhand eines Beispiels aus dem Bereich der Medizin dargestellt.

Medizinische Leitlinien sind Behandlungsempfehlungen für das medizinische Personal, die auf Grundlage von empirisch gewonnenen Erkenntnissen und Erfahrungen erstellt werden. Sie werden üblicherweise in Form von systematisch entwickelten Aussagen angegeben, um Mediziner bei Behandlungsentscheidungen zu unterstützen [FL92]. Leitlinien können die Qualität der Behandlung verbessern. Dies wurde in eingehenden Untersuchungen nachgewiesen [ehc94]. Außerdem können Leitlinien die Behandlungskosten senken [CH95]. Die Dokumente der Leitlinien bestehen aus einer komplexen Folge von bedingten Anweisungen, die in natürlicher Sprache verfasst sind. Die Qualität der Leitlinien selbst sind dabei eine wichtige Voraussetzung. Diese dürfen keine Behandlungen zulassen, die dem aktuellen Stand der medizinischen Forschung widersprechen. Da Leitlinien oft sehr komplex sind, ist dies jedoch nicht immer von vornherein ersichtlich. Um die Qualität von medizinischen Leitlinien zu verbessern, wurde in der neueren Forschung die Verwendung von Verifikationsmethoden zur Qualitätssicherung der Leitlinien untersucht (siehe z.B. [tTMB+06, GTB+06, Luc03]). Zu diesem Zweck werden die Leitlinien in ein formales Modell umgewandelt, damit sie mit formalen Analysewerkzeugen überprüfbar sind. Auf diese Weise können mehrdeutige, inkonsistente oder unvollständige Teile einer Leitlinie aufgedeckt werden.

5.3.1 Überblick über das PROTOCURE-Projekt

Ein Ansatz zur Verifikation von Leitlinien wurde im europäischen Projekt PROTOCURE entwickelt [tTMB+06, Sch08]. Dabei werden medizinische Leitlinien als parallele Programme aufgefasst. Diese können mit den bekannten Methoden zur Verifikation von parallelen Programmen untersucht werden. Zur Verifikation wurde im PROTOCURE-Projekt die hier vorgestellte Logik ITL$^+$ mit dem Werkzeug KIV benutzt. Zur Formalisierung von Leitlinien in ITL$^+$ wurde als Zwischenschritt die Planbeschreibungssprache Asbru benutzt [SKM02]. Diese hierarchische Plansprache dient zur Modellierung von natürlichen Abläufen,

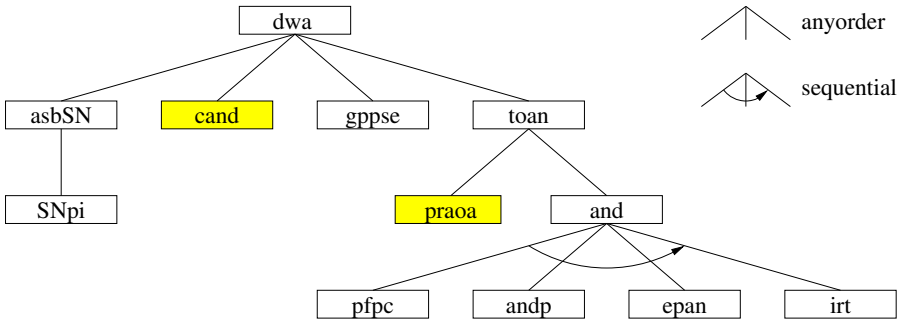


Abbildung 5.3: Ausschnitt aus der Planhierarchie der Brustkrebsfallstudie

wie sie in medizinischen Leitlinien vorgegeben sind. Dabei wird in Asbru ein besonderer Fokus auf die zeitliche Abfolge der spezifizierten Abläufe gelegt. Die Übersetzung von Asbru-Plänen in ITL$^+$-Spezifikationen wurde von Schmitt in [Sch08] beschrieben. Als Fallstudien wurden im PROTOCURE-Projekt mehrere Leitlinien betrachtet: Diabetes Mellitus Typ 2 [MRtTvH02], Gelbsucht bei Neugeborenen [MBtTH] und die Behandlung von Brustkrebs [PCMS$^+$05], die bezüglich des Umfangs die bedeutendste Fallstudie darstellt.

5.3.2 Asbru-Pläne

Wie im vorhergehenden Abschnitt erwähnt, ist Asbru eine hierarchische Plansprache. Ein Beispiel für eine solche Planhierarchie ist in Abbildung 5.3 dargestellt. Dieser Plan stammt aus dem ersten Kapitel der Leitlinie zur Brustkrebsbehandlung [PCMS$^+$05]. Die gesamte Planhierarchie dieses Kapitels enthält insgesamt 49 Pläne. In diesem Kapitel wird eine bestimmte Phase des Brustkrebses beschrieben, die „DCIS“ genannt wird. Dabei handelt es sich um krankhafte Wucherungen in den Milchgängen der weiblichen Brust. Der oberste Plan der dargestellten Hierarchie betrifft einen Teil der Behandlung, in dem die Lymphknoten der Achsel untersucht werden (*dwa* für *dealing with axilla*). Dieser Plan enthält vier Unterknoten, die nacheinander in beliebiger Reihenfolge ausgeführt werden sollen (d. h., die Reihenfolge kann durch den behandelnden Arzt bestimmt werden). Von besonderem Interesse ist hier der Plan *cand*, der eine komplette Entfernung der Lymphknoten der Achsel beinhaltet, und der Plan *toan*, der eine Kombination aus chirurgischer Entfernung einiger Lymphknoten (Plan *and*) und Strahlentherapie (Plan *praoa*) enthält.

Die Semantik von Asbru-Plänen ist detailliert in [Sch08] beschrieben. Formal wird die Semantik durch SOS-Regeln definiert; eine leichter verständliche, informelle Beschreibung durch Statecharts ist ebenfalls angegeben. Die für das betrachtete Beispiel relevanten Teile der Statechart-Semantik sind in Abbildung 5.4 dargestellt. Der Statechart stellt die Auswertungsphasen eines Planes dar. Ein Plan kann dabei vier Phasen durchlaufen: *Inactive*, *Selection*, *Execution* und *Terminated*. In der Selection-Phase wird überprüft, ob der Plan ausgeführt werden kann. Dabei werden verschiedene medizinische Bedingungen geprüft (Bedingungen sind im Plan durch eckige Klammern gekennzeichnet). Von dieser

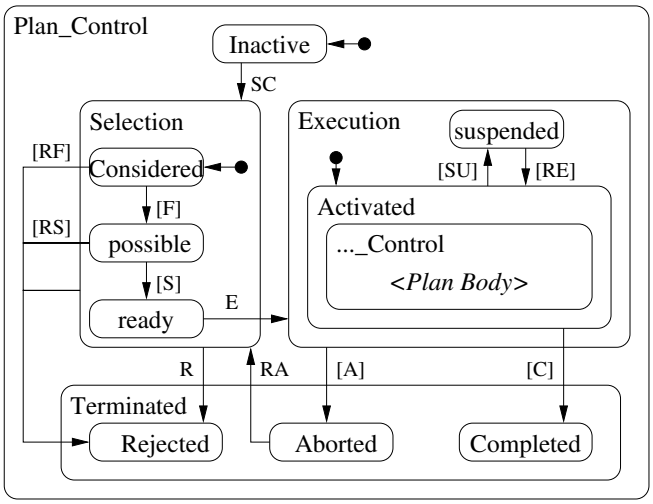


Abbildung 5.4: Statechartsemantik von Asbru (nach [BDRS06])

Phase kann der Plan in die Execution-Phase übergehen. In dieser werden die eigentlichen Aktionen des Plans ausgeführt. Falls der Plan Unterpläne hat, werden diese beim Betreten der Execution-Phase gestartet, so dass sie gleichzeitig ihre Selection-Phase durchlaufen. Die Unterpläne werden dabei durch Signale kontrolliert (SC, E, R und RA). Wann die Unterpläne aktiviert werden, so dass diese ihrerseits wieder ihren Execution-Zustand betreten, wird im Planrumpf eines Planes spezifiziert. Dabei gibt es mehrere Möglichkeiten. Zum Beispiel darf bei sogenannten *anyorder*-Plänen immer nur ein Unterplan im Zustand *Activated* sein, die Reihenfolge der Aktivierung der verschiedenen Unterpläne ist dabei aber nicht festgelegt. Die anderen Unterpläne verbleiben dabei entweder in der Selection-Phase oder sind terminiert. Bei *sequential*-Plänen darf sich ebenfalls immer nur ein Unterplan im Zustand *Activated* befinden. Dabei werden die Unterpläne aber nach einer festgelegten Reihenfolge aktiviert.

5.3.3 Formalisierung von Asbru in ITL$^+$

Asbru muss in eine Spezifikationssprache eines Verifikationswerkzeugs eingebettet werden, damit eine Prüfung mit diesem Werkzeug durchgeführt werden kann. Im PROTOCURE-Projekt sind Pläne als synchron parallele Programme in ITL$^+$ spezifiziert [Sch08].

Grundlage dieser Einbettung war dabei die Statechart-Semantik. Die entsprechenden Zustände und Transitionen können direkt in sequentielle Programme übersetzt werden. Jeder Zustand wird durch eine Prozedur *Zustandsname#* modelliert. Diese Prozedur realisiert das dynamische Verhalten eines Plans. Zum Beispiel wird für einen Plan *plan-name*, der sich im Zustand *Considered* befindet, die Prozedur *considered#*(*plan-name*, AS) ausgeführt. Diese testet die beiden Bedingungen [F] und [RF]. Falls eine dieser Bedingungen wahr ist, wird die entsprechende Prozedur ausgeführt, z. B. *possible#*(*plan-name*, AS). Falls keine Bedingung wahr ist, wird weiterhin *considered#* ausgeführt.

In der Variable AS sind die aktiven Zustände aller Pläne als dynamische

```
inactive-rec#(AS, start-plan-list){
    if*  start-plan-list = [] then {
        skip
    } else if* start-plan-list.rest = [] then {
        inactive#(start-plan-list.first, AS)
    } else {
            inactive-rec#(AS, start-plan-list.rest)
        ||_s inactive#(start-plan-list.first, AS)
    }
}
```

Abbildung 5.5: Prozedur zum rekursiven Starten von Unterplänen

Funktion gespeichert. Mit $AS[plan\text{-}name]$ wird der aktuelle Zustand des Plans *plan-name* selektiert. Asbru-Pläne benutzen noch weitere Variablen, z. B. für die Patientendaten. Diese sind teilweise sehr komplex, zum Verständnis der hier vorgestellten Anwendung aber nicht notwendig. Daher werden sie aus Gründen der Übersichtlichkeit ausgelassen.

Die Ausführung wird komplexer, falls ein Plan sich in der *Execution*-Phase befindet. Zuerst wird in einer Prozedur *activated#* geprüft, ob der Plan abgebrochen (d. h., er geht in den Zustand *Terminated* über) oder suspendiert wird. Falls dies nicht der Fall ist, wird die entsprechende *Control*-Prozedur des Planes ausgeführt, das heißt, falls die Unterpläne zum Beispiel in der Reihenfolge *anyorder* ausgeführt werden sollen, wird die entsprechende Prozedur *anyorder#* ausgeführt. Diese Prozedur erzeugt die Prozesse für die Unterpläne mit der Prozedur *inactive-rec#*, falls diese noch inaktiv sind, da es in diesem Fall auch keinen laufenden Prozess für die Unterpläne gibt. Die Prozedur *inactive-rec#* ist in Abbildung 5.5 dargestellt. Sie wird ähnlich zur *SpawnProc*-Prozedur aus Kapitel 4 rekursiv definiert. Nachdem die Unterpläne gestartet wurden, durchlaufen die Unterpläne parallel ihre *Selection*-Phase. Falls kein Unterplan in der *Execution*-Phase ist, aber einer der Unterpläne sich im Zustand *Ready* befindet, wird diesem Unterplan das Startsignal gesendet.

5.3.4 Behandlung von Schreibkonflikten in Asbru

Die wichtigste Datenstruktur in Asbru ist die dynamische Funktion *asbru-state*. Diese bildet die Plannamen auf den aktuellen Zustand des entsprechenden Plans ab. Die Variable *AS* in obigen Plänen ist vom Typ *asbru-state*. In jedem Schritt aktualisiert ein aktiver Plan den *asbru-state* mit seinem aktuellen Zustand. Dadurch werden in einem Schritt des Gesamtsystems mehrere Werte der Funktion gleichzeitig abgeändert. Da jeder Plan nur seinen eigenen Zustand ändert, kommt es hier zu keinen Konflikt. Um die verschiedenen Schreibzugriffe zu synchronisieren und den neuen Wert des *asbru-state* nach einem Schritt zu ermitteln wird das in Abschnitt 5.1 beschriebene *sync*-Prädikat benutzt. Das *sync*-Prädikat wird dabei so definiert, dass sich immer der geänderte Wert

durchsetzt:

$$\begin{aligned} sync(as, as_1, as_2, as_0) \quad :\leftrightarrow \quad \forall\, p. \quad & as[p] = as_1[p] \wedge as_2[p] = as_0[p] \\ & \vee\ as[p] = as_2[p] \wedge as_1[p] = as_0[p] \\ & \vee\ as_1[p] = as_0[p] \wedge as_2[p] = as_0[p] \end{aligned}$$

Wenn die erste bzw. zweite Komponente $as[p]$ nicht verändert hat, wird als globales Ergebnis das Ergebnis der jeweils anderen Komponente benutzt (erste bzw. zweite Zeile). Falls beide Komponenten $as[p]$ ändern, müssen beide das gleiche Ergebnis haben (dritte Zeile). Alle anderen Fälle werden als ungültig betrachtet und führen daher zu einem Widerspruch.

Eine alternative Möglichkeit zur Synchronisierung wäre, bei Schreibkonflikten zum Beispiel indeterministisches Verhalten zuzulassen. Damit würde dann bei der Verifikation zusätzlich gezeigt werden, dass solche Konflikte nicht auftreten können bzw. nicht für den korrekten Ablauf relevant sind. In der vorliegenden Arbeit wurde jedoch das *sync*-Prädikat aus der PROTOCURE-Fallstudie beibehalten, um eine bessere Vergleichbarkeit der Ergebnisse zu gewährleisten.

5.3.5 Anwendung des Modularisierungstheorems

Anhand des oben vorgestellten Planes *dwa* soll als Beispiel die Anwendung des Modularisierungstheorems erläutert werden. Als zu verifizierende Eigenschaft wurde aus der Brustkrebs-Fallstudie [PCMS$^+$05] die medizinisch relevante Aussage 20 gewählt. Diese Eigenschaft besagt, dass keine Strahlenbehandlung an Patienten durchgeführt werden darf, deren Achsellymphknoten chirurgisch entfernt wurden. Die beiden dafür relevanten Pläne sind *cand* (Entfernung der Achsellymphknoten) und *praoa* (Strahlenbehandlung). Die Aussage 20 besagt also, dass der Plan *praoa* nicht aktiviert werden darf, falls der Plan *cand* davor einmal aktiv war. Diese Aussage wurde im PROTOCURE-Projekt bewiesen. Der Beweis ist ausführlich in [Sch08], Kapitel 20 beschrieben.

Formal wird die Aussage 20 durch die folgende Formel ausgedrückt:

$$\begin{aligned} & [\mathit{inactive}\#(\mathit{dwa}, AS)]_{AS}, \\ & \Box\ (R(AS', AS'') \wedge R_{GDL}(\mathit{Start}', \mathit{End}', \mathit{Start}'', \mathit{End}'')), \\ & \neg\ \mathit{Start}, \neg\ \mathit{End} \\ \vdash\ & \Box\ (\mathit{Start} \rightarrow AS[\mathit{praoa}] \neq \mathit{activated} \vee \mathit{End}) \end{aligned}$$

Diese Sequenz besagt, dass bei einem Ablauf von *dwa* der Plan *praoa* nicht im kritischen Zeitraum aktiviert werden darf. Der kritische Zeitraum ist das zeitliche Intervall, in dem eine Aktivierung von *praoa* zu einer sofortigen Verletzung der Eigenschaft führt. In diesem Fall besteht der kritische Zeitraum aus allen Zuständen nach einer erfolgreichen Terminierung von *cand*. Er wird durch die beiden Variablen *Start* und *End* markiert, das heißt, ein Zustand ist kritisch, wenn $\mathit{Start} \wedge \neg\ \mathit{End}$ gilt. Die beiden Variablen *Start* und *End* werden entsprechend von der Umgebungsbedingung R_{GDL} gesetzt.[1] Diese fordert, dass *Start*

[1] GDL steht dabei für Guideline-Description-Language.

wahr ist, sobald *cand* erfolgreich terminiert und dass danach *Start* immer wahr bleibt. *End* wird erst gesetzt, sobald *dwa* und alle Unterpläne terminiert sind.

Beweis mit Beweisdekomposition

Die Technik, mit der die Aussage 20 in [Sch08] bewiesen ist, wird „Beweisdekomposition“ genannt. Dabei handelt es sich um eine Technik, welche die Kompositionalität von $||_s$ und Rely-/Guarantee-Bedingungen verwendet, aber kein Modularisierungstheorem. Das Vorgehen bei dieser Technik ist dabei wie folgt:[2]

1. Schritt: Zuerst werden für jeden Plan *plan-name* entsprechende Rely- und Guarantee-Bedingungen $R_{plan\text{-}name}$ und $G_{plan\text{-}name}$ formuliert. Diese werden dann wie in Beweisverpflichtung (5.2) des Modularisierungstheorems bewiesen, für den Plan *asbSN* zum Beispiel mit dem Beweis der folgenden Sequenz:

$$[inactive\#(asbSN, AS)]_{AS} \vdash R_{asbSN}(AS', AS'') \xrightarrow{+} G_{asbSN}(AS, AS') \tag{5.9}$$

Zusätzlich werden aus den lokalen Garantien die zusammengefassten Garantien $Gto[asbSN, cand]$, $Gto[asbSN, cand, gppse]$, etc. und aus den Rely-Bedingungen die zusammengefassten Umgebungsannahmen $Rto[asbSN, cand]$, etc. definiert. Diese können nach einem einfachen Schema aus den lokalen Garantien (bzw. Umgebungsannahmen) abgeleitet werden (siehe [Sch08]).

2. Schritt: Zuerst werden für die beiden Pläne *asbSN* und *cand* deren entsprechende *Rto*- und *Gto*-Bedingungen gezeigt:

$$\begin{aligned} & R_{cand}(AS', AS'') \xrightarrow{+} G_{cand}(AS, AS') \\ ||_s\ & R_{asbSN}(AS', AS'') \xrightarrow{+} G_{asbSN}(AS, AS') \\ \vdash\ & Rto[asbSN, cand](AS', AS'') \xrightarrow{+} Gto[asbSN, cand](AS, AS') \end{aligned} \tag{5.10}$$

Die weiteren *Rto*- und *Gto*-Eigenschaften werden dann mit dem folgenden Beweisschema gezeigt:

$$\begin{aligned} & R_{plan}(AS', AS'') \xrightarrow{+} G_{plan}(AS, AS') \\ ||_s\ & Rto[\langle plan\text{-}list\rangle](AS', AS'') \xrightarrow{+} Gto[\langle plan\text{-}list\rangle](AS, AS') \\ \vdash\ & Rto[\langle plan\text{-}list\rangle, plan](AS', AS'') \xrightarrow{+} Gto[\langle plan\text{-}list\rangle, plan](AS, AS') \end{aligned} \tag{5.11}$$

Dieses Schema wird zum Beweis der Formel

$$\begin{aligned} & Rto[asbSN, cand, gppse, toan, dwa](AS', AS'') \\ \xrightarrow{+}\ & Gto[asbSN, cand, gppse, toan, dwa](AS, AS') \end{aligned} \tag{5.12}$$

fortgesetzt. All diese Beweise sind temporallogische Beweise, bei denen der $||_s$ Operator symbolisch ausgeführt werden muss.

[2] Hier ist der Beweis nur für die oberste Hierarchiestufe von *dwa* beschrieben. Die untergeordneten Stufen können analog bewiesen werden.

3. Schritt: Der eigentliche Beweis kann dann dadurch geführt werden, indem *dwa* solange ausgeführt wird, bis alle Unterpläne erzeugt wurden. Nun kann mit den in den Schritten 1 und 2 bewiesenen Eigenschaften und der Kompositionalität der Beweis zu folgender Sequenz umgeschrieben werden:

$$\begin{aligned} & Rto[asbSN, cand, gppse, toan, dwa] \\ \xrightarrow{+}\ & Gto[asbSN, cand, gppse, toan, dwa](AS, AS'), \\ & \Box\ (R(AS', AS'') \wedge R_{GDL}(Start', End', Start'', End'')), \\ & \neg\ Start, \neg\ End \\ \vdash\ & \Box\ (Start \rightarrow AS[praoa] \neq activated \vee End) \end{aligned} \tag{5.13}$$

Diese Sequenz kann einfach durch symbolische Ausführung gezeigt werden.

Beweis durch Modularisierung

Zur Evaluation der Modularisierungstechnik wurde der oben beschriebene Beweis mit Hilfe des in Abschnitt 5.2 vorgestellten Modularisierungstheorems durchgeführt. Dieser Beweis wird ebenfalls in drei Schritten ausgeführt.

1. Schritt: Auch hier müssen die lokalen Rely- und Guarantee-Bedingungen für die Komponenten festgelegt und bewiesen werden. Die Bedingungen können dabei praktisch unverändert aus dem Dekompositionsbeweis übernommen werden. Die Eigenschaften vom Typ (5.2) des Modularisierungstheorems müssen hier ebenfalls gezeigt werden. Daher ist dieser Schritt vom Aufwand her identisch mit dem 1. Schritt des dekompositionalen Beweises. Es ist jedoch sinnvoll, Teile aus den Garantien in die Invariante auszulagern. Dies macht die Garantien übersichtlicher und verkürzt die Beweise in Schritt 2.

2. Schritt: Anstatt der temporallogischen „lifting“-Beweise im 2. Schritt des Dekompositionsbeweises wird hier das Modularisierungstheorem benutzt, um die Formel 5.12 aus den Komponenten zu folgern. Durch die prädikatenlogischen Prämissen des Modularisierungstheorems entstehen deutlich mehr Beweise als beim Dekompositionsbeweis. Diese sind jedoch wesentlich einfacher zu beweisen.

3. Schritt: Dieser Schritt ist wieder sehr ähnlich zum entsprechenden Schritt des Dekompositionsbeweises. Auch hier werden die Komponenten durch die Formel 5.12 ersetzt und die entstandene Formel kann durch einfache symbolische Ausführung gezeigt werden.

5.3.6 Fazit

Die Technik der Beweisdekomposition ist sehr effektiv und erlaubt eine deutliche Reduktion des Beweisaufwands [Sch08]. Die Abstraktion der Komponenten mit Rely- und Guaranteebedingungen und die beiden Beweisschritte 1. und 3.

sind bei dieser Technik bereits identisch mit der hier vorgestellten Modularisierungstechnik. Allerdings muss noch mit symbolischer Ausführung der abstrakten Rely-Guarantee-Bedingungen gezeigt werden, dass die Komponenten nicht gegenseitig ihre Umgebungsannahmen verletzen. Diese Beweise werden im synchronen Fall bei weitem nicht so umfangreich wie bei interleaved parallelen Systemen.[3] Aber auch hier können diese Beweise über 100 Interaktionen erfordern.

Mit der hier vorgestellten Technik kann dieser Aufwand weiter verringert werden. Durch Reduktion auf prädikatenlogische Beweisverpflichtungen mit dem Modularisierungstheorem können etwa weitere 40% der dabei notwendigen Interaktionen eingespart werden. Diese Reduktion resultiert daher, dass viele für die temporallogischen Beweise notwendigen Schritte wie Induktionsanwendung, Generalisierungen, symbolische Ausführung etc. in das generische Theorem verlagert sind.

Bedeutend ist dabei auch ein anderer Vorteil: Bei komplexeren Systemen ist es kaum möglich, die korrekten Rely- und Guaranteebedingungen auf Anhieb richtig zu formulieren. Meistens ist ein iterativer Prozess notwendig, um die korrekten Bedingungen zu finden. Erfahrungsgemäß sind Rely- oder Guaranteebedingungen häufig zu stark oder zu schwach, so dass diese sich gegenseitig verletzen. Diese Fehler werden in Schritt 2 aufgedeckt. Somit ist es empfehlenswert, diesen Schritt vorzuziehen, um die Fehler frühzeitig zu entdecken. Nachdem die betreffende fehlerhafte Bedingung geeignet verstärkt oder abgeschwächt wurde, muss man alle Beweise mit der fehlerhaften Bedingung erneut ausführen. Da die Beweise bei Beweisdekomposition monolithisch sind, muss der komplette Beweis bei dieser Technik wiederholt werden. Bei Verwendung des Modularisierungstheorems muss dagegen nur ein Teil der Beweise wiederholt werden, da nicht alle von einer Veränderung einer Rely- oder Guaranteebedingung betroffen sind. Auch können Fehler in den Bedingungen wegen der kleineren und übersichtlicheren Beweise schneller lokalisiert und nachvollzogen werden. Dadurch können die Iterationen bei der Formalisierung dieser Bedingungen beschleunigt werden.

[3] Dies resultiert daraus, dass bei synchroner Ausführung von abstrakten Formeln weniger Fälle unterschieden werden. Es vollziehen immer alle Komponenten einen Schritt gleichzeitig und eine Programmblockierung muss nicht separat behandelt werden.

Kapitel 6

Lock-freie Algorithmen

Durch die größere Verbreitung von Multi-Core Architekturen wurde deren effiziente Nutzung durch geeignete parallele Algorithmen immer wichtiger. Dadurch wurden auch einige Arten von Algorithmen populär, die zuvor nur wenig beachtet wurden. Eine solche Klasse sind lock-freie Algorithmen. Diese Algorithmen dienen dazu, effizient auf parallel genutzte Datenstrukturen (wie zum Beispiel Stacks, Queues oder Sets) zugreifen zu können. Im Gegensatz zu den klassischen Algorithmen für parallelen Datenzugriff vermeiden diese Algorithmen – wie der Name impliziert – die Verwendung von Datenstruktur-Locks wie bei den klassischen Mutex-Algorithmen, die zum Beispiel bei dem im Kapitel 3.1 vorgestellten Semaphor-Algorithmus eingesetzt werden. Stattdessen werden atomare Operationen wie *Compare-and-Swap* oder *Load-link/store-conditional* zur Synchronisation der Prozesse verwendet. Diese Operationen sind üblicherweise in allen aktuellen Prozessorarchitekturen vorhanden und auch Hochsprachen wie Java oder C# unterstützen entsprechende Funktionen.

6.1 Nachteile von Locks

Obwohl die Verwendung von Locks bei parallelen Algorithmen weit verbreitet ist, haben sie einige inhärente Nachteile. Deadlocks sind dabei die bekannteste Gefahr. Zwei Prozesse können sich gegenseitig blockieren und so in einen Deadlock geraten, wenn jeder Prozess eine kritische Ressource des anderen durch ein Programm-Lock blockiert, während er selbst auf die andere Ressource wartet. Zu einem ähnlichen Fehlerfall kann es auch kommen, wenn ein Prozess abstürzt, während er eine Ressource blockiert.

Außerdem bilden Locks einen Flaschenhals: Alle Prozesse müssen warten, bis das Lock freigegeben ist, um auf eine Datenstruktur zugreifen zu können. Neben den offensichtlichen Performanzproblemen, die dadurch auftreten, kann dies auch zu sehr subtilen Problemen wie der *„Priority-Inversion“* führen. Dieses Problem tritt auf, wenn ein hochpriorisierter Prozess auf einen niedrig priorisierten Prozess warten muss, weil dieser die betreffende Datenstruktur durch ein Lock blockiert. Dabei kann der hochpriorisierte Prozess die ihm zugewiesenen Prozessorzyklen nicht nutzen, sondern er muss warten bis der niedrigpriorisierte Prozess Prozessorzeit zugewiesen bekommt. Effektiv wird dabei die

Prozesspriorität des hochpriorisierten Prozesses auf die des niedrig priorisierten Prozesses reduziert. Da solche Fehler vom Scheduling der verschiedenen Prozesse abhängen, lassen sie sich nur sehr schwer voraussehen oder durch Softwaretests finden. So führte ein Priority-Inversion-Fehler während der NASA-Mars-Pathfinder-Mission 1997 immer wieder zu einem Reset des Bordcomputers des Pathfinder-Roboters. Ein Team am Jet Propulsion Laboratory der NASA benötigte insgesamt drei Wochen, um den Fehler zu lokalisieren [Dur98].

6.2 Lock-Freie Algorithmen

Alle die oben genannten Nachteile werden bei lock-freien Algorithmen vermieden. Allerdings wird dieser Vorteil mit einer höheren Komplexität der Algorithmen selbst erkauft. Außerdem sind lock-freie Algorithmen datenstrukturspezifisch, das heißt, sie müssen für jede Datenstruktur wie zum Beispiel Stacks, Queues oder Heaps angepasst werden. Da infolge der höheren Komplexität die Korrektheit lock-freier Algorithmen nicht immer offensichtlich ist, sind sie ein interessantes Anwendungsgebiet für formale Verifikation. Die Anwendung klassischer Verifikationstechniken scheitert aber bei diesen Algorithmen oft, da sowohl Zeigerstrukturen als auch parallele Programmabläufe berücksichtigt werden müssen.

Lock-freie Algorithmen arbeiten üblicherweise in drei Phasen, um auf eine globale Datenstruktur zuzugreifen. In der ersten Phase werden die benötigten Daten vorbereitet, das heißt, Speicher wird alloziert oder lokale Variablen werden initialisiert. All diese Vorbereitungsschritte sind jedoch unabhängig vom aktuellen Zustand der Datenstruktur, so dass Änderungen der Datenstruktur durch andere Prozesse keinen Einfluss auf die Schritte in dieser Phase haben. In der zweiten Phase wird zunächst eine lokale Kopie der für die auszuführende Operation relevanten Speicherzellen oder Zeiger der globalen Datenstruktur erstellt. Auf Basis dieser Kopie wird danach die eigentliche Operation vorbereitet, indem zunächst die passenden Zeiger in die oben vorbereiteten Daten eingefügt oder Rückgabewerte gespeichert werden. Eine Modifikation der globalen Datenstruktur durch andere Prozesse machen die in dieser Phase vorgenommenen Änderungen hinfällig. In der dritten Phase wird daher eine so genannte CAS-Operation (für *Compare-And-Swap* – Vergleich-Und-Austausch) durchgeführt. Diese atomare Operation vergleicht zwei Zeiger und tauscht bei Gleichheit einen der beiden Zeiger durch einen dritten Zeiger aus. Durch die CAS-Operation wird getestet, ob sich die globale Datenstruktur seit dem Zeitpunkt, in dem die lokale Kopie erstellt wurde, geändert hat. Ist dies nicht der Fall wird durch den Austausch eines Zeigers die gewünschte Änderung in der globalen Datenstruktur persistent gemacht. Falls sich die globale Datenstruktur aber geändert hat, muss die zweite Phase erneut ausgeführt werden.

Lock-freie Algorithmen haben viele Anwendungsgebiete: Zum einen eignen sie sich zum Einsatz in Anwendungen, die die Parallelität von Multi-Core-Architekturen ausnutzen wollen. Dazu gehören so unterschiedliche Anwendungen wie Model Checking [vdPW08] oder Echtzeit-3D-Spiele [Leo07]. Ein weiterer wichtiger Einsatzbereich sind Algorithmen, die keine zentralen

Locks benutzen dürfen. Ein Beispiel dazu sind lock-freie Garbage-Collections [GGH07, DMMJ01] oder auch lock-freie Hashtabellen [GGH05, Cli08], die zum Beipiel in Datenbanken oder Webservern Verwendung finden. Schließlich werden lock-freie Algorithmen auch in Betriebssystemen für Kernfunktionen wie zum Beispiel in der Implementierung für die Verwaltung von Prozess-Queues verwendet [HH01]. Da lock-freie Algorithmen mittlerweile auch in die Standardbibliotheken von Programmiersprachen wie Java integriert sind, ist ihre Verwendung in weiteren Anwendungsgebieten sehr wahrscheinlich.

Im nächsten Abschnitt wird mit Treibers Stack-Algorithmus [Tre86] ein einfacher lock-freier Algorithmus vorgestellt, der einer der ersten entwickelten lock-freien Algorithmen ist. Die verwendeten Grundprinzipien sind jedoch mit den neueren lock-freien Algorithmen identisch. Daher wird dieser Algorithmus in dieser Arbeit dazu benutzt, um die Anwendung der in den folgenden Kapiteln vorgestellten Verifikationstechniken zu verdeutlichen.

6.3 Treibers Stack-Algorithmus

Die CAS-Operation vergleicht einen lokalen Zeiger *Old* mit einem globalen Zeiger G. Wenn beide Zeiger identisch sind, wird G auf den Wert eines zweiten lokalen Zeigers *New* umgesetzt und die CAS-Operation wird erfolgreich beendet. Falls die beiden Werte *Old* und G jedoch ungleich sind, schlägt die CAS-Operation fehl und der globale Zeiger G bleibt unverändert.

Das CAS-Commando wird durch die folgende Prozedur formal spezifiziert:

```
CAS(Old, New; G, Success) {
    if*  G = Old then {
        G := New, Success := true;
    } else {
        Success := false;
    }
}
```
(6.1)

Die drei Zeiger *Old*, *New* und G werden der Operation als Parameter übergeben. Da nur der Zeiger G verändert werden muss, werden die Zeiger *Old* und *New* als Value-Referenzen übergeben. Zusätzlich zu den drei Zeigern wird noch der boolesche Wert *Success* der Prozedur als Reference-Parameter übergeben. Diese Variable wird genutzt um anzuzeigen, ob die Operation erfolgreich war (Rückgabewert gleich true) oder nicht (Rückgabewert gleich false). Der Prozedurrumpf besteht aus einer einzigen Fallunterscheidung, die testet, ob G gleich *Old* ist. In diesem Fall wird in einer parallelen Zuweisung die Variable G auf den Wert *New* gesetzt und die Variable *Success* auf den Wert true. Im anderen Fall wird die Variable *Success* auf den Wert false gesetzt. Durch Verwendung der Sternform des If-Operators und paralleler Zuweisungen wird sichergestellt, dass die Operation nur einen Schritt benötigt, also atomar ist.

Der Stack wird durch eine verlinkte Liste repräsentiert, die im Heap *Hp* gespeichert wird. Jede der Heap-Zellen besteht aus zwei Feldern. Das erste Feld wird für den Datenwert benutzt. Auf dieses Feld kann mit der Funktion .val

```
1  CPush(V; Top, Hp, UNew, USuccess) {
2      choose RefNew with RefNew ≠ Null ∧ ¬RefNew ∈ Hp in {
3          Hp := Hp ∪ {RefNew}, UNew := RefNew, USuccess := false;
4          Hp[UNew].val := V;
5          let UTop = Null in {
6              while ¬USuccess do {
7                  UTop := Top;
8                  Hp[UNew].nxt := UTop;
9                  CAS(UTop, UNew; Top, USuccess)
10             }
11         }
12     }
13 }
```

Abbildung 6.1: Deklaration des Push-Prozesses in KIV

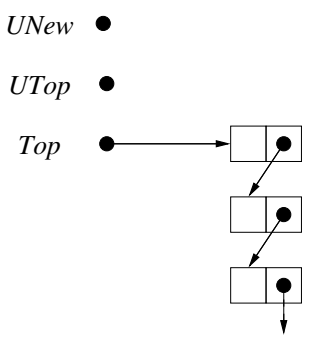
zugegriffen werden. Das zweite Feld enthält den Zeiger auf die nächste Zelle, auf den mit der Funktion .nxt zugegriffen werden kann. In diesem Feld kann auch der Null-Pointer Null stehen. Durch diesen Wert wird das Ende des Stacks markiert. Zusätzlich wird noch die Variable *Top* benutzt. Der in ihr enthaltene Zeiger markiert die oberste Zelle des Stacks. Falls der Stack leer ist, hat die Variable *Top* ebenfalls den Wert Null.

Die Spezifikation des Push-Algorithmus ist in Abbildung 6.1 dargestellt. Die Zeilennummern sind nur zur Erläuterung hinzugefügt. Sie werden nicht in KIV benötigt. Als Parameter bekommt ein Prozess *CPush* die Variable V als Value-Parameter. Diese Variable enthält den Wert, der zum Stack hinzugefügt werden soll. Der Reference-Parameter *Top* enthält den Zeiger auf die oberste Stack-Zelle und der Reference-Parameter *Hp* enthält den Heap, in dem der Stack gespeichert wird. Zusätzlich werden die beiden Variablen *UNew* und *USuccess* als Reference-Parameter übergeben.[1] Im Originalalgorithmus sind diese beiden Variablen lokal. Da es notwendig ist, ihren Werteverlauf über die Rely- und Guarantee-Eigenschaften zu beschreiben, wurden diese beiden Variablen als globale Variablen deklariert und dem Prozess als Parameter übergeben.

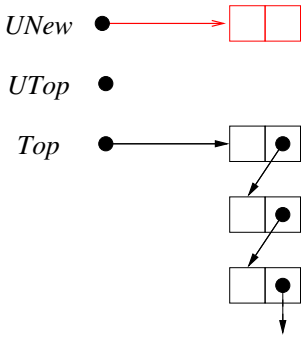
Die Konfiguration des Heaps und der wichtigen Variablen zu verschiedenen Zeitpunkten während des Programmablaufes von *CPush* ist in Abbildung 6.2 dargestellt. Abbildung 6.2(a) stellt den Zustand beim Aufruf von *CPush* dar. Im ersten Schritt alloziert der Algorithmus eine neue Zelle im Heap und speichert in der Variable *UNew* eine Referenz auf die neu allozierte Zelle (Zeile 2 und 3, bzw. Abbildung 6.2(b)). Zusätzlich wird der Wert von *USuccess* mit false initialisiert. Danach wird der in den Stack einzufügende Wert in das *val*-Feld der neu initialisierten Zelle geschrieben (Zeile 4, bzw. Abbildung 6.2(c)). In Zeile 5 wird dann die lokale Variable *UTop* initialisiert. Damit ist die erste Phase der Push-Prozedur abgeschlossen.

Die zweite Phase von *CPush* (Zeile 6 – 9) besteht aus einer Schleife, die iteriert wird, solange die Operation nicht abgeschlossen ist. Am Anfang einer

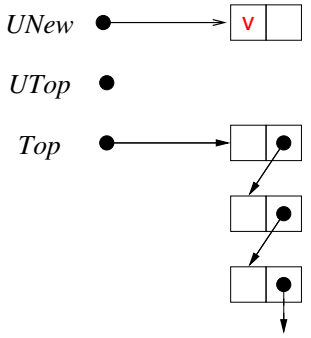
[1] Lokale Variablen des Push-Prozess beginnen mit U, die des Pop-Prozess mit O.



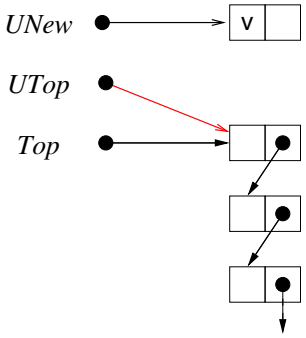
(a) Zeile 1



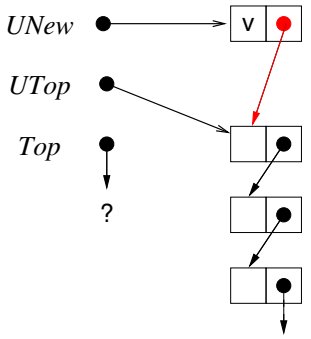
(b) Zeile 3



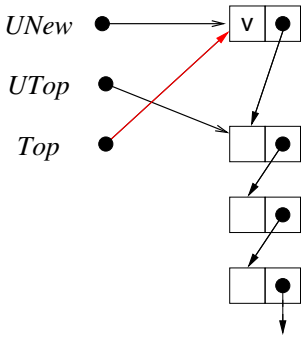
(c) Zeile 4



(d) Zeile 7



(e) Zeile 9 - Vor dem *CAS*



(f) Zeile 9 - Nach erfolgreichem *CAS*

Abbildung 6.2: Konfigurationen des Stacks während einer Push-Operation

Schleifeniteration wird von der Variable *Top* eine lokale Kopie in der Variable *UTop* erstellt (Zeile 7, bzw. Abbildung 6.2(d)). Anhand dieser Kopie wird später getestet, ob sich der Stack verändert hat oder nicht. Zunächst wird jedoch der *nxt*-Wert der allozierten Zelle *UNew* auf die Kopie des Top-Zeigers gesetzt (Zeile 8). Schließlich versucht der Algorithmus, die neu allozierte Zelle zum Stack hinzuzufügen (Zeile 9, bzw Abbildung 6.2(e)). Dazu wird die oben vorgestellte *CAS*-Operation benutzt. Durch einen Vergleich der beiden Variablen *Top* und *UTop* wird festgestellt, ob der Stack in den letzten beiden Schritten verändert wurde. Falls das nicht zutrifft, wird der Top-Zeiger auf die neu allozierte Zelle *UNew* und *USuccess* auf True gesetzt (siehe Abbildung 6.2(f)) Im anderen Fall bleiben die Variablen *Top* und *USuccess* unverändert und die zweite Phase wird erneut iteriert.

Der Pop-Algorithmus geht nach einem ähnlichem Prinzip vor. Der zusätzliche Reference-Parameter *O* wird zur Ausgabe des aus dem Stack gelesenen Wertes benutzt. Falls der Stack leer ist, soll *CPop* den speziellen Wert $\emptyset$ über die Variable *O* ausgeben. Im ersten Schritt werden die beiden lokalen Variablen *Lo* und *ONxt* initialisiert (Zeile 2). In *Lo* soll der gelesene Wert und in *ONxt* die Next-Referenz der obersten Stack-Zelle zwischengespeichert werden. Danach wird in Zeile 3 die Variable *OSuccess* initialisiert.

Die zweite Phase von *CPop* (Zeile 4 bis 15) ist deutlich umfangreicher als bei *CPush*. Wieder wird eine Schleife ausgeführt bis die Operation erfolgreich durchgeführt werden konnte. In Zeile 5 wird eine Kopie vom *Top*-Zeiger erstellt. In den Zeilen 6 bis 8 wird geprüft, ob der Stack leer ist und gegebenenfalls der Wert $\emptyset$ ausgegeben. Falls der Stack nicht leer ist, wird in den Zeilen 10 und 11 eine Referenz auf die zweite Stackzelle in der Variable *ONxt* und der Wert der obersten Stackzelle in der Variable *Lo* zwischengespeichert. In Zeile 12 wird mittels einer CAS-Operation versucht, die oberste Zelle aus dem Stack zu entfernen, indem der Top-Zeiger auf die zweite Zelle gesetzt wird. Falls die CAS-Operation erfolgreich ist, wird in Zeile 15 der in der lokalen Variable *Lo* zwischengespeicherte Ausgabewert an die Variable *O* übergeben.

Beide Algorithmen benutzen Zeiger, um Änderungen an der Stack-Datenstruktur festzustellen. Dabei kann das so genannte ABA-Problem auftauchen. Dieses Problem tritt auf, wenn eine Zelle dealloziert wird, zum Beispiel, durch ein zusätzliches *free*(*OTop*) am Ende der Prozedur *CPop*. Damit kann die Zelle *OTop* freigegeben sein, während eine lokale Kopie des Zeigers *OTop* in einem anderen Prozess *P* immer noch auf diese Zelle verweist. Durch diese Freigabe kann die gleiche Zelle danach durch einen *CPush*-Prozess neu alloziert und mit anderem Inhalt wieder oben in den Stack integriert werden, so dass *Top* auf diese Zelle verweist. Beim Ausführen der CAS-Operation kann der Prozess *P* nun nicht mehr die Änderung des Stacks feststellen, da die lokale Kopie von *Top* identisch mit dem aktuellen Zeiger *Top* ist. *P* kann in diesem Fall trotz einer Änderung des Stacks ein erfolgreiches CAS ausführen und so den Stack basierend auf veralteten Informationen ändern, was zu einem Fehler führen kann.

Es gibt mehrere Techniken um dieses Problem zu vermeiden:

- Eine Möglichkeit ist die Verwendung einer Garbage-Collection. Dabei wer-

```
1   CPop(; Top, Hp, OTop, OSuccess, O) {
2       let Lo = ∅, ONxt = NULL in {
3           OSuccess := false;
4           while ¬OSuccess do {
5               OTop := Top;
6               if OTop = NULL then {
7                   Lo := ∅;
8                   OSuccess := true;
9               } else {
10                  ONxt := Hp[OTop].nxt;
11                  Lo := Hp[OTop].val;
12                  CAS(OTop, ONxt; Top, OSuccess);
13              }
14          }
15          O := Lo
16      }
17  }
```

Abbildung 6.3: Deklaration des Pop-Algorithmus in KIV

den Zellen erst dann dealloziert, wenn kein Zeiger mehr auf diese verweist. In dieser Arbeit wird eine implizite Garbage-Collection angenommen, um das ABA-Problem zu vermeiden. Dies wird so realisiert, dass die globale Umgebung alle Zellen deallozieren kann, auf die kein Zeiger mehr zeigt.

- Eine andere Möglichkeit dieses Problem zu vermeiden, besteht darin, so genannte *Modification-Counters* einzusetzen. Dabei besitzt jede Zelle einen Zähler, der bei jeder Modifikation um eins erhöht wird. Um festzustellen, ob die Zelle nicht verändert wurde, muss neben dem Zeiger zusätzlich noch der Modification-Counter verglichen werden. Es ist möglich, lock-freie Algorithmen mit Modification-Counters zu verifizieren. Diese können in einem separaten Verfeinerungsschritt hinzugefügt werden (siehe z. B. [GC09]).

- Eine dritte Möglichkeit ist die Verwendung von so genannten *Hazard Pointern* [Mic04]. Dabei werden potenziell kritische lokale Zeiger von einem einzelnen Prozess P in einer so genannten *Hazard-Pointer-Liste* bekannt gegeben und von dieser Liste erst wieder gelöscht, wenn diese Referenz für P nicht mehr kritisch ist. Jeder Prozess besitzt eine solche Hazard-Pointer-Liste. Bevor eine zuvor freigegebene Speicherzelle erneut alloziert werden kann, wird überprüft, ob eine Referenz auf diese Zelle in einer der Hazard-Pointer-Listen existiert. Nur wenn dies nicht der Fall ist, darf diese Zelle erneut alloziert werden. Die von Herlihy et al. [HLM02] vorgestellte Methode zur Vermeidung des ABA-Problems verwendet eine im Prinzip gleiche Technik.

6.4 Weitere lock-freie Algorithmen

Inzwischen gibt es eine Vielzahl von lock-freien Algorithmen. Neben dem oben vorgestellten Treiber-Stack-Algorithmus gibt es noch einen lock-freien Stack-Algorithmus von Hendler, Shavit & Yerushalmi [HSY04]. Dieser Algorithmus benutzt so genannte Eliminationsarrays, mit denen er besonders unter hoher Last effizienter als der Treiber-Stack Algorithmus ist. Ein weiterer lock-freier Algorithmus für Stacks wurde von Herlihy et al. [HLMM05] vorgestellt. Dieser verwendet Guards zur Optimierung.

Für Queues als Datenstruktur ist der bekannteste lock-freie Algorithmus die Implementierung von Michael & Scott [MS96]. Für diesen Algorithmus gibt es zwei Optimierungen, die erste wurde von Fober [FOL01] und die andere von Doherty et al. [DGLM04] vorgestellt. Die Variante der von Doherty et al. beschriebenen Optimierung wird in Kapitel 9 vorgestellt und detailliert beschrieben. Eine weitere lock-freie Queue-Implementierung wurde von Herlihy & Wing [HW90] beschrieben. Diese ist zwar praktisch nicht relevant, aber aus theoretischer Sicht besonders schwierig zu untersuchen, da bei diesem Algorithmus keine Abstraktionsfunktion für die Datenstruktur angegeben werden kann. Neben einfachen Queues gibt es auch lock-freie Algorithmen für so genannte Dequeues (*double-ended-queues*), die das Hinzufügen bzw. Herausnehmen von Elementen an beiden Enden der Queue erlauben. Algorithmen hierfür wurden von Michael [Mic03] und Doherty et al. [DDG+04] vorgestellt.

Auch für andere Datenstruckturen gibt es mittlerweile einige lock-freie Algorithmen. Für verzeigerte Listen gibt es zwei verschiedene lock-freie Implementierungen [HHL+07, Har01]. Für lock-freie Hashtablellen wurden Algorithmen von Michael [Mic02], von Gao, Groote & Hesselink [GGH05], und von Click [Cli08] vorgestellt. Ein weiterer lock-freier Algorithmus von Afek et al. implementiert atomar zugreifbare Register [AAD+93].

Kapitel 7

Korrektheit lock-freier Algorithmen

In diesem Kapitel wird die Korrektheit von lock-freien Algorithmen untersucht und eine modulare Technik zu deren Verifikation vorgestellt. Dabei werden zunächst nur Sicherheitsaspekte im Sinn von Safety („Something bad never happens“) untersucht. Die Lebendigkeit (im Sinn von liveness) von lock-freien Algorithmen wird dann in Kapitel 8 behandelt.

In Abschnitt 7.1 wird zunächst ein formales Systemmodell vorgestellt, in dem lock-freie Algorithmen nebenläufig ausgeführt werden können. Die Korrektheit von lock-freien Algorithmen wird in dieser Arbeit mittels einer Verfeinerung von atomaren abstrakten Operationen durch die konkreten Operationen nachgewiesen. Dieser Ansatz wird in Abschnitt 7.2 genauer erläutert. Da klassische Verfeinerungstechniken üblicherweise nur isoliert ablaufende Programme betrachten, können diese Techniken nicht für lock-freie Algorithmen verwendet werden. Beim hier gewählten Ansatz werden modulare Techniken zum Nachweis der Verfeinerungsbeziehung verwendet. Das in Kapitel 4 vorgestellte Modularisierungstheorem ermöglicht den Beweis einfacher Sicherheitsaussagen, wie sie auch mit klassischen kompositionalen Beweistechniken (siehe z. B. [dRdBH$^+$01] oder [CC96]) gezeigt werden können. Verfeinerungseigenschaften können aber mit diesen Techniken nicht ausgedrückt werden. Daher wird in Abschnitt 7.3 ein Modularisierungstheorem beschrieben, mit dem sich der Nachweis einer Verfeinerungseigenschaft des Gesamtsystems auf den Nachweis von Verfeinerungseigenschaften der einzelnen Operationen zurückführen lässt. Die Grundkonzepte des Beweises für dieses Modularisierungstheorem und eine Beweisskizze werden in Abschnitt 7.4 vorgestellt. Um eine Verfeinerung zu zeigen, reicht das in Kapitel 2.5 vorgestellte Induktionsschema von ITL$^+$ allerdings nicht aus. Daher wird in Abschnitt 7.5 ein erweitertes Induktionsschema für den Beweis von allgemeinen Safety-Formeln erörtert, das sich auch im Verfeinerungsbeweis der Operationen anwenden lässt. Schließlich wird im letzten Abschnitt 7.6 die praktische Anwendung der in diesem Kapitel vorgestellten Modularisierungstechnik anhand der Fallstudie „Treiber-Stack“ beschrieben. Ein Teil der hier beschriebenen Ergebnisse wurde vom Autor bereits in [BSTR09] veröffentlicht.

7.1 Modell für lock-freie Algorithmen

Lock-freie Algorithmen werden üblicherweise zum Zugriff auf gemeinsame Datenstrukturen in parallelen Programmen eingesetzt. Ein System, welches lock-freie Algorithmen einsetzt, besteht nicht nur aus den lock-freien Operationen selbst, sondern ruft diese nur im Rahmen eines größeren Programms auf. Zur Verifikation von lock-freien Algorithmen muss daher dieses Szenario hinreichend berücksichtigt werden, das heißt, es muss ein formales Modell angegeben werden, in dem mehrere sequentielle Programme interleaved-parallel ausgeführt werden. Die einzelnen lock-freien Operationen werden von diesen sequentiellen Programmen immer wieder aufgerufen. Der Zugriff auf die lock-freien Datenstrukturen erfolgt ausschließlich über die entsprechenden lock-freien Operationen *COP*. Jedes der sequentiellen Programme ruft dabei eine Operation *COP* beliebig oft auf. Zwischen zwei derartigen Operationsaufrufen kann das sequentielle Programm beliebig viele Schritte durchführen, in denen nicht auf die lock-freie Datenstruktur zugegriffen wird.

Wie in den vorhergehenden Kapiteln wird das Gesamtsystem durch eine Spawn-Prozedur erzeugt. Allerdings werden von der Spawn-Prozedur nicht die einzelnen lock-freien Operationen, sondern n parallelgeschaltete Prozeduren *CSeq* erzeugt. Damit ergibt sich die folgende Spawn-Prozedur:

Definition 27 *(CSpawn).*

$$
\begin{array}{l}
\mathit{CSpawn}(n; \mathit{In}, \mathit{CS}, \mathit{Out})\ \{ \\
\quad \textbf{if } n = 0 \textbf{ then} \\
\quad\quad \mathit{CSeq}(0; \mathit{In}, \mathit{CS}, \mathit{Out}) \\
\quad \textbf{else} \\
\quad\quad \mathit{CSeq}(n; \mathit{In}, \mathit{CS}, \mathit{Out}) \parallel \mathit{CSpawn}(n-1; \mathit{In}, \mathit{CS}, \mathit{Out}) \\
\}
\end{array}
$$

□

Die Prozedur *CSpawn* erzeugt $n+1$ interleaved-parallel geschaltete sequentielle *CSeq*-Prozesse. Anstatt einer einzelnen generischen Variable V als Parameter werden drei verschiedene Variablen als Reference-Parameter verwendet: Die Variable *CS* enthält alle anderen für die Verifikation benötigte Daten. Zum Beispiel würde diese Variable für das „Treiber-Stack"-Beispiel unter anderen auch die konkrete Repräsentation des Stacks, also den *Top*-Zeiger und den Heap *Hp* enthalten. Für die Theorie wird für die Variable *CS* zunächst ein unspezifizierter Datentyp **Cstate** angenommen:

$$\mathit{CS} \ : \ \textbf{Cstate}$$

In der Variable *Out* werden die Ausgabewerte der lock-freien Operationen gespeichert, während die Variable *In* die Eingabewerte enthält. Da jeder Prozess seine eigenen Ein- und Ausgabewerte besitzt, werden diese Variablen durch die Funktionen

$$
\begin{aligned}
\mathit{In} &\ : \ \textbf{Nat} \to \textbf{Input} \\
\mathit{Out} &\ : \ \textbf{Nat} \to \textbf{Output}
\end{aligned}
$$

spezifiziert, die jeder Prozessnummer eine Ein- und Ausgabevariable zuordnet. Alle drei Variablen werden als globale Variablen spezifiziert, so dass diese bei der Verifikation berücksichtigt werden können.

Die sequentiellen Komponenten *CSeq* werden formal spezifiziert durch:

Definition 28 *(CSeq).*

$$
\begin{array}{l}
\mathit{CSeq}(i; \mathit{In}, \mathit{CS}, \mathit{Out}) \; \{ \\
\quad (\, \mathbf{skip} \; \vee \; \mathit{COP}(i, \mathit{In}; \mathit{CS}, \mathit{Out}))^* \\
\}
\end{array}
$$

□

Wie oben beschrieben, soll *CSeq* neben Aufrufen der lock-freien Operation auch interne Operationen ausführen. Als Parameter werden wieder die Variablen *In*, *CS* und *Out* benutzt. Zusätzlich wird die statische Variable i als Prozessnummer verwendet. Der Prozedurrumpf von *CSeq* beschreibt die Schritte, die ein solches sequentielles Programm ausführen kann: Entweder führt das Programm in einem Schritt interne Berechnungen aus, oder es wird eine neue Operation auf der lock-freien Datenstruktur gestartet. Die internen Berechnungsschritte werden durch einen **skip**-Schritt repräsentiert, da hierbei keine Variablen der lock-freien Datenstruktur geändert werden sollen. Durch den Stern-Operator wird sichergestellt, dass *CSeq* mehrere *COP*-Operationen und **skip**-Schritte in beliebiger Reihenfolge ausführen kann, aber auch nach jeder lock-freien Operation bzw. nach jedem **skip**-Schritt terminieren kann.

Zum Aufruf einer neuen Operation auf der lock-freien Datenstruktur wird die Prozedur *COP* benutzt. Diese Prozedur führt bei Aufruf eine der konkreten lock-freien Operationen aus. *COP* besitzt die gleichen Parameter wie *CSeq*. Allerdings wird die Variable *In* als Value-Parameter an *COP* übergeben, da der Eingabewert von *COP* nicht geändert werden darf.

7.2 Verifikationsansatz für lock-freie Algorithmen

Die in Kapitel 4 vorgestellte Modularisierungstechnik wie auch die klassischen Modularisierungstechniken sind geeignet, um einfache Eigenschaften wie Invarianten oder Verhaltensgarantien, die über einzelne Transitionen formuliert werden können, nachzuweisen. Wie anhand der Fallstudien in Kapitel 4 gezeigt wurde, können diese Modularisierungstechniken benutzt werden, um die Korrektheit klassischer Mutex-Algorithmen zur Prozesssynchronisierung, zu denen der Semaphor-Algorithmus gehört, zu beweisen. Dies gelingt, da die Korrektheit dieser Algorithmen sehr einfach durch die Anzahl der Prozesse, die sich in der kritischen Sektion befinden, ausgedrückt werden kann. Daher kann die Korrektheit dieser Algorithmen durch den Nachweis einer Invariante gezeigt werden. Der konkrete Zustand von gemeinsamen Datenstrukturen ist dabei irrelevant und muss bei der Verifikation dieser Algorithmen nicht berücksichtigt werden.

Dies ist bei lock-freien Algorithmen grundsätzlich anders. Die Korrektheit dieser Algorithmen basiert darauf, dass sie sich wie (atomare) Stack-Operationen

verhalten. Eine klassische Technik in der Informatik für den Nachweis der Korrektheit von Algorithmen ist die Verwendung von Datenverfeinerung (engl. „*data refinement*“ – siehe dazu z. B. [HHS86], [dRE98] oder [DB01]).

Datenverfeinerung betrachtet Datentypen. Ein Datentyp

$$DT = (S, In, Out, Init, (Op_i)_{i \in I})$$

besteht aus einer Zustandsmenge S, Eingaben In, Ausgaben Out, einer Menge von Anfangszuständen $Init \subseteq S$ und Operationen Op_i (jeweils indiziert mit $i \in I$), die mit Hilfe einer Eingabe einen Zustand modifizieren und eine Ausgabe liefern. Die Operationen werden als Relationen

$$Op_i \subseteq In \times S \times S \times Out$$

modelliert. In KIV werden dafür Prozeduren Op_i verwendet (die fehlende Betrachtung der Terminierung wird bei der Datenverfeinerung durch $\perp$-Elemente modelliert, siehe [DB01]). Datenverfeinerung betrachtet zwei Datentypen, einen abstrakten Datentyp ADT (mit abstrakten Zuständen AS) und einen konkreten Datentyp CDT (mit konkreten Zuständen CS). Beide besitzen dieselbe Zahl von Operationen und dieselben In und Out Mengen.

Beispiel 6 *Treiber-Stack.*
Ein Beispiel für eine solche Verfeinerung wäre folgender Fall: Ein abstrakter Stack wird durch den in Kapitel 6.3 vorgestellten Treiber-Stack verfeinert. Die abstrakte Datenstruktur ADT würde durch

$$ADT = (AS, In, Out, AInit, (AOp_i)_{i \in I})$$

definiert, mit:

- AS sind dabei alle Zustände, die der abstrakte Stack annehmen kann.
- $(AOp_i)_{i \in I}$ enthält genau die beiden Operationen $AOp_1 = APush$ und $AOp_2 = APop$ mit $I = \{1, 2\}$.
- In enthält die Eingabewerte für $APush$, das heißt, die Datenwerte, die auf den Stack gelegt werden sollen.
- Out Enthält die Ausgabewerte von $APop$, das heißt, die Datenwerte, die vom Stack genommen werden.
- $AInit$ enthält als Initialwert den leeren Stack.

Die konkrete Datenstruktur CDT wird durch

$$CDT = (CS, In, Out, CInit, (COp_i)_{i \in I})$$

definiert, mit:

- CS sind alle konkreten Zustände, das heißt, alle Zustände, die durch die verschiedenen Belegungen der Variablen Top und Hp beschrieben werden.

- $(COp_i)_{i \in I}$ enthält genau die beiden konkreten Operationen $COp_1 = CPush$ und $COp_2 = CPop$ mit $I = \{1, 2\}$.
- $CInit$ enthält alle Zustände mit $Top = \textsc{Null}$.

Dabei sind die Eingaben In und die Ausgaben Out in ADT und CDT identisch. □

Es wird angenommen, dass beide Datentypen in einem beliebigen sequentiellen Kontext verwendet werden, der

1. zunächst einen Initialzustand $Init$ herstellt,
2. eine Folge von Operationen AOp aufruft und
3. den Zustand (AS bzw. CS) außerhalb der Operationen nicht modifiziert.

Der Kontext kann also sowohl mit ADT als auch mit CDT verwendet werden und beobachtet nur die Eingaben In und die Ausgaben Out. Eine Datenverfeinerung ist korrekt, wenn für einen Beobachter jeder Ablauf konkreter Operationen auch mit abstrakten Operationen möglich ist („Substitutivitätsprinzip“). Ein Beobachter, der Abläufe mit abstrakten Operationen erwartet, bekommt „richtige“ Abläufe, wenn eine Implementierung verwendet wird, die stattdessen nur konkrete Operationen aufruft.

Da der Kontext den Zustand des Datentyps nicht verändert, genügt es, Folgen von konkreten Operationen COP und abstrakten Operationen AOP zu betrachten. Die Korrektheit von Datenverfeinerung kann wie folgt definiert werden [DB01, dRE98]:

Eine Datenverfeinerung von ADT zu CDT ist korrekt, wenn es zu jeder Folge $i_1, \ldots, i_n \in I$ und jeder Folge

$$cs_0, \ldots, cs_n \quad \text{mit}$$
$$cs_0 \in CInit \text{ und } cs_1, \ldots, cs_n \in CS$$
$$\text{und } COP_{i_1}(in_1, cs_0, cs_1, out_1), \ldots, COP_{i_n}(in_n, cs_{n-1}, cs_n, out_n)$$

eine Folge

$$as_0, \ldots, as_n \quad \text{mit}$$
$$as_0 \in AInit \text{ und } as_1, \ldots, as_n \in AS$$
$$\text{und } AOP_{i_1}(in_1, as_0, as_1, out_1), \ldots, AOP_{i_n}(in_n, as_{n-1}, as_n, out_n)$$

mit denselben Eingaben in_j und Ausgaben out_j gibt.

In der Temporallogik ITL$^+$ lässt sich dies für den konkreten sequentiellen Prozess $CSeq$ und einen analog definierten abstrakten Prozess $ASeq$ als

$$CSeq(In; CS, Out),\ \Box\ CS' = CS'',\ Init(CS) \tag{7.1}$$
$$\vdash\ \exists\ AS.\ ASeq(In; AS, Out)$$

ausdrücken. Dabei müssen die Variablen *In* und *Out* in den konkreten und den abstrakten Operationen gleich sein. Außerdem wird hier verallgemeinert, indem auch unendliche Folgen von Aufrufen betrachtet werden. Mit $\Box\ CS' = CS''$ wird angenommen, dass die Umgebung den konkreten Zustand *CS* nicht verändert.

Für die beschriebenen nebenläufigen Algorithmen ist die Betrachtung eines sequentiellen Kontextes natürlich nicht ausreichend. Formal wird daher in dieser Arbeit als Kontext das in Abschnitt 7.1 vorgestellte nebenläufige Modell für lock-freie Algorithmen *CSpawn* benutzt. Für die parallelen Algorithmen wird dann eine nebenläufige Verfeinerung mit

$$\begin{aligned} &\mathit{CSpawn}(n; \mathit{In}, \mathit{CS}, \mathit{Out}),\ \Box\ R(\mathit{CS}', \mathit{CS}''),\ \mathit{Init}(\mathit{CS}) \\ \vdash\ &\exists\ \mathit{AS}.\ \mathit{ASpawn}(n; \mathit{In}, \mathit{AS}, \mathit{Out}) \end{aligned} \tag{7.2}$$

gezeigt. Obwohl diese Sequenz syntaktisch der Sequenz (7.1) ähnelt, ist sie doch deutlich schwieriger zu beweisen, da die einzelnen Operationen hier nicht isoliert ablaufen, sondern sich gegenseitig beeinflussen können. Der Nachweis, dass diese wechselseitigen Beeinflussungen keine Auswirkung auf die Korrektheit der einzelnen Operationen haben, stellt bei lock-freien Algorithmen die Hauptschwierigkeit bei der Verifikation dar. Der hier gezeigte modulare Ansatz zielt darauf ab, mit dem Rely-Guarantee Paradigma den Beweis der Formel (7.2) auf Komponentenbeweise zu reduzieren. Der Beweis wird dabei mit einem geeigneten Modularisierungstheorem auf den Beweis von

$$\begin{aligned} &\mathit{COP}_i(\mathit{In}; \mathit{CS}, \mathit{Out}),\ \Box\ R_i(\mathit{CS}', \mathit{CS}''),\ I(\mathit{CS}) \\ \vdash\ &\exists\ \mathit{AS}.\ \mathit{AOP}_i(\mathit{In}; \mathit{AS}, \mathit{Out}) \end{aligned} \tag{7.3}$$

reduziert. Das heißt, zum Beweis von (7.2) müssen nur Beweise über einzelne Operationen geführt werden. Für die Umgebung müssen dabei natürlich geeignete Annahmen R_i getroffen werden, die das Verhalten der parallel ablaufenden Operationen mit einschließen. Außerdem muss berücksichtigt werden, dass der Initialzustand möglicherweise schon von anderen Operationen verändert wurde. Daher kann zu Beginn der Operation statt der Initialbedingung *Init* nur eine Invariante *I* angenommen werden. Dass weder die Invariante noch die lokalen Umgebungsannahmen R_i von anderen Operationen verletzt werden, muss dabei ebenfalls mit dem Modularisierungstheorem gezeigt werden.

Bei dieser Definition von Datenverfeinerung ist zu beachte, dass ein konkretes und ein abstraktes Intervall die gleiche Länge haben müssen, damit die Verfeinerungsbeziehung gilt.[1] Dies lässt sich durch das Einfügen zusätzlicher **skip***-Formeln erreichen, wie im nächsten Abschnitt gezeigt wird.

7.2.1 Definition der abstrakten Operationen

Die abstrakten Operationen *ASpawn* und *CSpawn* können wie die konkreten Operationen in Abschnitt 7.1 definiert werden:

[1] Dies folgt aus der Definition von $\vdash$: Die Formel $\varphi \vdash \psi$ kann kann nicht erfüllt sein, wenn ψ und φ Intervalle unterschiedlicher Länge beschreiben.

Definition 29 *(ASpawn).*

$$
\begin{array}{l}
ASpawn(n; In, AS, Out)\ \{ \\
\quad \textbf{if } n = 0 \textbf{ then} \\
\quad\quad ASeq(0; In, AS, Out) \\
\quad \textbf{else} \\
\quad\quad ASeq(n; In, AS, Out) \parallel ASpawn(n-1; In, AS, Out) \\
\}
\end{array}
$$

□

Die Struktur von *ASpawn* ist die gleiche wie bei *CSpawn*. Anstelle der Variable *CS*, welche die konkrete Datenstruktur enthält, wird in *ASpawn* die Variable *AS* für die abstrakte Datenstruktur benutzt. Für die abstrakten sequentiellen Komponenten *ASeq* wird die folgende Spezifikation benutzt:

Definition 30 *(ASeq).*

$$
\begin{array}{l}
ASeq(i; In, AS, Out)\ \{ \\
\quad (\textbf{skip} \vee AOP(i, In; CS, Out))^* \\
\}
\end{array}
$$

□

Auch diese Prozedur hat die gleiche Struktur wie die konkreten sequentiellen Komponenten *CSeq*.

Schließlich muss noch eine entsprechende abstrakte Operation *AOP* definiert werden:

Definition 31 *(AOP).*

$$
\begin{array}{l}
AOP(; AS)\ \{ \\
\quad \textbf{skip}^*; \\
\quad AS := atomicAOp(AS); \\
\quad \textbf{skip}^* \\
\}
\end{array}
$$

□

Die Funktion *atomicAOp* definiert die eigentliche abstrakte Operation. Sie wird üblicherweise algebraisch definiert. Die **skip***-Formeln vor und nach der atomaren Operation lassen die Datenstruktur unverändert und dienen dazu, ein Intervall mit der gleichen Länge des konkreten Intervalls zu erzeugen.

7.2.2 Abstraktionsfunktionen

Mit der Formel (7.2) wird die Korrektheit von lock-freien Algorithmen ausgedrückt. Allerdings ergeben sich aus dem Existenzquantor einige Schwierigkeiten bei der Konstruktion und Anwendung eines Modularisierungstheorems mit dem diese Formel gezeigt werden kann. Zum einen wird für den Beweis der Formel (7.3) Induktion benötigt, da man davon ausgehen muss, dass weder COP_i noch AOP_i in allen Fällen terminieren. In Abschnitt 7.5 wird eine Technik vorgestellt,

mit der aus den Sicherheitseigenschaften im Sukzedent einer Sequenz eine Induktionshypothese generiert werden kann. Existenzquantifizierte Formeln sind aber im Allgemeinen keine Sicherheitseigenschaften (siehe z.B. [JT96]). Daher lässt sich die in Abschnitt 7.5 beschriebene Technik nicht zum Beweis der Sequenz (7.3) verwenden.

Ein Lösungsansatz zum Beweis einer solchen Formel wurde von Cohen & Lamport [CL98] für die Temporallogik TLA beschrieben. Dabei wird ausgenützt, dass in TLA Formeln in der TLA-Normalform $[\Box\ \mathit{Trans}]_{\underline{V}}$ spezifiziert werden. Diese Normalform beschreibt alle möglichen Transitionen *Trans* des Systems bzw. der Formel (zusätzlich ist auch eine Stottertransition möglich, das heißt, das System verändert sich nicht). Der Beweis einer Datenverfeinerung kann dabei auf einen Beweis über einzelne Transitionen reduziert werden. Dabei muss für jede konkrete Transition gezeigt werden, dass es zu jedem abstrakten Ausgangszustand einen abstrakten Nachfolgezustand mit einer zugehörigen abstrakten Transition gibt. Dieser abstrakte Nachfolgezustand muss dem Nachfolgezustand der konkreten Transition entsprechen. Um diesen Ansatz zu verwenden, müssen allerdings die Systeme mittels Normalform (d. h. einer Menge von Transitionen) spezifiziert werden. Da es aber ein Ziel dieser Arbeit ist, Systeme durch eine Spezifikationssprache zu modellieren, die realen Programmiersprachen gleicht, ist diese Art, Systeme zu spezifizieren nicht für den hier verfolgten Ansatz geeignet.

Ein anderer Lösungsansatz ist das von Moszkowski [Mos00] beschriebene Axiom:

$$\vdash \mathit{inf} \wedge (\forall\ u.\exists\ V.(V = u \wedge P))^* \rightarrow \forall\ u.\exists\ V.(V = u \wedge P^*)$$

Damit können auf unendlichen Intervallen (unendliche Intervalle werden durch die Formel *inf* spezifiziert) existenzquantifizierte dynamische Variablen V in die einzelnen Intervallabschnitte des Stern-Operators „gezogen“ werden. Dabei muss aber zu Beginn jedes Abschnitts V einen beliebigen Wert u annehmen können. Aus diesem Axiom lässt sich eine Regel für While-Schleifen ableiten. Experimente zeigen aber, dass dies die Beweise deutlich komplexer macht, weshalb hier ein anderer Ansatz gewählt wurde.

Zusätzlich zur Generierung einer Induktionshypothese besteht aber noch ein weiteres Problem: Es ist nicht möglich, den Existenzquantor aus einem interleaved-parallelen Programm zu „liften“, das heißt, im allgemeinen gilt nicht:

$$(\exists\ AS.\varphi) \parallel (\exists\ AS.\psi) \vdash \exists\ AS.(\varphi \parallel \psi) \tag{7.4}$$

Aber genau eine solche Eigenschaft ist notwendig, damit man mit den Komponentenbeweisen (7.3) die globale Eigenschaft (7.2) zeigen kann.

Um diese beiden Probleme zu lösen, wird der Ansatz auf Systeme eingeschränkt, für die eine Abstraktionsfunktion

$$\mathit{absf} : CS \rightarrow AS$$

angegeben werden kann, so dass gilt:

$$\begin{aligned} & \mathit{COP}_i(\mathit{In}; CS, \mathit{Out}),\ \Box\ R_i(CS', CS''),\ I(CS) \\ \vdash\ & \mathit{AOP}_i(\mathit{In}; \mathit{absf}(CS), \mathit{Out}) \end{aligned} \tag{7.5}$$

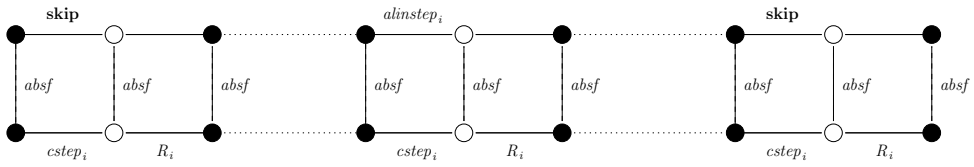


Abbildung 7.1: Beziehung zwischen einem $COP(i, \dots)$-Ablauf und einem $AOP(i, \dots)$-Ablauf

Das heißt, die quantifizierte Variable AS wird durch die Werte der Abstraktionsfunktion $absf(CS)$ instanziert. Nun kann die in Abschnitt 7.5 beschriebene Technik angewandt werden, um aus AOP_i eine Induktionshypothese zu generieren. Auch das mit der Formel (7.4) beschriebene „lifting-Problem“ wird mit diesem Ansatz gelöst. Als Gesamteigenschaft zum Beweis der Korrektheit von lock-freien Algorithmen ergibt sich damit:

$$\begin{aligned} & CSpawn(n; In, CS, Out),\ \Box\ R(CS', CS''),\ Init(CS) \\ \vdash\ & ASpawn(n; In, absf(CS), Out) \end{aligned} \tag{7.6}$$

Für fast alle lock-freien Algorithmen kann eine solche Abstraktionsfunktion *absf* angegeben werden. Der einzige bekannte lock-freie Algorithmus, für den dies nicht möglich ist, ist der lock-freie Queue-Algorithmus von Herlihy & Wing [HW90], der jedoch kaum praktische Relevanz hat.

Wenn man die in Formel (7.5) angegebene lokale Verfeinerungsbedingung für eine Komponente i näher betrachtet, so wird damit die in Abbildung 7.1 dargestellte Beziehung gezeigt. Die Anweisungen des konkreten Programms werden durch die $cstep_i$-Schritte repräsentiert, die Umgebungsschritte durch R_i. Jeder Zustand eines Ablaufes der konkreten Operation steht mit einem entsprechenden Ablauf der abstrakten Operation über die Abstraktionsfunktion in Beziehung. Der abstrakte Ablauf besteht dabei bis auf einen Schritt (*alinstep* – dies entspricht der Anweisung $AS := atomicAOp(AS)$ in AOP) nur aus **skip**-Schritten (Stotterschritten). Das heißt alle Änderungen der Operation an der abstrakten Datenstruktur müssen in *alinstep* stattfinden. Da der konkrete Ablauf in allen Zuständen über die Abstraktionsfunktion *absf* mit dem abstrakten Ablauf übereinstimmen muss, darf auch die konkrete Datenrepräsentation nur in diesem Schritt verändert werden. Informell entspricht damit diese gezeigte Eigenschaft auch dem Korrektheitskriterium *Linearisierbarkeit* [HW90]. Die Linearisierbarkeit eines Algorithmus wird üblicherweise dadurch nachgewiesen, indem gezeigt wird, dass jede Operation einen so genannten *Linearisierungspunkt* besitzt. Alle Änderungen einer Operation an der logischen Datenstruktur müssen in einem einzelnen Schritt – dem Linearisierungspunkt – geschehen. Damit entspricht *alinstep* genau dem Linearisierungspunkt.

7.3 Modularisierungstheorem für Verfeinerung

Mit den im vorhergehenden Abschnitt beschriebenen Vorüberlegungen lässt sich das folgende Modularisierungstheorem konstruieren:

Satz 5 *(Modulares Verfeinerungstheorem).*
Seien *CSpawn*, *COP*, *ASpawn* und *AOP* wie in den Abschnitten 7.1 und 7.2 spezifiziert, und sei I ein einstelliges Prädikat und G, R, G_i und R_i zweistellige Prädikate über *Cstate*. Wenn für alle $i = 1, \ldots, n$ gilt:

1. $[COP(i, In; CS, Out)]_{In,CS,Out}, I(CS) \vdash R_i(CS', CS'') \xrightarrow{+} G_i(CS, CS')$
2. $G_i(CS, CS') \wedge I(CS) \rightarrow G(CS, CS') \wedge \bigwedge_{j \in \{1..n\} \wedge j \neq i} R_j(CS, CS')$
3. $R_i(CS, CS') \wedge R_i(CS', CS'') \wedge I(CS') \wedge I(CS') \rightarrow R_i(CS, CS'')$
4. $R(CS, CS') \wedge I(CS) \rightarrow \bigwedge_{j \in \{1..n\}} R_j(CS, CS')$
5. $G_i(CS, CS') \wedge I(CS) \rightarrow I(CS')$
6. $R(CS', CS'') \wedge I(CS') \rightarrow I(CS'')$
7. $G_i(CS, CS)$
8. $Init(CS) \rightarrow I(CS)$
9. $[COP(i, In; CS, Out)]_{In,CS,Out}, \Box\, R_i(CS', CS''), \Box\, (I(CS) \wedge I(CS'))$
   $\vdash [AOP(i, In; absf(CS), Out)]_{In,CS,Out}$

gilt, dann gilt auch:

$$[CSpawn(n; In, CS, Out)]_{In,CS,Out}, \Box\, R(CS', CS''), Init(CS)$$
$$\vdash [ASpawn(n; In, absf(CS), Out)]_{In,CS,Out}$$

□

Die Beweisverpflichtungen 1. - 6. des Theorems sind identisch zu den Beweisverpflichtungen des Modularisierungstheorems aus Kapitel 4.2. Zusätzlich müssen noch die Beweisverpflichtungen 7. bis 9. gezeigt werden. In der Beweisverpflichtung 7. muss nachgewiesen werden, dass alle lokalen Garantien reflexiv sind. Das heißt, die lokalen Garantien müssen auch in Schritten gelten, in denen der Zustand *CS* sich nicht ändert. Diese Bedingung ist notwendig, damit die Garantien auch in den **skip**-Schritten der sequentiellen Komponenten $CSeq_i$ erfüllt werden. Zusätzlich wird in Bedingung 8. gezeigt, dass aus der Initialbedingung *Init* die Invariante *I* folgt. In der Beweisverpflichtung 9. wird schließlich gezeigt, dass jeder Ablauf einer konkreten Operation *COP* auch ein Ablauf der abstrakten Operation *AOP* bezüglich der Abstraktionsfunktion *absf* ist. Diese Formel entspricht der im vorhergehenden Abschnitt beschriebenen Beweisverpflichtung für die einzelnen Operationen (Formel (7.5)). Allerdings darf die Vorbedingung dieser Beweisverpflichtung noch verstärkt werden, indem man annimmt, dass die Invariante in jedem Grund- und Zwischenzustand gilt. Als Folgerung aus diesen Beweisverpflichtungen ergibt sich dann die Gesamteigenschaft, dass jeder Ablauf des konkreten Systems *CSpawn* auch ein bezüglich der Abstraktionsfunktion *absf* passender Ablauf des abstrakten Systems *ASpawn* ist. Diese Eigenschaft ist identisch mit der im vorhergehenden Abschnitt motivierten Gesamteigenschaft (Formel (7.6)).

7.4 Beweis des modularen Verfeinerungstheorems

Im Folgenden wird der Beweis des Verfeinerungstheorems beschrieben. Zunächst wird in Abschnitt 7.4.1 ein Überblick über den Beweis gegeben. In den Abschnitten 7.4.2 bis 7.4.4 werden wichtige für den Beweis benötigte Lemmas erläutert. Schließlich wird in Abschnitt 7.4.5 der eigentliche Beweis des Verfeinerungstheorems beschrieben.

7.4.1 Grundidee für den Beweis des Verfeinerungstheorems

Zunächst wird das Problem vereinfacht, indem nur Prozeduren betrachtet werden, die nicht durch das Verhalten der Umgebung gestört werden können. Diese Annahme ist für nahezu keinen parallelen Algorithmus erfüllt und genügt auch nicht für den Beweis des Modularisierungstheorems, erlaubt aber die Grundidee des Beweises zu skizzieren. Wenn man nur Prozeduren *CSeq* betrachtet, die sich gegenseitig nicht beeinflussen, ist für den modularen Beweis der Gesamteigenschaft (Formel (7.6)) zunächst kein spezielles Modularisierungstheorem nötig. Informell soll gezeigt werden, dass eine Menge von parallel ausgeführten konkreten Operationen $CSeq_i(CS)$ die gleichen Abläufe erzeugt wie die gleiche Anzahl von abstrakten atomaren Operationen $ASeq_i(absf(CS))$. Vereinfacht kann diese Gesamteigenschaft durch die folgende Sequenz ausgedrückt werden:[2]

$$\begin{aligned} & CSeq_1(CS) \parallel \ldots \parallel CSeq_n(CS) \\ \vdash\ & ASeq_1(absf(CS)) \parallel \ldots \parallel ASeq_n(absf(CS)) \end{aligned} \tag{7.7}$$

Durch die in Kapitel 4.1.1 vorgestellte Kompositionalitätseigenschaft des Interleavingoperators lässt sich diese Sequenz für jede Komponente i auf die folgenden lokalen Verfeinerungseigenschaften reduzieren:

$$CSeq_i(CS) \vdash ASeq_i(absf(CS)) \tag{7.8}$$

Wenn man diese lokalen Eigenschaften weiter auf die einzelnen lokalen Operatoren reduziert, erhält man für alle Komponenten i die folgende Beweisverpflichtung:

$$COP_i(CS) \vdash AOP_i(absf(CS)) \tag{7.9}$$

Wenn man jedoch Prozesse betrachtet, die sich gegenseitig beeinflussen, bzw. die durch das Verhalten der Umgebung gestört werden können, lässt sich die Sequenz (7.9) nicht verwenden, da keinerlei Annahmen über das Umgebungsverhalten getroffen werden. Diese sind jedoch für einen solchen Beweis fast immer notwendig. Um diese Verfeinerungseigenschaft zum Beispiel für den in Kapitel 6.3 vorgestellten Push-Algorithmus zu zeigen, muss man davon ausgehen, dass die Umgebung keine Werte in der am Anfang neu allozierten Zelle überschreibt, bevor diese in den Stack integriert wurde. Wie in den vorangehenden Kapiteln sollen dafür die lokale Rely-Bedingungen R_i benutzt werden. Zusätzlich braucht man zumeist Wissen über den initialen Zustand zu Beginn

[2] In dieser und den folgenden Sequenzen dieses Abschnitts wurden die Variablen für die Ein- und Ausgabeparameter zugunsten der Lesbarkeit der Sequenzen ausgelassen.

der Operation. Im Modularisierungstheorem aus Kapitel 4 wurde dafür die Invariante I benutzt. Wenn man nun in der Sequenz (7.9) eine zusätzliche Umgebungsannahme $\Box\ R_i(CS', CS'')$ und eine Invariante $\Box\ I(CS)$ annimmt, erhält man die folgende Sequenz:

$$\begin{aligned} & COP_i(CS), \Box\ R_i(CS', CS''), \Box\ I(CS) \\ \vdash\ & AOP_i(absf(CS)) \end{aligned} \tag{7.10}$$

Diese Sequenz entspricht der Beweisprämisse 9. aus Satz 5. Aus dieser Sequenz kann die folgende Eigenschaft für die sequentiellen Komponenten *CSeq* abgeleitet werden:

$$\begin{aligned} & CSeq_i(CS), \Box\ R_i(CS', CS''), \Box\ I(CS) \\ \vdash\ & ASeq_i(absf(CS)) \end{aligned} \tag{7.11}$$

Wenn man diese Sequenz zur Anwendung der Kompositionalitätsregel benutzt, so kann damit insgesamt die Sequenz

$$\begin{aligned} & CSeq_1(CS) \wedge \Box\ R_1(CS', CS'') \wedge \Box\ I(CS) \\ \| & \ldots \\ \| & \ CSeq_n(CS) \wedge \Box\ R_n(CS', CS'') \wedge \Box\ I(CS) \\ \vdash & \ ASeq_1(absf(CS)) \parallel \ldots \parallel ASeq_n(absf(CS)) \end{aligned} \tag{7.12}$$

mit der Kompositionalität des Interleaving-Operators gezeigt werden.

Im vorgestellten Modularisierungstheorem für Interleaving (Kap. 4.3, Satz 3 auf Seite 54) wurde mit der Rely-Guarantee-Technik gezeigt, dass die Komponenten ihre Umgebungsannahme untereinander aufrecht erhalten. Die wesentliche Grundidee des Modularisierungstheorems für Verfeinerung ist daher, diese Rely-Guarantee-Technik zu erweitern, um die folgende Sequenz zu beweisen:

$$\begin{aligned} & CSeq_1(CS) \parallel \ldots \parallel CSeq_n(CS),\ \Box\ R(CS', CS''),\ I(CS) \\ \vdash\ & \ CSeq_1(CS) \wedge \Box\ R_1(CS', CS'') \wedge \Box\ I(CS) \\ & \parallel \ldots \\ & \parallel\ CSeq_n(CS) \wedge \Box\ R_n(CS', CS'') \wedge \Box\ I(CS) \end{aligned} \tag{7.13}$$

Damit können die lokalen Umgebungsannahmen (und die Invariante) unter den Interleaving-Operator „gezogen“ werden. Der Antezedent dieser Sequenz entspricht dem (expandierten) Antezedent der Folgerung des Verfeinerungstheorems (Satz 5 auf Seite 112), während der Sukzedent identisch mit dem Antezedent aus Formel (7.12) ist. Die beiden Sequenzen (7.13) und (7.12) ergeben zusammen die zu zeigende Gesamteigenschaft aus Formel (7.7).

Der Beweis gliedert sich in vier Teile:

- Im Abschnitt 7.4.2 wird mit der Rely-Guarantee Eigenschaft der konkreten Operation *COP* die Rely-Guarantee Eigenschaft der sequentiellen Komponente *CSeq* und des gesamten Spawn-Prozesses *CSpawn* gezeigt. Diese Eigenschaften werden für den Beweis der Sequenz (7.13) benutzt.

- In Abschnitt 7.4.3 wird mit dem Lemma 7 bewiesen, dass die lokalen Umgebungsannahmen unter dem Interleaving-Operator erhalten bleiben. Das Lemma 7 entspricht der oben beschriebenen Formel (7.13). Dies ist der zentrale Beweis des Modularisierungstheorems.

- In Abschnitt 7.4.4 wird mit Lemma 8 gezeigt, dass die abstrakte sequentielle Komponente *ASeq* durch die konkrete sequentielle Komponente *ASeq* verfeinert werden kann. Dies entspricht der Formel (7.12).

- In Abschnitt 7.4.5 wird schließlich das Modularisierungstheorem für Verfeinerung bewiesen.

7.4.2 Rely-Guarantee Eigenschaft von *CSpawn*

Die nebenläufigen Komponenten dürfen ihre lokalen Umgebungsannahmen gegenseitig nicht verletzen. Demzufolge müssen alle Komponenten ihre Rely--Guarantee-Bedingungen erfüllen. Um dies zu beweisen, wird das folgende Lemma benutzt:

Lemma 3 *(CSeq-rely-guarantee).*
Unter den Voraussetzung des modularen Verfeinerungstheorems (Satz 5) gilt für alle n:

$$\begin{aligned} & [CSeq(n; In, CS, Out)]_{In,CS,Out}, \\ & I(CS) \\ \vdash\ & R(n, CS', CS'') \overset{+}{\rightarrow} G(n, CS, CS') \end{aligned}$$

□

Beweis

Im ersten Schritt wird der Prozeduraufruf von *CSeq* mit der entsprechenden Spezifikation aus Kapitel 7.1 auf Seite 104 expandiert. Danach kann eine Induktionshypothese aus der $\overset{+}{\rightarrow}$-Formel im Sukzedent generiert werden. Die *-Formel im Programmrumpf von *CSeq* kann terminieren bzw. im nächsten Schritt entweder einen **skip**-Schritt oder eine *COP*-Operation ausführen. Falls die *-Formel terminiert, kann der Beweis trivialerweise geschlossen werden. Falls der nächste Schritt ein **skip**-Schritt ist, kann mithilfe der Beweisverpflichtung 7. des Modularisierungstheorems gezeigt werden, dass die Garantie in diesem Schritt erhalten bleibt, und der Beweis danach mit Hilfe der Induktionshypothese geschlossen werden. Falls eine *COP*-Operation aufgerufen wird, muss diese mit Hilfe der Beweisverpflichtung 1. des Modularisierungstheorems zur Rely-Guarantee-Bedingung

$$R(n, CS', CS'') \overset{+}{\rightarrow} G(n, CS, CS')$$

generalisiert werden. Mit einem symbolischen Ausführungsschritt kann nun gezeigt werden, dass die Garantiebedingung erhalten bleibt, falls die Rely-Bedingung erfüllt ist. Danach kann auch dieser Fall durch Induktionsanwendung geschlossen werden. □

Lemma 4 *(CSpawn-rely-guarantee).*
Unter der Voraussetzung des modularen Verfeinerungstheorems (Satz 5 auf Seite 112) gilt für alle n:

$$\begin{array}{l} [CSpawn(n; In, CS, Out)]_{In,CS,Out}, \\ I(CS) \\ \vdash Rto(n, CS', CS'') \xrightarrow{+} Gto(n, CS, CS') \end{array}$$

□

Beweis
Das Lemma kann mit dem Modularisierungstheorem für Interleaving (Satz 3, siehe Seite 54) bewiesen werden. Dazu wird der abstrakte Datentyp **State** durch den Produktdatentyp

$$(\mathbf{Nat} \to \mathbf{Input}) \times \mathbf{Cstate} \times (\mathbf{Nat} \to \mathbf{Output})$$

instanziert, was den Datentypen der drei Variablen *In*, *CS*, und *Out* entspricht. Die abstrakten Komponenten P aus Satz 3 werden durch die konkreten sequentiellen Komponenten *CSpawn* instanziert. Die Beweisverpflichtung 1. aus Satz 3 folgt aus Lemma 3 und die Beweisverpflichtungen 2. bis 5. aus Satz 3 folgen direkt aus den Beweisverpflichtungen 2. bis 5. des Verfeinerungstheorems (Satz 5).
□

Zusätzlich zu diesen beiden Lemmas sind noch zwei weitere Lemmas für die späteren Beweise notwendig. Zum einen kann man mit der zusätzlichen Voraussetzung, dass die Umgebungsannahme *Rto* immer erfüllt ist, zeigen, dass auch die Garantien *Gto* immer erfüllt sind:

Lemma 5 *(CSpawn-alw-guarantee).*
Unter der Voraussetzung des modularen Verfeinerungstheorems (Satz 5) gilt für alle n:

$$\begin{array}{l} [CSpawn(n; In, CS, Out)]_{In,CS,Out}, \\ \Box\ Rto(n, CS', CS''),\ I(CS) \\ \vdash \Box\ Gto(n, CS, CS') \end{array}$$

□

Beweis

Dieses Lemma lässt sich einfach beweisen, indem man zunächst aus der Always-Formel im Sukzedent eine Induktionshypothese generiert und mit Lemma 4 die Prozedur *CSpawn* durch die Rely-Guarantee-Formel

$$Rto(n, CS', CS'') \xrightarrow{+} Gto(n, CS, CS')$$

ersetzt. Danach kann der Beweis durch einen symbolischen Ausführungsschritt und Anwendung der Induktionshypothese geschlossen werden. □

Ebenso kann man zeigen, dass bei gleichen Voraussetzungen die Invariante immer erhalten bleibt:

Lemma 6 *(CSpawn-alw-inv).*
Unter den Voraussetzung des modularen Verfeinerungstheorems (Satz 5) gilt für alle n:

$$\begin{aligned} & [CSpawn(n; In, CS, Out)]_{In,CS,Out}, \\ & \Box\ Rto(n, CS', CS''),\ I(CS) \\ \vdash\ & \Box\ (I(CS) \wedge I(CS')) \end{aligned}$$

□

Beweis

Diese Sequenz lässt sich durch Instanzierung von Korollar 1 (siehe Seite 64) beweisen. Die Instanzierung erfolgt analog zum Beweis von Lemma 4. □

7.4.3 Lokale Umgebungsannahmen innerhalb des Interleavings

Die in Formel (7.13) auf Seite 114 informell dargestellte Formel kann durch die folgende Sequenz formalisiert werden:

Lemma 7 *(CSpawn-has-rely).* Unter den Voraussetzung des Verfeinerungstheorems (Satz 5 auf Seite 112) gilt für alle n:

$$\begin{aligned} & [CSpawn(n+1; In, CS, Out)]_{In,CS,Out}, \\ & \Box\ Rto(n+1, CS', CS''), \\ & I(CS) \\ \vdash\ & [CSeq(n+1; In, CS, Out)]_{In,CS,Out} \\ & \wedge\ \Box\ (R(n+1, CS', CS'') \wedge I(CS) \wedge I(CS')) \\ \|\ & [CSpawn(n; In, CS, Out)]_{In,CS,Out} \\ & \wedge\ \Box\ (Rto(n, CS', CS'') \wedge I(CS) \wedge I(CS')). \end{aligned}$$

□

Diese Formel beschreibt, dass *CSpawn* so expandiert werden kann, dass innerhalb jeder Komponente die lokale Umgebungsannahme immer erfüllt ist. Als Abkürzung wird im Folgenden für den Sukzedent dieser Formel der Bezeichner Δ benutzt.

Beweis
Der Beweis wird geführt, indem die Prozedur *CSpawn* im Antezedent zu Δ umgeformt wird. Die Seitenfälle, die dabei entstehen, werden durch Induktion bzw. Widerspruch bewiesen.

Zunächst wird Lemma 6 benutzt, um die Formel

$$\Box\ (I(CS) \wedge I(CS'))$$

als zusätzliche Voraussetzung in den Antezedent aufzunehmen. Danach kann die Prozedur *CSpawn* im Antezedent mit ihrer Spezifikation (siehe Definition 27 auf Seite 104) zu

$$\begin{aligned} & [CSeq(n+1; In, CS, Out)]_{In,CS,Out} \\ \|\ & [CSpawn(n; In, CS, Out)]_{In,CS,Out} \end{aligned}$$

expandiert werden. Damit ergibt sich die folgende noch zu beweisende Sequenz:

$$\begin{aligned} & [CSeq(n+1; In, CS, Out)]_{In,CS,Out} \\ \| \; & [CSpawn(n; In, CS, Out)]_{In,CS,Out}, \\ \Box \; & Rto(n+1, CS', CS''), \\ \Box \; & (I(CS) \wedge I(CS')) \\ \vdash \; & \Delta \end{aligned}$$

Nun wird für die rechte Komponente des Interleavings eine Fallunterscheidung durchgeführt, ob die Formel

$$\Box \; (Rto(n, CS', CS'') \wedge I(CS) \wedge I(CS')) \tag{7.14}$$

innerhalb der rechten Komponente des Interleavings gilt oder nicht.

Fall 1 Falls diese Formel (7.14) für die rechte Komponenten erfüllt ist, wird eine weiteren Fallunterscheidung durchgeführt, ob

$$\Box \; (R(n+1, CS', CS'') \wedge I(CS) \wedge I(CS')) \tag{7.15}$$

innerhalb der linke Komponente des Interleavings gilt oder nicht. Dabei ergeben sich wiederum zwei Fälle:

Fall 1.1 Für beide Komponenten des Interleavings gilt die Formel (7.15). Damit ergibt sich im Antezedenten genau die Formel Δ und der Fall kann sofort geschlossen werden.

Fall 1.2 In diesem Fall ist die Formel (7.15) innerhalb der rechten Komponente erfüllt, wird aber irgendwann im Ablauf der linken Komponente verletzt. Dies wird durch die folgende Sequenz ausgedrückt:

$$\begin{aligned} & [CSeq(n+1; In, CS, Out)]_{In,CS,Out} \\ & \wedge \; \Diamond \; (\neg \; R(n+1, CS', CS'') \vee \neg \; I(CS) \vee \neg \; I(CS')) \\ \| \quad & [CSpawn(n; In, CS, Out)]_{In,CS,Out} \\ & \wedge \; \Box \; (Rto(n, CS', CS'') \wedge I(CS) \wedge I(CS')), \\ \Box \; & Rto(n+1, CS', CS''), \\ \Box \; & (I(CS) \wedge I(CS')) \\ \vdash \; & \Delta \end{aligned}$$

Der Fall, in dem irgendwann in der linken Komponente die Invariante verletzt ist, kann mit der globalen Formel $\Box \; (Inv(CS) \wedge Inv(CS'))$ zu einem Widerspruch geführt werden. Dagegen ist der Fall schwieriger zu zeigen, in dem die lokale Rely-Bedingung R_{n+1} der linken Komponente irgendwann verletzt wird. Auch hier wird der Beweis durch Widerspruch geführt, indem gezeigt wird, dass die lokale Umgebungsbedingung weder durch die globale Umgebung noch durch *CSpawn* verletzt werden kann.

Zunächst wird Lemma 5 benutzt, um die Prozedur *CSpawn* in der rechten Komponente durch die Formel

$$\Box\ Gto(n, CS, CS')$$

zu ersetzen. Damit ergibt sich die folgende Sequenz (nicht mehr benötigte Formeln sind weggelassen):

$$\begin{array}{rl} & \Diamond\ \neg\ R(n+1, CS', CS'') \\ \|\ & \Box\ Gto(n, CS, CS'), \\ \Box\ & Rto(n+1, CS', CS'') \qquad \vdash \end{array}$$

Mit den Definitionen von *Gto* und *Rto* (siehe Seite 57) kann die Formel wie folgt umgeformt werden:

$$\begin{array}{rl} & \Diamond\ \neg\ R(n+1, CS', CS'') \\ \|\ & \Box\ G(n, CS, CS'), \\ \Box\ & R(n+1, CS', CS'') \qquad \vdash \end{array}$$

Dieser Fall kann mit Induktion über die Formel $\Diamond\ \neg\ R(n+1, CS', CS'')$ bewiesen werden. Nach einem symbolischen Ausführungsschritt kann die Induktionshypothese angewandt werden, wenn die Rely-Bedingung $R(n+1, CS', CS'')$ der linken Komponente in diesem Schritt nicht verletzt wurde. Falls die Rely-Bedingung jedoch verletzt wurde, kann der Widerspruch gezeigt werden, da sowohl die globale Umgebung als auch die rechte Komponente diese nicht verletzen können (da die Guarantee $G(n, CS, CS')$ über die Beweisverpflichtung 2. des Modularisierungstheorems auch die Rely-Bedingung der linken Komponente erhalten muss).

2. Fall In diesem Fall wird die Formel (7.8) in der rechten Komponente irgendwann verletzt. Dies führt zur folgenden Sequenz:

$$\begin{array}{rl} & [CSeq(n+1; In, CS, Out)]_{In,CS,Out} \\ \|\ & \quad [CSpawn(n; In, CS, Out)]_{In,CS,Out} \\ & \quad \wedge\ \Diamond\ (\neg\ Rto(n, CS', CS'') \vee \neg\ I(CS) \vee \neg\ I(CS')), \\ & \Box\ Rto(n+1, CS', CS''), \\ & \Box\ (I(CS) \wedge I(CS')) \\ \vdash\ & \Delta \end{array}$$

Zuerst werden die Prozeduren *CSeq* und *CSpawn* mit Lemma 3 bzw. Lemma 4 durch die Rely-Guarantee-Formeln

$$R(n+1, CS', CS'') \stackrel{+}{\rightarrow} G(n+1, CS, CS')$$

und

$$Rto(n, CS', CS'') \stackrel{+}{\rightarrow} Gto(n, CS, CS')$$

ersetzt. Dass die Invariante innerhalb der beiden Komponenten erfüllt ist, um die beiden Lemmas anwenden zu können, kann einfach durch die globale Formel $\Box\ (Inv(CS) \wedge Inv(CS'))$ gezeigt werden. Die neue zu beweisende Sequenz lautet:

$$\begin{aligned} & R(n+1, CS', CS'') \xrightarrow{+} G(n+1, CS, CS') \\ \| \quad & Rto(n, CS', CS'') \xrightarrow{+} Gto(n, CS, CS') \\ & \wedge \Diamond\ (\neg\ Rto(n, CS', CS'') \vee \neg\ I(CS) \vee \neg\ I(CS')), \\ \Box\ & Rto(n+1, CS', CS''), \\ \Box\ & (Inv(CS) \wedge Inv(CS')) \quad \vdash \end{aligned}$$

Wieder wird die Formel

$$\Diamond\ (\neg\ Rto(n, CS', CS'') \vee \neg\ I(CS) \vee \neg\ I(CS'))$$

benutzt, um eine Induktionshypothese zu generieren. Mit einem symbolischen Ausführungsschritt muss nun gezeigt werden, dass die Rely-Bedingung beider Komponenten nicht verletzt werden kann. Dieser Beweis ist im wesentlichen identisch mit dem Beweis des Lifting-Lemmas (Lemma 1 auf Seite 58). Im Unterschied dazu muss man jedoch einen Widerspruch zeigen, wenn die Invariante in der rechten Komponente verletzt ist. Dies lässt sich einfach mit der globalen Formel

$$\Box\ (I(CS) \wedge I(CS')$$

im Antezedent beweisen.

□

7.4.4 Verfeinerung der sequentiellen Komponenten

Da in den Voraussetzungen des Verfeinerungstheorems die Gültigkeit der Verfeinerungsbeziehung nur für eine einzelne Operation angenommen wird, muss diese Eigenschaft noch auf die sequentielle Komponente *CSeq* „geliftet“ werden, bevor sie für das Gesamtsystem gezeigt werden kann. Dies wurde in dem informellen Überblicksabschnitt (Abschnitt 7.4.1) durch die Sequenz (7.10) ausgedrückt. Das folgende Lemma formalisiert diese Sequenz:

Lemma 8 *(*CSeq-*verfeinert*-ASeq*).*
Unter der Voraussetzung des Verfeinerungstheorems (Satz 5 auf Seite 112) gilt für alle n:

$$\begin{aligned} & [CSeq(n; In, CS, Out)]_{In,CS,Out},\ \Box\ R(CS', CS''),\ I(CS) \\ \vdash\ & [ASeq(n; In, absf(CS), Out)]_{In,CS,Out} \end{aligned}$$

□

Beweis
Zuerst wird mit einer Fallunterscheidung gezeigt, dass die Invariante in allen Zuständen gilt. Das heißt, dass die Formel

$$\Box\ I(CS)$$

global gültig ist. Falls die Invariante irgendwann verletzt wird, kann der Widerspruch gezeigt werden, indem zuerst die Prozedur *CSeq* mit Lemma 3 durch die Rely-Guarantee-Formel

$$R(n, CS', CS'') \overset{+}{\rightarrow} G(n, CS, CS')$$

ersetzt wird. Danach kann mit Induktion über die Länge des Intervalls bis zur Verletzung der Invariante und durch symbolische Ausführung gezeigt werden, dass die Invariante in jedem Zustand erhalten bleibt.

Nun kann die Prozedur *CSeq* expandiert werden, so dass die neue Sequenz entsteht

$$\begin{aligned} & [(\,\mathbf{skip}\ \vee\ COP(i, In; CS, Out))^*]_{In,CS,Out}, \\ & \Box\ R(CS', CS''), \\ & \Box\ I(CS) \\ \vdash\ & [ASeq(n; In, absf(CS), Out)]_{In,CS,Out} \end{aligned}$$

Hier kann die konkrete Operation *COP* mit der Beweisverpflichtung 9. des Verfeinerungstheorems (siehe Seite 112) durch die abstrakte Komponente *AOP* ersetzt werden. Dies ist möglich, da die Rely R und die Invariante I auf dem gesamten Intervall gelten und damit auch in jedem Teilintervall des *-Operators. Nach dieser Ersetzung hat die Prozedur im Antezedent genau die gleiche Form wie der Rumpf von *ASeq* im Sukzedent. □

7.4.5 Beweis des Modularisierungstheorems für Verfeinerung

Schließlich kann das Verfeinerungstheorem von Seite 112 selbst bewiesen werden. Die dazu beweisende Sequenz ist:

$$\begin{aligned} & [CSpawn(n; In, CS, Out)]_{In,CS,Out},\ \Box\ R(CS', CS''),\ Init(CS) \\ \vdash\ & [ASpawn(n; In, absf(CS), Out)]_{In,CS,Out} \end{aligned}$$

Beweis

Der Beweis wird durch Induktion über die Variable n geführt.

Induktionsanfang: $n = 0$
Da $n = 0$ erzeugt die Expansion der Prozedur *CSpawn* einen einzelnen Prozess *CSeq*. Mit Lemma 8 kann *CSeq* durch ein *ASeq* ersetzt werden. Damit ist der Antezedent identisch zur Expansion von *ASpawn* mit $n = 0$ im Sukzedent.

Induktionsschritt: $n \to n+1$

Zuerst kann mit Lemma 7 die Prozedur *CSpawn* zu der Formel

$$\begin{aligned} & [CSeq(n+1; In, CS, Out)]_{In,CS,Out} \\ \wedge\ & \Box\ (R(n+1, CS', CS'') \wedge I(CS)) \\ \| \quad & [CSpawn(n; In, CS, Out)]_{In,CS,Out} \\ \wedge\ & \Box\ (Rto(n, CS', CS'') \wedge I(CS)). \end{aligned}$$

expandiert werden. Nun kann Lemma 8 benutzt werden, um den linken Teil des Interleavings durch die abstrakte Prozedur *ASeq* zu substituieren. Der rechte Teil des Interleavings kann mit der Induktionshypothese durch *ASpawn* ersetzt werden. Das Interleaving im Antezedent hat damit die folgende Form:

$$\begin{aligned} & [ASeq(n; In, absf(CS), Out)]_{In,CS,Out} \\ \| \ & [ASpawn(n; In, absf(CS), Out)]_{In,CS,Out} \end{aligned}$$

Eine Expansion der abstrakten Prozedur *ASpawn* im Sukzedent ergibt genau die gleiche Formel.

□

7.5 Beweis allgemeiner Safety-Eigenschaften und der Prefix-Operator

Der in Kapitel 2 vorgestellte Kalkül ist hauptsächlich dafür geeignet, um temporallogische Eigenschaften zu untersuchen. Mit dem dort vorgestellten Induktionsprinzip, kann man Formeln folgender Art beweisen:

$$Sys \vdash TLprop$$

Das heißt, alle Traces, die durch ein System *Sys* beschrieben werden, erfüllen eine Formel *TLprop*. Damit eine Induktionshypothese generiert werden kann, muss entweder *TLprop* eine Formel der Form $\Box\ \varphi$ bzw. φ **unless** ψ oder im Antezedent muss eine geeignete Lebendigkeitsformel wie zum Beispiel $\Diamond\ \varphi$ vorhanden sein (siehe Kapitel 2.5.1). Eine Formel dieser Form ist jedoch nicht in allen interessanten Systemeigenschaften gegeben. So kann zum Beispiel aus einer Verfeinerungseigenschaft, wie sie im vorherigen Abschnitt für den Beweis von Beweisverpflichtung 9. des Verfeinerungstheorems zu zeigen ist, keine Induktionshypothese generiert werden. Generell zeigt man eine Verfeinerung von Programmen durch Trace-Inklusion, das heißt, alle Abläufe eines konkreten Programms entsprechen einem Ablauf im abstrakten Programm. Dass ein konkretes Programm *CSys* ein abstraktes Programm *ASys* verfeinert, würde man in ITL$^+$ durch die folgende Sequenz ausdrücken:

$$CSys \vdash ASys$$

das heißt, alle Intervalle, die durch das konkrete Programm beschrieben werden, sind auch Intervalle des abstrakten Programms. Für den Beweis einer solchen Sequenz ist jedoch meistens Induktion nötig. Sollten *CSys* und *ASys* jedoch sequentielle Programme sein, ist die in Kapitel 2.5.1 beschriebene Technik nicht ausreichend, um einen Induktionsterm zu generieren.

Prinzipiell lässt sich die in Kapitel 2.5.1 beschriebene Technik jedoch so erweitern, dass aus jeder Sicherheitseigenschaft ein Induktionsterm generiert werden kann. Informell kann eine Sicherheitseigenschaft durch „Something bad never happens“ beschrieben werden, das heißt, es wird niemals ein unerwünschter Zustand erreicht. Der Begriff „Sicherheitseigenschaft“ (engl. *safety property*) wurde zum ersten Mal von Lamport [Lam79] informell beschrieben und in [LS85] erstmals formalisiert. Diese Formalisierung berücksichtigt jedoch nur stotteräquivalente Formeln. Eine allgemeinere Formalisierung von Sicherheitseigenschaften für nicht stotteräquivalente Formeln findet sich in [AS85]. Semantisch äquivalente Formalisierungen sind in [AS87] und [Sis94] beschrieben. Diese Formalisierung wird heute üblicherweise in der Literatur zur Definition von Sicherheitseigenschaften verwendet. Im Unterschied zu ITL$^+$ werden in den oben genannten Arbeiten nur unendliche Abläufe betrachtet. Allerdings kann die Formalisierung aus [Sis94] trotzdem auch für ITL$^+$ verwendet werden, indem man zur Unterscheidung, ob eine Formel eine Sicherheitseigenschaft ist oder nicht, nur unendliche Intervalle betrachtet. Um Sicherheitseigenschaften formal zu definieren, wird zunächst eine Definition für Intervallkonkatenation benötigt:

Definition 32 *(Intervallkonkatenation).*
Sei $I = (\sigma_{i_0}, \sigma'_{i_0}, \ldots, \sigma_{i_n})$ ein endliches Intervall und sei $J = (\sigma_{j_0}, \sigma'_{j_0}, \sigma_{j_1}, \ldots)$ ein endliches oder unendliches Intervall mit $\sigma_{i_0}(s) = \sigma_{j_0}(s)$ für alle statischen Variablen $s \in \mathbf{SV}$. Dann ist

$$I + J := (\sigma_{i_0}, \sigma'_{i_0}, \ldots, \sigma_{i_n}, \sigma'_{j_0}, \sigma_{j_1}, \ldots)$$

die *Konkatenation* von I und J. □

Man beachte, dass im konkatenierten Intervall $I + J$ der erste Zustand σ_{j_0} des Intervalls J entfällt und durch den letzten Zustand σ_{i_n} von I ersetzt wird. Damit wird sichergestellt, dass die ungestrichenen Grundzustände und die gestrichenen Zwischenzustände alternieren. Mit dieser Definition kann nun der Begriff Sicherheitseigenschaft wie folgt formal definiert werden:

Definition 33 *(Sicherheitseigenschaft).*
Eine Formel $\varphi \in \mathbf{E}_{Bool}$ ist genau dann eine *Sicherheitseigenschaft*, wenn für alle Intervalle I mit $|I| = \infty$ gilt:

$$\text{Falls für alle } n \in \mathbb{N}_0 \text{ ein Intervall } I_0 \text{ existiert mit } I|^n + I_0 \models \varphi$$

so gilt auch

$$I \models \varphi$$

Die Menge aller Sicherheitseigenschaften wird mit **SF** bezeichnet.

□

Eine Formel φ ist also genau dann eine Sicherheitseigenschaft, wenn für alle

unendlichen Intervalle I gilt: Wenn für jedes Präfix von I eine geeignete Fortsetzung I_0 existiert, so dass φ auf dem Intervall $I + I_0$ gilt, dann gilt φ auf dem Intervall I.

Im Zusammenhang mit Sicherheitseigenschaften ist folgende Eigenschaft nützlich: Der Punkt, in dem man ein Intervall I durch ein neues Intervall I_0 ergänzt, kann immer nach vorne verlegt werden:

Lemma 9. Sei I ein Intervall, $\varphi \in \mathbf{E}_{Bool}$ eine Formel und $n \in \mathbb{N}_0$. Falls

$$\text{es ex. Intervall } I_0 \text{ mit } I|^n + I_0 \models \varphi$$

gilt, dann gilt auch

$$\text{f.a. } m \leq n \text{ ex. Intervall } I_0 \text{ mit } I|^m + I_0 \models \varphi$$

□

Beweis
Wähle ein $m \leq n$ beliebig. Nach oben genannter Voraussetzung gibt es ein I_0 mit

$$I|^m + I|_m^n + I_0 \models \varphi$$

Damit ist $I|_m^n + I_0$ das gesuchte Intervall.

□

Die Grundidee, wie aus einer Sicherheitseigenschaft φ eine Induktionshypothese generiert wird, ist folgende: Angenommen φ gilt nicht, dann gibt es ein n, so dass nach n Schritten ein Zustand erreicht wird, in dem φ verletzt ist. Damit kann eine Induktion über die Anzahl n von Schritten bis zu diesem Zustand durchgeführt werden. Um dies zu erreichen, wird der so genannte *Präfix-Operator* $\mathbf{prefix}(\varphi, \psi)$ in ITL$^+$ integriert. Dieser Operator betrachtet nur das Präfix eines Intervalls bis eine boolesche Variable (hier das ψ) zum ersten Mal wahr wird. Informell ähnelt der Präfix-Operator damit der *Safety-Closure* von Abadi & Lamport [AL95], welche nur den Sicherheitsanteil einer Formel betrachtet. Formal wird die Semantik des Präfix-Operators wie folgt definiert:

Definition 34 *(Semantik des Präfix-Operators).*
Sei $\varphi, \psi \in \mathbf{E}_{Bool}$. Dann gilt

$$\begin{aligned} I \models \mathbf{prefix}(\varphi, \psi) \quad \text{gdw} \quad & I \models \varphi \\ \text{oder ex. } & n < |I| \text{ und } I_0 \\ & \text{mit } I|_n \models \psi \text{ und } I|^n + I_0 \models \varphi \end{aligned}$$

□

Informell bedeutet die Semantik des Präfix-Operators, dass φ bis zum ersten Auftreten von ψ erfüllt sein muss. Damit ist eine Formel $\mathbf{prefix}(\varphi, \psi)$ für ein Intervall I in den folgenden drei Fällen erfüllt:

1. Falls ψ niemals in I erfüllt ist, dann muss $I \models \varphi$ gelten.

2. Falls I endlich ist und ψ nur im letzten Zustand erfüllt ist, muss ebenfalls $I \models \varphi$ gelten. Dieser Fall resultiert daraus, dass in der Definition des Operators für n nur kleinere Werte als $|I|$ gewählt werden dürfen.

3. In allen anderen Fällen gibt es ein minimales n mit $I|_n \models \psi$ und ein Intervall I_0 mit $I|^n + I_0 \models \varphi$. Dass es ein minimales n gibt, folgt aus Lemma 9.

Der erste Fall könnte auch anders gewählt werden. Hier wurde diese Variante bevorzugt, damit sich möglichst einfache symbolische Ausführungsregeln ergeben. Der zweite Fall ist notwendig, damit endliche Intervalle nicht beliebig erweitert werden können. Das folgende Beispiel erläutert das Problem das dabei entstehen kann: Auf einem Intervall I der Länge null gilt $X' = X$ (siehe Def. 8 auf Seite 13). Das einzige Präfix von I ist I selbst. Dieses könnte erweitert werden, so dass $X' \neq X$ gilt. Damit würde Satz 6 (siehe unten) nicht mehr gelten.

Für alle Sicherheitseigenschaften gilt der folgende Satz:

Satz 6.
Sei $\varphi \in \mathbf{SF}$ und $B \in \mathbf{D}_{bool}$ mit $B \notin \mathit{free}(\varphi)$. Dann gilt die folgende Äquivalenz:

$$\varphi \quad \leftrightarrow \quad (\forall\, B.\, \Diamond\, B \rightarrow \mathbf{prefix}(\varphi, B))$$

□

Beweis

„←“: Sei I ein Intervall mit:

$$I \models \forall\, B.\, \Diamond\, B \rightarrow \mathbf{prefix}(\varphi, B) \tag{7.16}$$

Falls $|I| = n$ endlich ist, gibt es ein Intervall J mit $J =_B I$ und $J|_n \models B$ und für alle $m < n$ gilt $J|_m \not\models B$. Nach Definition von **prefix** gilt dann $J \models \varphi$, also auch $I \models \varphi$, da $B \notin \mathit{free}(\varphi)$

Falls I unendlich ist, genügt es zu zeigen, dass

$$\text{f.a. } n \in \mathbb{N}_0 \text{ ex. } I_0 \text{ mit } I|^n + I_0 \models \varphi$$

gilt, da φ eine Sicherheitsformel ist. Wähle ein beliebiges $n \in \mathbb{N}_0$ und wähle dazu das Intervall $J =_B I$ mit $J|_n \models B$ und $J|_m \models \neg\, B$ für alle $m \neq n$. Nach (7.16) und der Definition für **prefix** gilt:

$$\text{ex. } I_0 \text{ mit } J|^n + I_0 \models \varphi$$

Aus $J =_B I$ und $B \notin \mathit{free}(\varphi)$ folgt für n die zu zeigende Formel:

$$\text{ex. } I_0 \text{ mit } I|^n + I_0 \models \varphi$$

„→“: Gelte $I \models \varphi$. Da $B \notin \mathit{free}(\varphi)$ gilt $J \models \varphi$ für alle $J =_B I$. Nach Definition von **prefix** gilt damit für alle J auch $J \models \mathbf{prefix}(\varphi, B)$. Also gilt:

$$I \models \forall\, B.\, \Diamond\, B \rightarrow \mathbf{prefix}(\varphi, B)$$

$$\frac{\vdash \varphi_1 \rightarrow \varphi_2}{\vdash \mathbf{prefix}(\varphi_1, \psi) \rightarrow \mathbf{prefix}(\varphi_2, \psi)} \; \textit{prfx lem}$$

prfx dis:	$\mathbf{prefix}((\varphi_1 \vee \varphi_2), \psi)$	$\leftrightarrow$	$\mathbf{prefix}(\varphi_1, \psi) \vee \mathbf{prefix}(\varphi_2, \psi)$
prfx ex:	$\mathbf{prefix}((\exists\ a.\ \varphi), \psi)$	$\leftrightarrow$	$\exists\ a.\ \mathbf{prefix}(\varphi, \psi)$
	wobei o.B.d.A. $a \notin \text{free}(\psi)$		
prfx lst1:	$\neg\ \psi$	$\rightarrow$	$(\mathbf{prefix}((\tau \wedge \mathbf{last}), \psi) \leftrightarrow \tau \wedge \mathbf{last})$
prfx stp1:	$\neg\ \psi$	$\rightarrow$	$(\mathbf{prefix}((\tau \wedge \circ\ \varphi), \psi) \leftrightarrow \tau \wedge \circ\ \mathbf{prefix}(\varphi, \psi))$
prfx lst2:	ψ	$\rightarrow$	$(\mathbf{prefix}((\tau \wedge \mathbf{last}), \psi) \leftrightarrow \tau^{A,A}_{A',A''})$
prfx stp2:	ψ	$\rightarrow$	$(\mathbf{prefix}((\tau \wedge \circ\ \varphi), \psi) \leftrightarrow \neg\ \mathbf{last} \wedge \mathbf{E}\,(\tau \wedge \circ\ \varphi))$

Tabelle 7.1: Normalformregeln für den Präfix-Operator

□

Mit Hilfe dieser Äquivalenz kann die folgende Induktionsregel für Safety-Formeln φ abgeleitet werden:

$$\frac{\Diamond\ B, \Gamma \vdash \mathbf{prefix}(\varphi, B), \Delta}{\Gamma \vdash \varphi, \Delta} \quad \varphi \in \boldsymbol{SF} \textit{ und } B \notin \textit{free}(\varphi, \Gamma, \Delta) \tag{7.17}$$

Informell wird hier angenommen, dass φ nur auf einem endlichen prefix des Traces gilt, dessen Ende durch das erste Auftreten von $B \leftrightarrow \mathit{true}$ markiert wird. Aus der Formel $\Diamond\ B$ kann nun – wie in Kapitel 2.5.1 beschrieben – eine Induktionshypothese generiert werden.

Um dieses allgemeine Induktionsschema für Sicherheitsformeln zu nutzen, ist es notwendig, Präfix-Formeln symbolisch ausführen zu können. Tabelle 7.1 gibt die benötigten Regeln für die symbolische Ausführung des Präfix-Operators an. Die Regeln *prfx lem*, *prfx dis* und *prfx ex* werden zur Umformung in die Normalform benötigt (siehe dazu auch Kapitel 2.4.1). Bei der symbolischen Ausführung von $\mathbf{prefix}(\varphi, \psi)$ werden, abhängig vom aktuellen Wert von ψ, zwei Fälle generiert. Falls $\psi \leftrightarrow$ false, führt die symbolische Ausführung von $\mathbf{prefix}(\varphi, \psi)$ zum gleichen Ergebnis wie ein symbolischer Ausführungsschritt mit φ und der Anwendung des Präfix-Operators nach dem Schritt (Regel *prfx stp1*). Dies ist möglich, da der ausgeführte Schritt ein Schritt des betrachteten Intervalls ist. Das gleiche gilt auch, wenn φ terminiert, das heißt, wenn die Normalform von φ gleich $\tau \wedge \mathbf{last}$ ist (Regel *prfx lst1*). In diesem Fall muss auch der letzte Zustand des betrachteten Intervalls erreicht sein. Falls $\psi \leftrightarrow$ true und φ terminiert, muss die Zustandsübergangsformel τ ein Stotterschritt sein, unabhängig davon, ob das betrachtete Intervall terminiert oder nicht (Regel *prfx lst2*). Falls das betrachtete Intervall nicht terminiert, ist damit sichergestellt, dass das leere Intervall das gesuchte Intervall I_0 des Präfix-Operators ist. Falls ψ gilt und die Formel φ nicht terminiert, muss gezeigt werden, dass es ein Intervall gibt, in dem φ erfüllt ist, und dass noch nicht der letzte Zustand des betrachteten Intervalls erreicht ist. Das heißt, in diesem Fall ist $\mathbf{prefix}(\varphi, \psi)$ äquivalent zur Formel $\neg\ \mathbf{last} \wedge \mathbf{E}\,\varphi$. Für reine sequentielle Programme α ist die

Formel $\mathbf{E}\,\alpha$ immer wahr. Da aber in dem hier verwendeten Formalismus Programme und Formeln gemischt werden können, ist diese Bedingung notwendig. So ist zum Beispiel die Formel $\mathbf{E}\,\alpha;\mathrm{false}$ für alle terminierenden Programme α falsch.

Mit dem oben angegeben Präfix-Operator kann aus jeder Sicherheitseigenschaft im Sukzedent eine Induktionshypothese generiert werden. Um aus sequentiellen Programmen eine Induktionshypothese erzeugen zu können, ist noch ein syntaktisches Kriterium notwendig, damit Sicherheitseigenschaften identifiziert werden können. Der folgende Satz gibt ein solches Kriterium an:

Satz 7.
Sei $\alpha \in \mathbf{E}_{Bool}$ eine Formel, die nur aus den folgenden Operatoren besteht:

- Boolesche Operatoren $\wedge$ und $\vee$.
- Temporallogische Operatoren $\Box$ und **unless**.
- Zuweisungen $V := e$ und Stotterschritte **skip**.
- Kontrollstrukturen **if***, **if**, **while*** und **while**.
- Programmkompositionen $\alpha_1;\alpha_2$ und $\alpha*$.

Dann ist α eine Sicherheitsformel. □

Sowohl Satz 7 als auch die Korrektheit der Regeln zum symbolischen Ausführen des Prefix-Operators wurden formal in KIV bewiesen.

7.5.1 Andere Ansätze

Im Zusammenhang mit Sicherheitseigenschaften wird in der Literatur häufig auch der Begriff *Sicherheitsabschluss* (engl. *safety closure*) benutzt. In [JT96] wird der Sicherheitsabschluss wie folgt definiert:

> The *safety closure* of a given property is the strongest safety property implied by the given property, i.e., a safety property satisfied by exactly those sequences σ such that each prefix of σ is a prefix of some sequence that satisfies the given property.

Informell wird mit der Formel $\forall\, B.\ \Diamond\, B \rightarrow \mathbf{prefix}(\varphi, B)$ aus Satz 6 der Sicherheitsabschluss von φ berechnet. Die Länge des betrachteten Präfixes wird dabei zur Generierung der Induktionshypothese benutzt. In [JT96] sind andere Charakterisierungen des Sicherheitsabschlusses angegeben, die theoretisch zur Generierung einer Induktionshypothese benutzt werden könnten. In der Praxis haben diese Charakterisierungen jedoch einige Nachteile im Vergleich zur oben benutzten Formalisierung mit dem Prefix-Operator.

Eine Formalisierung des Sicherheitsabschlusses einer Formel $\varphi(x)$ wird in [JT96] für LTL mit Vergangenheitsoperatoren angegeben. Um diese in ITL$^+$ anstelle einer Formalisierung mit dem Prefix-Operator zu verwenden, müsste

der Kalkül um Vergangenheitsoperatoren erweitert werden. Dazu müssten die entsprechenden Formeln auch rückwärts ausführbar sein.

In [JT96] findet sich ebenfalls eine Charakterisierung des Sicherheitsabschlusses in TLA für eine Formel $\varphi(x)$:[3]

$$\begin{aligned} \forall\, b : (\quad & b = 1 \wedge \Box\,(b' = 0 \vee b' = b) \wedge \Box \Diamond\,(b' = 0)) \\ & \Rightarrow (\exists\, y : \Box\,(b = 1 \Rightarrow x = y) \wedge \varphi[y/x]) \end{aligned} \tag{7.18}$$

Mit dieser Charakterisierung wird die freie Variable x von φ durch eine neue gebundene Variable y ersetzt. Solange $b = 1$ gilt, nimmt y die gleichen Werte an wie x. Dass $b = 1$ nur auf einem Präfix erfüllt ist, wird durch die Formeln $\Box\,(b' = 0 \vee b' = b)$ und $\Box \Diamond\,(b' = 0)$ sichergestellt. Aus dieser Idee lässt sich für den ITL$^+$-Kalkül eine abgeleitete Regel herleiten:

$$\frac{B \textbf{ until } \Box \neg B, \Gamma \vdash \exists\, Y.(X = Y \wedge \Box\,(B \rightarrow \circ\, Y = X) \wedge \varphi_X^Y), \Delta}{\Gamma \vdash \varphi, \Delta} \tag{7.19}$$

Die Variable X wird durch eine gebundene Variable Y ersetzt. Wie in Formel (7.18) gilt B nur auf einem Präfix des Intervalls, auf dem auch $X = Y$ gilt. Im Unterschied zu (7.18) muss $X = Y$ noch einen Zustand länger gelten, also auch noch im ersten Zustand, in dem nicht B gilt.[4] Der durch diese Regel eingeführte **until**-Operator kann benutzt werden, um eine Induktionshypothese zu generieren. Durch diese Regel muss nur am Ende des Präfixes gezeigt werden, dass eine gültige Fortsetzung des Intervalls existiert. Allerdings muss im Unterschied zur Regel (7.17) diese Fortsetzung die gleiche Länge haben wie das durch Γ beschriebene Intervall. Falls dieses unendlich ist, würde für den Beweis, dass φ auf dem Fortsetzungsintervall gilt, keine Induktionshypothese mehr zur Verfügung stehen.

7.6 Verifikation des Treiber-Stack Algorithmus

In diesem Abschnitt wird anhand von Treiber-Stack Algorithmus beschrieben, wie das in Abschnitt 7.3 vorgestellte Verfeinerungstheorem auf lock-freie Algorithmen angewandt werden kann. Im ersten Unterabschnitt wird zunächst die Instanzierung der generischen Operationen *COP* und *AOP*, sowie die Abstraktionsfunktion *absf* beschrieben. In den anderen Unterabschnitten werden die für die Verifikation benötigte Invariante und die Rely-Guarantee Bedingungen vorgestellt. Im letzten Abschnitt wird schließlich die eigentliche Verifikation der Beweisverpflichtungen behandelt.

[3] In TLA werden dynamische Variablen immer klein geschrieben. Außerdem bezeichnen gestrichene Variablen den Wert der Variable im nächsten Zustand (in TLA gibt es keine Zwischenzustände wie in ITL$^+$). Die etwas umständliche Formalisierung mit $b = 1 \wedge \Box\,(b' = 0 \vee b' = b) \wedge \Box \Diamond\,(b' = 0)$ anstatt eines einzelnen **until**-Operators (siehe Formel (7.19)) ist notwendig, um die von TLA geforderte Normalform zu erreichen.

[4] Dieser Unterschied ist notwendig, da mit der Formalisierung aus (7.18) auf einem endlichen Intervall im letzten Zustand der Wert von Y immer beliebig sein dürfte, da B (bzw b) in diesem immer falsch wäre. Damit könnte z. B. auch die ITL$^+$-Formel $X = 6, \textbf{last} \vdash X = 5$ bewiesen werden.

$$
\begin{array}{l}
APush(v; Stack)\ \{ \\
\quad \textbf{skip}^*; \\
\quad Stack := push(v, Stack); \\
\quad \textbf{skip}^* \\
\}
\end{array}
\qquad
\begin{array}{l}
APop(; Stack, O)\ \{ \\
\quad \textbf{let}\ Lo = \emptyset\ \textbf{in}\ \{ \\
\quad\quad \textbf{skip}^*; \\
\quad\quad \textbf{if}^*\ \ Stack \neq \emptyset\ \textbf{then}\{ \\
\quad\quad\quad Lo := top(Stack), \\
\quad\quad\quad Stack := pop(Stack); \\
\quad\quad \} \\
\quad\quad \textbf{skip}^*; \\
\quad\quad O := Lo \\
\quad \} \\
\}
\end{array}
$$

Abbildung 7.2: Formale Definition der abstrakten Stack Operationen

7.6.1 Konkrete und abstrakte Stack-Operationen

Die generische Zustandsvariable *CS* wird durch ein Tupel bestehend aus den Variablen *Top Hp* und den lokalen Zuständen $UNew(i)$, $USuccess(i)$, $OTop(i)$ und $OSuccess(i)$ für jeden Prozess i instanziert.

Eine konkrete lock-freie Operation $COP(i, \dots)$ wird wie folgt implementiert:

$$
\begin{array}{l}
COP(i, In; Top, Hp, UNew, USuccess, OTop, OSuccess, Out)\ \{ \\
\quad CPush(In(i); Top, Hp, UNew(i), USuccess(i)) \\
\quad\quad \vee \\
\quad CPop(; Top, Hp, OTop(i), OSuccess(i), Out(i)) \\
\}
\end{array}
$$

In der Operation $COP(i, \dots)$ wird nichtdeterministisch entweder eine Push-Operation *CPush* oder eine Pop-Operation *CPop* aufgerufen. Beiden Operationen wird bei jedem Aufruf von *COP* jeweils ein neuer Eingabewert zugewiesen, da die Eingabevariable *In* in jedem Umgebungsschritt von *CSeq* einen neuen Wert erhält.

Die abstrakte Operation $AOP(i, \dots)$ kann ebenso durch eine Fallunterscheidung definiert werden:

$$
\begin{array}{l}
AOP(i, In; Stack, Out)\ \{ \\
\quad APush(In(i); Stack) \\
\quad\quad \vee \\
\quad APop(; Stack, Out(i)) \\
\}
\end{array}
$$

Die Operation *AOP* bekommt die algebraisch definierte Datenstruktur „*Stack*“ als Parameter. Dieser entspricht dem abstrakten Zustand *AS*. Genauso wie in der Operation *COP* wird von *AOP* eine Push- oder Pop-Operation nichtdeterministisch aufgerufen. Die beiden entsprechenden Operationen *APush* und *APop* sind in Abbildung 7.2 definiert.

In beiden Operationen wird der Stack nur durch eine einzige atomare Operation *push* oder *pop* verändert. Zusätzlich wird die Operation *top* in einer

parallelen Zuweisung benutzt, um das erste Element des Stacks auszugeben. Die abstrakte Datenstruktur *Stack* wird algebraisch von Listen abgeleitet. Dabei bezeichnet $\emptyset$ die leere Liste, die Operation $push(v, Stack)$ fügt das Element v an die Liste *Stack* vorne an, $top(Stack)$ gibt das erste Element von *Stack* aus und $pop(Stack)$ entfernt das erste Element der Liste *Stack*.

Die zusätzlichen **skip**-Schritte in den beiden Operationen *APush* und *APop* stellen sicher, dass der abstrakte Ablauf die gleiche Länge wie der konkrete Ablauf hat. Wie schon in Abschnitt 7.2 beschrieben, müssen in ITL$^+$ zwei Programme die gleiche Anzahl von Schritten ausführen, damit eine Verfeinerung gezeigt werden kann. Die Ausgabe der abstrakten Pop-Operation wird bis zum letzten Schritt in einer lokalen Variable *Lo* zwischengespeichert. Damit wird erreicht, dass die globale Ausgabevariable $Out(i)$ von der konkreten und der abstrakten Operation im gleichen Schritt – dem letzten Schritt in beiden Operationen – verändert wird. Dadurch wird sichergestellt, dass die Ausgabewerte in einander entsprechenden abstrakten und konkreten Abläufen simultan verändert werden.

Um die Prämissen 4. und 6. aus dem Verfeinerungstheorem zu instanzieren, wird eine Umgebungsannahme $R(CS', CS'')$ für die globale Umgebung benötigt. Da die globale Umgebungsannahme alle lokalen Umgebungsannahmen implizieren muss (Prämisse 4. des Verfeinerungstheorems), wird dafür die Konjunktion aller lokalen Umgebungsannahmen R_i verwendet. Zusätzlich muss R noch die Invariante I erhalten (Prämisse 6.). Damit ergibt sich die folgende globale Umgebungsannahme:

$$R(CS', CS'') \;:\leftrightarrow\; (I(CS') \rightarrow I(CS'')) \wedge \forall i. R_i(CS', CS'')$$

Kein anderer Prozess außerhalb des Systems soll den Ablauf eines lokalen Systems beeinflussen und deren Annahmen verletzen. Mit dieser Umgebungsannahme wird keinerlei Annahme darüber getroffen, wie Zellen, auf die kein Zeiger mehr existiert, von der Umgebung verändert werden. Damit wird zum Beispiel der Umgebung erlaubt, diese Zellen wieder freizugeben, wie dies bei einem System mit Garbage Collection der Fall wäre.

Zusätzlich wird eine Abstraktionsfunktion *absf* und eine Invariante *I* benötigt. Zur Spezifikation der Abstraktionsfunktion wird zunächst eine Abstraktionsrelation *Abs* rekursiv definiert:

$$Abs(\emptyset, Top, Hp) \;:\leftrightarrow\; Top = \textsc{Null} \tag{7.20}$$

$$\begin{aligned} Abs(push(V, Stack), Top, Hp) \;:\leftrightarrow\;& Top \neq \textsc{Null} \wedge Top \in Hp \\ &\wedge\; Hp[Top].\mathrm{val} = V \\ &\wedge\; Abs(Stack, Hp[Top].\mathrm{nxt}, Hp) \end{aligned} \tag{7.21}$$

Der abstrakte leere Stack $\emptyset$ wird durch einen Nullzeiger in der Variable *Top* repräsentiert (Formel (7.20)). Im rekursiven Fall in Formel (7.21) muss die von *Top* referenzierte Zelle in *Hp* alloziert sein, der Datenwert in der Zelle muss dem obersten Datenwert des abstrakten Stacks entsprechen und der restliche Stack muss durch den .nxt-Zeiger der von *Top* referenzierten Zelle repräsentiert

werden. Wie man anhand dieser Definition sieht, stellen nicht alle Konfigurationen der konkreten Datenstruktur einen gültigen Stack dar. So gibt es zum Beispiel für eine zirkulär verkettete konkrete Datenstruktur, die von *Top* aus erreichbar ist, keinen abstrakten Stack, der bezüglich *Abs* dieser Konfiguration entsprechen würde. Um zu entscheiden, welche der konkreten Konfigurationen einen gültigen Stack darstellen, wird das Prädikat *valid* benutzt. Dieses kann einfach mit Hilfe der Abstraktionsrelation definiert werden:

$$valid(Top, Hp) \quad :\leftrightarrow \quad \exists\ Stack.\ Abs(Stack, Top, Hp) \tag{7.22}$$

Eine Konfiguration der konkreten Datenstruktur *Hp* und *Top* ist gültig, wenn es einen abstrakten *Stack* gibt, der bezüglich *Abs* mit der konkreten Datenstruktur in Relation steht.

Die Abstraktionsrelation *Abs* ist eindeutig definiert, d.h. es gilt

$$Abs(S_1, Top, Hp) \wedge Abs(S_2, Top, Hp) \rightarrow S_1 = S_2$$

Daher kann die partielle Abstraktionsfunktion *absf* mit Hilfe von *Abs* und dem Prädikat *valid* wie folgt definiert werden:

$$valid(Top, Hp) \quad \rightarrow \quad Abs(absf(Top, Hp), Top, Hp) \tag{7.23}$$

Falls die konkrete Repräsentation des Stacks gültig ist, so ist das Ergebnis der Abstraktionsfunktion *absf* einer der bezüglich *Abs* zur konkreten Repräsentation passenden Werte. Damit die Funktion *absf* in jedem erreichbaren Zustand definiert ist, wird die Formel $valid(Top, Hp)$ in die Invariante integriert (siehe dazu Abschnitt 7.6.2). Diese Definition führt nicht zu einem Zirkelschluss, da für den Beweis, dass die Invariante immer erfüllt ist, die Abstraktionsfunktion nicht benötigt wird.

Um die Beweisverpflichtungen des Verfeinerungstheorems zu beweisen, müssen noch eine Invariante I und die lokalen Umgebungsannahmen R_i für jeden Prozess i definiert werden. Die temporallogischen Beweisverpflichtungen können dann durch einfache symbolische Ausführung bewiesen werden. Außer zu Beginn der **while**-Schleifen, die eine manuelle Generalisierung benötigen, benötigen die Beweise keine aufwändigen Interaktionen.

7.6.2 Invariante

Obwohl Teile der Invariante und der lokalen Umgebungsannahmen schematisch konstruiert werden können, ist die Formalisierung dieser beiden Formeln der wesentliche kreative Schritt bei der Verifikation der lock-freien Algorithmen. Die Invariante I besteht aus vier Teilen: Der erste Teil kann als schematisch für alle lock-freien Algorithmen angesehen werden. Er sagt aus, dass die Invariante durch den Heap und den *Top*-Zeiger gültig ist, d.h. keine zyklischen Zeiger oder ungültige Referenzen enthält. Dazu wird das in Formel (7.22) definierte Prädikat *valid* benutzt. Im zweiten und dritten Teil der Invariante werden Eigenschaften der lokalen Variablen der Operationen *CPush* und *CPop* beschrieben. Eine generische Annahme setzt voraus, dass lokale Zeiger-Variablen entweder den

NULL-Zeiger oder eine Referenz auf eine allozierte Heap-Zelle enthalten. Damit wird sichergestellt, dass keine nicht allozierten Zellen fehlerhaft referenziert werden. Für die Pop-Operation ist die einzige relevante Variable *OTop*. Für diese Variablen werden die folgenden Formeln in der Invariante benutzt:

$$Inv_{pop}(OTop, Hp) \quad :\leftrightarrow \quad \forall\, i.\ OTop(i) = \text{NULL} \vee OTop(i) \in Hp$$

Eine identische Eigenschaft wie für *OTop* wird auch für die Variable *UNew* der Push-Operation benötigt. Die vierte und letzte Eigenschaft, die für die Invariante benötigt wird, sagt aus, dass eine Zelle, die von einer Push-Operation alloziert wird, nicht ein Teil der Stack-Repräsentation auf dem Heap ist, solange sie nicht durch ein erfolgreiches *CAS* von der allozierenden Push-Operation in den Stack integriert wird. Ohne diese Eigenschaft wäre es möglich, dass durch eine Modifikation dieser Zelle unbeabsichtigt der Stack verändert wird. Ob sich eine Push-Operation noch vor einem CAS befindet, kann man feststellen, indem überprüft wird, ob die Variable *USuccess* falsch ist. Damit ergibt sich als Invariante für die Push-Operation die folgende Formel:

$$\begin{aligned} &Inv_{push}(UNew, USuccess, Top, Hp) \ :\leftrightarrow \\ &\qquad \forall\, i. \quad (UNew(i) = \text{NULL} \vee UNew(i) \in Hp) \\ &\qquad\quad \wedge\ (\neg\, USuccess(i) \rightarrow \neg\, reachable(Top, UNew(i), Hp)) \end{aligned}$$

Das Prädikat $reachable(Top, R, Hp)$ ist genau dann wahr, wenn R eine Referenz auf eine Zelle enthält, die von *Top* aus erreichbar ist. Die Erreichbarkeit einer Zelle wird dabei mithilfe einer Liste $p = [r_1, \ldots r_n]$ von Referenzen definiert, die einen Pfad im Heap darstellen:

$$reachable(Top, R, Hp) \quad :\leftrightarrow \quad \exists\, p.\ path(Top + p + R, Hp) \tag{7.24}$$

Das Prädikat *path* wird wie folgt rekursiv definiert:

$$\begin{aligned} \neg\, path([\,], Hp) & \\ path([r + [\,], Hp) \quad &\leftrightarrow \quad r \in Hp \wedge r \neq \text{NULL} \\ path(r_1 + r_2 + p, Hp) \quad &\leftrightarrow \quad r_1 \in Hp \wedge r_1 \neq \text{NULL} \wedge Hp[r_1].\text{nxt} = r_2 \\ &\qquad\quad \wedge\ path(r_2 + p, Hp) \end{aligned}$$

Der „+“-Operator ist überladen und konkateniert sowohl Listen als auch Elemente.

Als letzte Eigenschaft für die Invariante wird noch gefordert, dass alle Referenzen der lokalen Variablen *UNew* und *OTop* jeweils disjunkt sind. Dadurch wird sichergestellt, dass durch eine Modifikation auf einer neu allozierten *UNew*-Zelle einer Push-Operation keine anderen Operationen beeinflusst werden, indem der Wert der referenzierten *UNew*- bzw. *OTop*-Zelle verändert wird. Die Tatsache, dass die *UNew*-Zellen im Stack nicht erreichbar, die *OTop*-Zellen dagegen erreichbar sind, wenn ihnen mittels $OTop := Top$ ein neuer Wert zugewiesen wird, reicht nicht aus, um die Disjunktheit dieser Zellen sicherzustellen. Der Grund dafür ist folgender: Es kann nicht garantiert werden, dass eine

OTop-Zelle im Stack erreichbar bleibt, nachdem diese Zuweisung ausgeführt wurde. Um die Disjunktheit aller *UNew*- und *OTop*-Zellen auszudrücken, wird das Prädikat *disj* benutzt:

$$\begin{aligned} \mathit{disj}(\mathit{UNew}, \mathit{USuccess}, \mathit{OTop}) \quad :\leftrightarrow \\ \forall\, i, j.\, i \neq j \rightarrow \quad & (\mathit{UNew}(i) = \textsc{Null} \vee \mathit{UNew}(i) \neq \mathit{UNew}(j)) \\ & \wedge\, (\quad \neg\, \mathit{USuccess}(i) \wedge \mathit{OTop}(j) \neq \textsc{Null} \\ & \quad \rightarrow \mathit{UNew}(i) \neq \mathit{OTop}(j)) \end{aligned}$$

Insgesamt ergibt sich damit die folgende Invariante:

$$\begin{aligned} I(\mathit{CS}) \quad :\leftrightarrow \quad & \mathit{valid}(\mathit{Top}, \mathit{Hp}) \\ & \wedge\, \mathit{Inv}_{\mathit{pop}}(\mathit{OTop}, \mathit{Hp}) \\ & \wedge\, \mathit{Inv}_{\mathit{push}}(\mathit{UNew}, \mathit{USuccess}, \mathit{Top}, \mathit{Hp}) \\ & \wedge\, \mathit{disj}(\mathit{UNew}, \mathit{USuccess}, \mathit{OTop}) \end{aligned}$$

7.6.3 Lokale Rely- und Guarantee-Eigenschaften

Die lokalen Umgebungsannahmen R_i bestehen aus drei Teilen: Der erste Teil ist schematisch: Er enthält nur die Eigenschaft, dass die Invariante erhalten bleibt. Im zweiten und dritten Teil sind wieder Eigenschaften enthalten, die von der Push- bzw. Pop-Operation benötigt werden.

$$\begin{aligned} R_i(\mathit{CS}', \mathit{CS}'') \quad :\leftrightarrow \\ & (I(\mathit{CS}') \rightarrow I(\mathit{CS}'')) \\ \wedge\, & R_{i,\mathit{push}}(\mathit{UNew}', \mathit{USuccess}', \mathit{Hp}', \mathit{UNew}'', \mathit{USuccess}'', \mathit{Hp}'') \\ \wedge\, & R_{i,\mathit{pop}}(\mathit{OTop}', \mathit{OSuccess}', \mathit{Hp}', \mathit{OTop}'', \mathit{OSuccess}'', \mathit{Hp}'') \end{aligned}$$

Beide Umgebungsannahmen für Push und Pop enthalten einen trivialen Teil, in dem zugesichert wird, dass die lokalen Variablen während des Umgebungsschritts nicht verändert werden. Für die Umgebungsannahme der Push-Operation wird zusätzlich noch angenommen, dass der Inhalt der neu allozierten Zelle nicht geändert wird, solange diese noch nicht von der Push-Operation in den Stack integriert wurde. Damit kann die Rely-Bedingung für eine Push-Operation wie folgt definiert werden:

$$\begin{aligned} R_{i,\mathit{push}}(\mathit{UNew}', \mathit{USuccess}', \mathit{Hp}', \mathit{UNew}'', \mathit{USuccess}'', \mathit{Hp}'') \quad :\leftrightarrow \\ & \mathit{USuccess}'(i) = \mathit{USuccess}''(i) \wedge \mathit{UNew}'(i) = \mathit{UNew}''(i) \\ \wedge\, (\quad & \mathit{UNew}'(i) \neq \textsc{Null} \wedge \neg\, \mathit{USuccess}'(i) \\ & \rightarrow \mathit{Hp}'[\mathit{UNew}'(i)] = \mathit{Hp}''[\mathit{UNew}''(i)]) \end{aligned}$$

Die Rely-Bedingung für eine Pop-Operation ist ähnlich. Der Inhalt der *OTop*-Zelle darf nicht geändert werden, solange die Pop-Operation aktiv ist.

Mit dieser Bedingung werden ungewollte Änderungen des Stacks vermieden wie sie zum Beispiel durch das ABA-Problem auftreten können. Damit kann die Rely-Bedingung für eine Pop-Operation wie folgt formalisiert werden:

$$\begin{aligned} R_{i,pop}(OTop', OSuccess', Hp', OTop'', OSuccess'', Hp'') \;:\leftrightarrow\; & \\ & OSuccess'(i) = OSuccess''(i) \wedge OTop'(i) = OTop''(i) \\ \wedge\; (\quad & OTop'(i) \neq \textsc{Null} \wedge \neg\, OSuccess'(i) \\ & \rightarrow Hp'[OTop'(i)] = Hp''[OTop''(i)]) \end{aligned}$$

Zuletzt müssen noch die lokalen Guarantee-Bedingungen definiert werden. Da die lokalen Garantien nur stark genug sein müssen, um die lokalen Rely-Bedingungen aller anderen Prozesse zu erfüllen (Beweisverpflichtung *2.* des Verfeinerungstheorems), werden für die lokalen Guarantee-Bedingungen die Konjunktion der Rely-Bedingungen angenommen:

$$G_i(CS', CS'') \;:\leftrightarrow\; \forall\, j.\; j \neq i \rightarrow R_j(CS', CS'')$$

7.6.4 Verifikation

Mit diesen Eigenschaften können alle Beweisverpflichtungen des Verfeinerungstheorems für die Treiber-Stack Fallstudie bewiesen werden. Alle Beweise sind online unter [KIV] verfügbar. Die prädikatenlogischen Beweisverpflichtungen *2.* bis *8.* des Verfeinerungstheorems sind einfach und können in wenigen Schritten geschlossen werden. Die einzigen komplexeren Beweise sind der Beweis der lokalen Rely-Guarantee-Bedingung (Beweisverpflichtung *1.* des Verfeinerungstheorems) und der Verfeinerungsbeweis für die *COP*-Operation (Beweisverpflichtung *9.*). Beide Beweise teilen sich in einen Ast für die Push-Operation und einen Ast für die Pop-Operation auf. Zur Generierung der Induktionshypothese kann für den Rely-Guarantee-Beweis die **unless**-Formel des $\overset{+}{\rightarrow}$-Operators benutzt werden. Beim Verfeinerungsbeweis ist dies schwieriger. Als Ziel wird angestrebt, das Programm auf der rechten Seite so umzuformen, dass es einer reinen Safety-Formel entspricht. Dann kann — wie in Abschnitt 7.5 beschrieben — daraus eine Induktionshypothese generiert werden. Dazu muss jedoch der aus der Deklaration der lokalen Variable *Lo* stammende Existenzquantor der *APop*-Operation eliminiert werden, da Sicherheitseigenschaften unter Existenzquantifikation nicht abgeschlossen sind. Der Existenzquantor kann eliminiert werden, indem die **let**-Anweisung zum Existenzquantor expandiert wird und dann die quantifizierte Variable *Lo* durch die Formel

$$OSuccess \;\supset\; Lo \;;\; \emptyset$$

instanziert wird. Solange die Operation aktiv ist, wird für *Lo* der Wert der gleichnamigen Variable aus der *CPop*-Operation verwendet, ansonsten wird für *Lo* der Wert $\emptyset$ benutzt. Nachdem der Quantor eliminiert ist, bildet die Formel im Sukzedent eine Sicherheitseigenschaft, da das verbleibende Programm genau die in Satz 7 geforderte syntaktische Form hat. Damit kann aus dem Sukzedent eine Induktionshypothese generiert werden.

Der eigentliche Verfeinerungsbeweis kann nun durch symbolische Ausführung geführt werden. Der passende Linearisierungspunkt, an dem die abstrakte Operation ausgeführt wird, muss dabei interaktiv ausgewählt werden.

Interessanterweise ist es dabei nicht nötig, die Vorbedingung zu Beginn der **while**-Schleife zu einer (Hoare-ähnlichen) Invariante zu generalisieren. Es scheint eine Besonderheit von lock-freien Algorithmen zu sein, dass ein erneutes Ausführen der Schleife alle Ergebnisse aus vorherigen Schleifen-Iterationen verwirft. Das hat den Effekt, dass die Vorbedingung als Invariante genügt.

7.7 Andere Arbeiten zur Korrektheit von lock-freien Algorithmen

Die Verifikation von lock-freien Algorithmen ist ein sehr aktuelles Forschungsgebiet. Zu den ersten, die signifikante Arbeiten zur Verifikation von lock-freien Algorithmen vorgelegt haben, gehört die Gruppe von Colvin & Groves. Verifiziert wurden mit diesem Ansatz der „Treiber-Stack“-Algorithmus und eine Erweiterung um Elimination [CDG05], sowie der „Michael-Scott“-Queue-Algorithmus [DGLM04]. Fehler wurden außerdem in einer Arbeit von Detlefs et al. [DFG+00] zu lock-freien Deques (double-ended queues) gefunden und korrigiert [DDG+04]. Der Ansatz basierte auf der Verfeinerung von IO-Automaten [LV95]. Die Beweise wurden im Verifikationswerkzeug PVS [SOR93] formalisiert. Im Vergleich zu der hier vorgestellten Technik müssen die Algorithmen als IO-Automaten kodiert werden. Die Beweise sind nicht modular, sondern es werden alle parallel laufenden Operationen auf einmal vom verifizierten IO-Automaten betrachtet.

Neuere Arbeiten von Colvin & Groves verfolgen einen auf Reduktion (siehe [Lip75, LS89, CL98, Mis03]) basierenden Ansatz [GC07, GC09]. Dieser Ansatz erscheint allerdings kaum formalisierbar. Die zugrunde liegende Annahme, dass Anweisungen, die auf unterschiedliche Variablen zugreifen, vertauscht werden können, ist meistens nicht anwendbar, da bei lock-freien Algorithmen praktisch alle Anweisungen auf eine Variable zugreifen: den globalen Heap. Daher können die Anweisungen nur dann vertauscht werden, wenn sichergestellt ist, dass beide Anweisungen auf unterschiedliche Stellen im Heap zugreifen. Tatsächlich wird ein großer Teil der Rely- und Guarantee-Bedingungen in der vorliegenden Arbeit dafür benötigt, um diese Annahmen sicherzustellen. Dagegen wird in [GC07, GC09] nur informell argumentiert, dass solche Annahmen korrekt sind.

Von Colvin et al. stammt aktuell noch ein mechanisierter Ansatz zur Verifikation von Lebendigkeitseigenschaften bei lock-freien Algorithmen [CD07, CD09]. Dieser Ansatz wird in Kapitel 8.6 dieser Arbeit beschrieben.

Ein anderer Ansatz stammt von der Gruppe um Vafeiadis [VHHS06, CPV07, Vaf07]. Diese entwickelte das automatische Werkzeug SmallFootRG zur Verifikation von lock-freien Algorithmen. Grundlage der Theorie ist dabei ebenfalls ein Rely-Guarantee Ansatz. Für kritische Instruktionen im Programm (CAS, lock, unlock) werden explizite Zusicherungen angegeben. Mit Hilfe von Abstraktion und der Invarianten-Generierungs-Technik (nach [DOY06]) können dann die Beweise für die betrachteten Beispiele automatisch geführt werden. Die

Zusicherungen werden in einer Erweiterung von Separation Logic [Rey02] definiert, so dass nur lokal zugreifbare Speicherzellen von globalen Speicherzellen unterscheidbar sind. Da viele Beweise automatisch gefunden werden können, sind die Ergebnisse der Gruppe sehr beeindruckend. Allerdings hat der Ansatz auch einige Nachteile. Im Gegensatz zur hier vorgestellten Arbeit wird abstrakter Code (sog. „Ghost Code“) zu konkretem Code hinzugefügt, indem abstrakter Spezifikationscode (in diesem Fall abstrakte Operationen an den Linearisierungspunkten) in den konkreten Implementierungscode geschrieben wird. Anschließend wird gezeigt, dass beides ungefähr synchron die abstrakte und konkrete Datenstruktur modifiziert. Ein Vorteil dieses Ansatzes ist, dass auch Beispiele, in denen der Linearisierungspunkt außerhalb des eigenen Prozesses liegt, behandelt werden können. Im Vergleich zu anderen Ansätzen mit Verfeinerung, wie der von Colvin & Groves oder der hier beschriebene Ansatz, ist dieses Korrektheitskriterium informeller (u. a. wird informell argumentiert, dass mehrfaches Linearisieren für nicht-destruktive Operationen unproblematisch ist). Auch ist der Ansatz nicht inkrementell, da keine Verfeinerungsbeziehung gezeigt wird. Damit entspricht er nicht der in dieser Arbeit verfolgten Grundidee, Verfeinerung nachzuweisen, was eine inkrementelle Entwicklung von Algorithmen aus Spezifikationen ermöglicht. Auch werden mit dem Ansatz nur die Beweisverpflichtungen in Separation Logic verifiziert, während ein Großteil der Rely-Guarantee Theorie nur auf dem Papier entwickelt und bewiesen wurde.

Für diesen Ansatz wurde ebenfalls eine Methode zur Verifikation von Lebendigkeitseigenschaften entwickelt [GCPV09], die in Kapitel 8.6 genauer beschrieben wird.

Ein weiterer automatischer Ansatz, der auf der so genannten „Shape Difference Analysis“ basiert, wurde von der Gruppe von Yahav & Sagiv entwickelt [ARR$^+$07, BLAM$^+$08, VY08]. Bei dieser Arbeit wird ein vereinfachtes Problem betrachtet: Zum einen werden statt der Prozesse *CSeq* nur einzelne interleaved parallel ablaufende lock-freie Operationen *COP* betrachtet. Dabei wird informell argumentiert, dass zwei sequentielle Aufrufe von *COP* eines Prozesses auch durch zwei verschiedene *COP*-Prozesse, die nacheinander starten, ausgeführt werden können. Zum anderen wird anstatt einer abstrakten *AOP*-Operation eine atomar ausgeführte *COP*-Operation benutzt. Um nachzuweisen, dass eine abstrakte Operation (auf Stacks, Queues, etc.) korrekt durch den Algorithmus implementiert wurde, ist eine zusätzliche Verfeinerung notwendig. Der Ansatz ist dann aber vollautomatisch. Er zeigt, dass die Differenz zwischen Pointerstrukturen bei atomarer und interleaveter Ausführung innerhalb gewisser Grenzen bleibt, die symbolisch als Unterschiede zwischen Shape-Graphen codiert werden. Damit ist dieser Ansatz aber auf Zeigerstrukturen beschränkt, während in der vorliegenden Arbeit ein generisches Verfeinerungstheorem benutzt wird.

Eine der bisher aufwändigsten Verifikationen von lock-freien Algorithmen (2 Personenjahre) stammt von Gao & Hesselink [GGH05]. Dabei wurde ein komplexer Algorithmus für lock-freie Hashtabellen mit PVS untersucht. Weitere Arbeiten betreffen die Verifikation eines lock-freien Garbage Collectors [GGH07], sowie eine weitere Arbeit, die allgemeine Patterns in lock-freien Algorithmen identifiziert und das ABA Problem analysiert [GH07].

Ein weiterer Ansatz stammt von Derrick et al. [DSW07, DSW08, DSW10]. An diesen Arbeiten ist – wie bei der vorliegenden Arbeit – die Gruppe um Prof. Reif beteiligt. Dieser Ansatz benutzt ebenfalls den interaktiven Theorembeweiser KIV. Diese Arbeiten stellen den einzigen Ansatz dar, der das Korrektheitskriterium Linearisierbarkeit von Herlihy & Wing formal nachweisen, während alle anderen Ansätze nur informell Linearisierbarkeit zeigen. Obwohl das verwendete Beweiswerkzeug – KIV – und die grundlegende Strategie – Modularisierung des Korrektheitsbeweises – die gleichen wie in der vorliegenden Arbeit sind, benutzt der Ansatz von Derrick et al. doch andere Techniken: In [DSW07] wird CSP zur Spezifikation der Kontrollstrukturen benutzt, während [DSW08] dazu Programmzähler verwendet. In beiden Arbeiten werden die einzelnen Anweisungen der Algorithmen mit Z spezifiziert. Interleaving wird explizit modelliert anstatt einen eigenen Operator dafür zu benutzen. Die entstehenden Beweisverpflichtungen entsprechen denen der Owicki-Gries Technik [OG76]. Diese haben den Vorteil, rein prädikatenlogisch zu sein und sie können außerdem die Symmetrie vieler Zusicherungen besser ausnutzen. Allerdings hat dieser Ansatz auch den Nachteil, dass für jeden einzelnen Ausführungsschritt Zusicherungen angegeben werden müssen, während bei dem hier beschriebenen Ansatz diese Zusicherungen automatisch durch die symbolische Ausführung berechnet werden. Dadurch steigt der Aufwand bei der Verifikation von großen Algorithmen bei dem Ansatz von Derrick et al. deutlich stärker als bei dem hier beschriebenen Ansatz.

Zwei Einzelarbeiten zur Verifikation von lock-freie Algorithmen sind die von Abrial & Cansell zur inkrementellen Entwicklung einer einfachen Queue mit B [AC05] und die Arbeit von Lamport [Lam06b], die den in [DDG+04] gefundenen Fehler im Algorithmus von [DFG+00] mit Bounded Modelchecking rekonstruiert.

7.8 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie sich die in Kapitel 4 entwickelte Rely-Guarantee-Methode erweitern lässt, um Verfeinerungseigenschaften von parallelen Programmen nachzuweisen. Die eigentlichen Verfeinerungseigenschaften müssen dabei nur für die lokalen Operationen bewiesen werden. Das Verfeinerungstheorem kann wie das Modularisierungstheorem in Kapitel 4 direkt in ITL^+ bewiesen werden. Eine weitere Besonderheit dieses Ansatzes wird in Kapitel 9 anhand der „Michael-Scott“-Queue demonstriert: Einige komplexe Algorithmen, die normalerweise spezielle Techniken wie z. B. Rückwärtssimulation benötigen, können mit dem hier beschriebenen Ansatz direkt (d. h. ohne zusätzliche Techniken) bewiesen werden.

Eine interessante Aufgabe für die Zukunft ist es, die explizite Linearisierbarkeit aus dem Ansatz von Derrick et al. [DSW08, DSW10] auf den hier vorgestellten temporallogischen Ansatz zu übertragen. Momentan können nur Algorithmen untersucht werden, deren Linearisierungspunkt in einem Systemschritt liegt. Eine interessante wissenschaftliche Fragestellung ist daher die Erweiterung dieses Ansatzes auf Algorithmen, deren Linearisierungspunkt außerhalb

des eigenen Prozesses liegt. Zusätzlich scheint es möglich, die Formulierung der Rely- und Guarantee-Bedingungen weiter zu vereinfachen, indem die Symmetrie der Komponenten besser ausgenutzt wird. All diese Fragestellungen werden in dem gerade anlaufenden DFG-Forschungsprojekt VeriCAS – das diese Arbeit als Grundlage benutzt – untersucht.

Neben der Korrektheit von Algorithmen (im Sinne einer Sicherheitseigenschaft) spielt auch die Lebendigkeit eine wichtige Rolle. Im nächsten Kapitel wird beschrieben, wie der hier beschriebene Ansatz erweitert werden kann, um Lock-Freedom – eine Lebendigkeitseigenschaft für lock-freie Algorithmen – nachzuweisen.

Kapitel 8

Lebendigkeit von lock-freien Algorithmen

Neben der Verifikation von klassischen Sicherheitseigenschaften spielt auch die Verifikation von Lebendigkeitseigenschaften eine wichtige Rolle. Lebendigkeitseigenschaften sind eine Klasse von temporallogischen Eigenschaften, die oft durch den informellen Satz „*Something good eventually happens*“ charakterisiert werden [Lam79]. Sicherheitseigenschaften zeigen, dass die Prozesse keine falschen Operationen ausführen. Im Gegensatz dazu wird bei Lebendigkeitseigenschaften gezeigt, dass Prozesse auch einen Fortschritt erzielen. Das heißt zum Beispiel: Operationen werden immer wieder erfolgreich ausgeführt.

Im folgenden Kapitel wird beschrieben, wie modulare Verifikationstechniken auf parallele Algorithmen angewandt werden können, um Lebendigkeitseigenschaften zu zeigen. Dabei wird – wie schon in den beiden vorhergehenden Kapiteln – besonderes Augenmerk auf die Verifikation von lock-freien Algorithmen gelegt.

Im ersten Abschnitt 8.1 wird, aufbauend auf dem formalen Rahmen für lock-freie Algorithmen aus Kapitel 7, zunächst ein formales Kriterium für Fortschritt bei lock-freien Algorithmen spezifiziert. In Abschnitt 8.2 werden verschiedene Klassen von Lebendigkeitseigenschaften identifiziert und besprochen. Im darauf folgenden Abschnitt 8.3 wird für die schwierigste dieser Klassen – die *Lock-Freedom*-Eigenschaft – ein Modularisierungstheorem entwickelt. Der Beweis für dieses Theorem ist in Abschnitt 8.4 dargestellt. Die Anwendung dieses Theorems wird anhand der „Treiber-Stack“-Fallstudie in Abschnitt 8.5 und der „Michael-Scott Queue“ in Kapitel 9 gezeigt. Ein Teil der hier beschriebenen Ergebnisse wurde vom Autor bereits in [TBSR10] veröffentlicht.

8.1 Spezifikation von Fortschritt in lock-freien Systemen

Im vorhergehenden Kapitel wurde mit dem Verfeinerungstheorem gezeigt, dass lock-freie Algorithmen auch in einer parallelen Umgebung die Datenstruktur nicht ungültig modifizieren. Dies stellt nach der informellen Definition „*Something bad never happens*“ eine Sicherheitseigenschaft dar. Ein Fortschritt im Sin-

ne einer Lebendigkeitseigenschaft wird durch das Verfeinerungstheorem nicht gezeigt.

Im Kapitel 7 wurde ein formales Modell zur Verifikation eines parallelen, auf lock-freien Datenstrukturen basierenden Systems entwickelt (siehe Kapitel 7.1). Interne Operationen werden in diesem System durch **skip**-Schritte symbolisiert, während Aufrufe der lock-freien Operationen durch einen Aufruf von *COP* definiert werden. Dieses Modell wurde in Kapitel 7 benutzt, um die Korrektheit von lock-freien Algorithmen zu zeigen. Bei lock-freien Systemen bedeutet Fortschritt informell, dass momentan aktive Operationen irgendwann beendet werden. Um dies zu formalisieren, wird ein Verfahren benötigt, mit dem man in einem System beobachten kann, welche Komponenten gerade eine lock-freie Operation aktiv ausführen. Zu diesem Zweck wurde das in Kapitel 7 vorgestellte Modell erweitert: Für jede Komponente werden dabei boolsche Werte (so genannte *Activity-Flags*) eingeführt, die genau dann wahr sind, wenn die entsprechende Komponente gerade eine lock-freie Operation ausführt. Dazu muss der Prozess *CSeq* wie folgt angepasst werden:

Definition 35 $CSeq_l$.

$$
\begin{array}{l}
CSeq_l(i; Act, In, CS, Out)\ \{ \\
\quad (\quad \textbf{skip} \\
\qquad \vee\ Act(i) := \text{true}; COP(i, In; CS, Out); Act(i) := \text{false})^* \\
\}
\end{array}
$$

□

Gespeichert werden diese boolschen Werte im Array Act, das nach Komponentennummern indiziert ist. Direkt vor dem Aufruf von *COP* wird der entsprechende Wert auf true gesetzt und direkt nach der Operation wieder auf false zurückgesetzt. Somit kann durch $Act(i) = \text{true}$ festgestellt werden, ob die i-te Komponente gerade eine aktive Operation ausführt. Das Ende einer aktiven Operation wird formal durch die Formel $Fin(i)$ ausgedrückt:

$$Fin(i) \quad :\equiv \quad Act(i) = \text{true} \wedge Act'(i) = \text{false}$$

Der einfache Test $Act'(i) = \text{false}$ reicht dazu nicht aus, da so nicht erkannt wird, ob die Operation zuvor aktiv war.

Die Spawn-Prozedur kann analog zur gleichnamigen Prozedur in Kapitel 7 definiert werden. Lediglich die Variable Act muss als neuer Parameter berücksichtigt werden. Damit ergibt sich die neue Spawn-Prozedur:

Definition 36 $CSpawn_l$.

$$
\begin{array}{l}
CSpawn_l(n; Act, In, CS, Out)\ \{ \\
\quad \textbf{if } n = 0 \textbf{ then} \\
\qquad CSeq_l(0; Act, In, CS, Out) \\
\quad \textbf{else} \\
\qquad CSeq_l(n; Act, In, CS, Out) \parallel CSpawn(n-1; Act, In, CS, Out) \\
\}
\end{array}
$$

□

8.2 Vergleich von verschiedenen Lebendigkeitseigenschaften

Wie in den vorhergehenden Kapiteln erörtert, haben die klassischen Mutex-Algorithmen mit ihren *locking*-Strategien den Nachteil, dass sich laufende Prozesse in starkem Maße gegenseitig beeinflussen. Wenn zum Beispiel der in Kapitel 3.1 vorgestellte Semaphor-Prozess in der kritischen Sektion abstürzt, kann im weiteren Ablauf kein anderer Prozess diese Sektion erneut betreten, da die Semaphor vom abgestürzten Prozess nicht mehr freigegeben werden kann. Ein Fortschritt im Sinne von Lebendigkeit ist hier nicht mehr möglich.

Um diese Probleme zu umgehen, wurden in den letzten Jahren parallele Algorithmen entwickelt, die alternative Mechanismen zur Synchronisierung benutzen. Bei diesen Algorithmen ist die gegenseitige Beeinflussung zwischen den Prozessen geringer als bei den klassischen Algorithmen. Dies bringt viele Vorteile bei der Benutzung der Algorithmen im Softwareentwicklungsprozess bezüglich Fehlertoleranz und Skalierbarkeit (siehe z.B. [GC96]).

Die Synchronisationsstrategien dieser Algorithmen sind oft sehr unterschiedlich und meistens von der Datenstruktur abhängig, für die diese Algorithmen entwickelt wurden. Daher sind auch die Zusicherungen bezüglich der Lebendigkeitseigenschaften sehr verschieden.

Um diese Algorithmen trotzdem kategorisieren zu können, wurden folgende drei Lebendigkeitseigenschaften von Herlihy [Her03] formuliert: „*Wait-Free*“, „*Lock-Free*“ und „*Obstruction-Free*“, die manchmal auch *Wait-Freedom*, *Lock-Freedom* bzw. *Obstruction-Freedom* genannt werden.

Im nächsten Abschnitt wird zunächst beschrieben, wie Fortschritt mit dem in dieser Arbeit benutzten Formalismus ausgedrückt werden kann. In den drei nachfolgenden Abschnitten werden dann die drei verschiedenen Lebendigkeitseigenschaften vorgestellt.

8.2.1 Formalisierung von Fortschritt

Den Ausgangspunkt für die Formalisierung der Lebendigkeitseigenschaften bildet die in Abschnitt 8.1 erarbeiteten Struktur von lock-freien parallelen Systemen. Das System besteht aus mehreren interleaved ausgeführten Prozessen. Jeder dieser Prozesse führt entweder interne Schritte oder eine einzelne Operation auf einer gemeinsamen Datenstruktur aus.

Ein Prozess i macht einen Fortschritt, wenn eine aktive Operation terminiert. Im Folgenden wird dies durch das in Abschnitt 8.1 eingeführte Prädikat $Fin(i)$ angezeigt: $Fin(i)$ ist genau in den Schritten wahr, in denen eine vom Prozess i ausgeführte Operation terminiert.

Für die *Lock-Freedom*- und *Obstruction-Freedom*-Eigenschaften ist es zudem nötig, für einen Prozess i ausdrücken zu können, ob er gerade eine Operation ausführt oder nicht. Zu diesem Zweck wird dazu ebenfalls die Funktion $Act(i)$ benutzt.

8.2.2 Die Eigenschaft *Obstruction-Freedom*

In [Her03] wird die *Obstruction-Free*-Eigenschaft wie folgt definiert:

> A synchronization technique is *obstruction-free* if it guarantees progress for any thread that eventually executes in isolation. Even though other threads may be in the midst of executing operations, a thread is considered to execute in isolation as long as the other threads do not take any steps.

Ein Algorithmus ist also *Obstruction-Free*, wenn jeder Prozess irgendwann einen Fortschritt erzielt, sobald er ab einem beliebigen Zeitpunkt der einzige Prozess ist, der noch Schritte ausführt. Auf Komponentenebene entspricht die Isoliertheit eines Prozesses mit den Variablen CS der Formel $\Box\ CS' = CS''$, das heißt, die Umgebung ändert keine Prozessvariablen mehr.

Damit lässt sich *Obstruction-Freedom* als lokale Eigenschaft formalisieren:

$$\Box\ R_i(CS', CS''), CSeq_i(CS) \vdash \Box\ ((\Box\ CS' = CS'') \wedge Act(i) \rightarrow \Diamond\ Fin(i))$$

Ein einzelner *CSEQ*-Prozess kann von Beginn an annehmen, dass seine Umgebungsannahme R_i immer erfüllt ist. Sobald ein Zustand erreicht ist, von dem ab der Prozess isoliert abläuft und eine Operation aktiv ist, muss gezeigt werden, dass diese Operation terminiert.

Obstruction-Freedom bildet die schwächste der hier vorgestellten Eigenschaften. Allerdings sind einige klassische Synchronisationsalgorithmen nicht einmal *Obstruction-Free*. Zum Beispiel erfüllt der in Kapitel 3.1 vorgestellte Semaphor-Algorithmus diese Eigenschaft nicht, da ein einzelner Semaphor-Prozess, selbst wenn er isoliert abläuft, keinen Fortschritt machen kann, wenn die Semaphor durch einen anderen Prozess blockiert ist.

8.2.3 Die *Wait-Free*-Eigenschaft

Herlihy definiert in [Her03] die *Wait-Free*-Eigenschaft folgendermaßen:

> A synchronization technique is *wait-free* if it ensures that every thread will continue to make progress in the face of arbitrary delay (or even failure) of other threads.

Diese Eigenschaft verlangt, dass jede Komponente immer einen Fortschritt macht, egal welchen Fortschritt die anderen Komponenten erzielen. Damit ist diese Eigenschaft deutlich stärker als *Obstruction-Freedom*.

Mit dem hier verwendeten Formalismus lässt sich diese Eigenschaft einfach für das Gesamtsystem übersetzen:

$$\Box\ R(CS', CS''), CSpawn_l(n, CS) \vdash \Box\ \forall\ i.\ (Act(i) \rightarrow \Diamond\ Fin(i))$$

Wann immer eine Komponente eine Operation ausführt, muss diese irgendwann beendet sein. Dabei kann man für die Gesamtumgebung annehmen, dass sie immer die globale Annahme R erfüllt.

8.2.4 Die *Lock-Free*-Eigenschaft

Die in Kapitel 6 vorgestellte Klasse der lock-freien Algorithmen erfüllt nicht die *Wait-Free*-Eigenschaft. Ein einzelner Prozess kann beliebig lange durch andere Prozesse blockiert werden, indem diese die Datenstruktur immer so abändern, dass die *CAS*-Instruktion stets fehlschlägt.

Allerdings erfüllen diese Algorithmen normalerweise die *Obstruction-Free*-Eigenschaft. Sobald ein Prozess isoliert läuft, wird die jeweilige Datenstruktur nicht mehr von der Umgebung geändert. Die *CAS*-Operation testet, ob ein bestimmter Wert (z. B. der *Top*-Zeiger beim „Treiber-Stack"-Algorithmus) von der Umgebung geändert wurde. Da bei einem isoliert laufenden Prozess dieser Wert nicht von der Umgebung geändert werden kann, wird eine isoliert ablaufende lock-freie Operation immer erfolgreich ausgeführt werden.

Intuitiv erfüllen diese Algorithmen eine noch stärkere Lebendigkeitseigenschaft als *Obstruction-Freedom*. Da eine einzelne Operation nur dann fehlschlägt, wenn die Datenstruktur geändert wurde, muss irgendein anderer Prozess mit einer erfolgreichen Operation die Datenstruktur geändert haben.

Die Lebendigkeitseigenschaft, die diesen Sachverhalt ausdrückt wird *Lock-Free*-Eigenschaft oder auch *Lock-Freedom* genannt. Sie wurde zum ersten Mal von Massalin & Pu in [MP91b] beschrieben. In [Her03] wird diese Eigenschaft folgendermaßen charakterisiert:

> It (a synchronization technique) is *lock-free* if it ensures only that some thread always makes progress.

Formalisiert werden kann diese Eigenschaft durch die Formel

$$CSpawn_l(n, CS), \Box\ R(CS', CS'') \vdash \Box\ ((\exists\ i. Act(i)) \rightarrow \Diamond\ \exists\ j.\ Fin(j)) \quad (8.1)$$

Hier muss nicht der aktive Prozess i selbst, sondern nur irgendein aktiver Prozess j terminieren. Dies ist der wesentliche Unterschied im Vergleich zur *Wait-Free*-Eigenschaft, bei der jeder aktive Prozess terminieren muss. Damit ist diese Eigenschaft echt schwächer als *Wait-Freedom*. Sie impliziert auch, dass ein isoliert laufender Prozess Fortschritt machen muss, denn dieser Prozess ist dann der einzige, der den in (8.1) geforderten Fortschritt machen kann. Somit erfüllt jeder Algorithmus, der die *Lock-Free*-Eigenschaft erfüllt auch die *Obstruction-Freedom*-Eigenschaft. Diese Hierarchie wurde formal von Dongol bewiesen [Don06].

8.3 Modularisierungstheorem für die *Lock-Freedom*-Eigenschaft

Um eine Grundidee für die Modularisierung der *Lock-Freedom*-Eigenschaft zu gewinnen, ist es sinnvoll, am Beispiel des „Treiber-Stack"-Algorithmus zu analysieren, unter welchen Voraussetzungen diese Eigenschaft erfüllt ist: Wenn eine aktive Operation i eine nicht erfolgreiche CAS-Operation ausführt, muss eine andere Operation j den *Top*-Zeiger seit der letzten Momentaufnahme von

Top verändert haben. Der *Top*-Zeiger kann ausschließlich durch eine erfolgreiche CAS-Operation der *CPush*- oder *CPop*-Operation verändert werden. Nach einem erfolgreichen CAS-Kommando endet aber jede *CPush*- bzw. *CPop*-Operation immer nach wenigen Schritten. Das heißt, falls i keine erfolgreiche CAS-Operation ausführen konnte, muss es eine aktive Operation j geben, die in wenigen Schritten erfolgreich endet.[1]

Damit ergeben sich folgende zwei Punkte, die alternativ für den Fortschritt verantwortlich sind:

1. Eine aktive *Push*- oder *Pop*-Operation ist bei diesem Algorithmus immer erfolgreich, wenn der *Top*-Zeiger von der Umgebung nicht geändert wird.

2. Der *Top*-Zeiger wird nur durch eine erfolgreiche Operation geändert.

Bei diesen beiden Punkten ist entscheidend, ob ein Teil der gemeinsamen Datenstruktur geändert wurde oder nicht. Um dieses formal ausdrücken zu können, wird ein zweistelliges Prädikat *Uchg* (für *unchanged*) eingeführt, das genau diesen Sachverhalt ausdrückt. $Uchg(CS, CS')$ bedeutet hierbei, dass die Komponente die Datenstruktur nicht geändert hat, während $Uchg(CS', CS'')$ gilt, wenn sie von der Umgebung nicht verändert wurde.

Mit diesem Prädikat ist es nun möglich, die beiden oben genannten Punkte für Fortschritt zu formalisieren. Da angenommen wird, dass der Algorithmus bei einer erfolgreicher Operation immer terminiert, kann man die beiden obigen Punkte durch die folgenden beiden Formeln ausdrücken:

$$COP(i, CS) \vdash \Box\ (\Box\ Uchg(CS', CS'') \rightarrow \Diamond\ \mathbf{last}) \tag{8.2}$$

$$COP(i, CS) \vdash \Box\ (\neg\ Uchg(CS, CS') \rightarrow \Diamond\ \mathbf{last}) \tag{8.3}$$

Wie im vorhergehenden Kapitel 7 sind diese Eigenschaften in der Regel nicht zu zeigen, wenn man nicht bestimmte Annahmen über das Verhalten der Umgebung treffen kann. Analog zum Linearisierbarkeitstheorem (siehe Kapitel 7) müssen daher die lokale Umgebungsannahmen und die Invariante in diese Sequenzen integriert werden. Außerdem können diese Formeln vereinfacht werden, indem man beide zu einer einzigen Formel kombiniert:

$$\begin{aligned} &COP(i, CS), \Box\ I(CS) \wedge I(CS') \wedge R_i(CS', CS'') \\ &\vdash \Box\ (\Box\ Uchg(CS', CS'') \vee \neg\ Uchg(CS, CS')) \rightarrow \Diamond\ \mathbf{last} \end{aligned} \tag{8.4}$$

Analog zum Linearisierbarkeitstheorem muss man hier zeigen, dass weder die lokale Umgebungsannahme R_i noch die Invariante I verletzt werden. Dies kann mit den Beweisverpflichtungen des ursprünglichen Modularisierungstheorems aus Kapitel 4.2 gezeigt werden.

Zusätzlich sind noch zwei Bedingungen für das *Uchg*-Prädikat notwendig. Zum einen darf ein Stotterschritt, der die Variable *CS* nicht ändert, auch keine geänderte Datenstruktur signalisieren. Daher muss *Uchg* in diesem Fall gelten:

$$Uchg(CS, CS) \tag{8.5}$$

[1] Wegen der Fairness des Interleaved-Operators.

Das heißt, *Uchg* ist reflexiv. Zum anderen müssen zwei Schritte, von denen keiner die Datenstruktur geändert hat, zusammengenommen die Datenstruktur wieder unverändert belassen. Dies kann durch die folgende Bedingung ausgedrückt werden:

$$Uchg(CS, CS') \wedge Uchg(CS', CS'') \rightarrow Uchg(CS, CS'') \tag{8.6}$$

Das heißt, *Uchg* muss transitiv sein.

Insgesamt ergibt sich damit das folgende Modularisierungstheorem für die *Lock-Freedom*-Eigenschaft:

Satz 8 *(modulares Lock-Freedom-Theorem).*
Seien $CSpawn_l$, COP, $ASpawn$ und AOP wie in Abschnitt 7.1 und 7.2 spezifiziert und sei I und $Init$ einstellige Prädikate und G, R, G_i und R_i zweistellige Prädikate über die Variablen $cstate$. Wenn für alle $i = 1, \ldots, n$ gilt:

1. $[COP(i, In; CS, Out)]_{In,CS,Out}, I(CS) \vdash R_i(CS', CS'') \stackrel{+}{\rightarrow} G_i(CS, CS')$
2. $G_i(CS, CS') \wedge I(CS) \rightarrow G(CS, CS') \wedge \bigwedge_{j \in \{1..n\} \wedge j \neq i} R_j(CS, CS')$
3. $R_i(CS, CS') \wedge R_i(CS', CS'') \wedge I(CS') \wedge I(CS') \rightarrow R_i(CS, CS'')$
4. $R(CS, CS') \wedge I(CS) \rightarrow \bigwedge_{j \in \{1..n\}} R_j(CS, CS')$
5. $G_i(CS, CS') \wedge I(CS) \rightarrow I(CS')$
6. $R(CS', CS'') \wedge I(CS') \rightarrow I(CS'')$
7. $G_i(CS, CS)$
8. $Init(CS) \rightarrow I(CS)$
9. $Uchg(CS, CS)$
10. $Uchg(CS, CS') \wedge Uchg(CS', CS'') \rightarrow Uchg(CS, CS'')$
11. $[COP(i, In; CS, Out)]_{In,CS,Out},$
    $\Box\ I(CS) \wedge I(CS') \wedge R_i(CS', CS'')$
    $\vdash \Box\ (\Box\ Uchg(CS', CS'') \vee \neg\ Uchg(CS, CS')) \rightarrow \Diamond\ \mathbf{last}$
12. $R(CS', CS'') \rightarrow Uchg(CS', CS'')$

gilt, dann gilt auch die *Lock-Freedom*-Eigenschaft für $CSpawn_l$:

$$\begin{aligned} & [CSpawn_l(n; Act, In, CS, Out)]_{Act,In,CS,Out}, \\ & Init(CS),\ \Box\ R(CS', CS''), \\ & \forall\ m. \neg\ Act(m),\ \Box\ \forall\ m. Act'(m) = Act''(m) \\ \vdash\ & \Box\ (\exists\ i. Act(i) \rightarrow \Diamond\ \exists\ j.\ Fin(j)) \end{aligned}$$

□

Die Eigenschaften 1. bis 8. sind identisch mit den Eigenschaften des Verfeinerungstheorems aus Kapitel 7.3. Die Beweisverpflichtungen 9. und 10. entsprechen den oben in Formel (8.5) bzw. (8.6) motivierten Eigenschaften. Die mit Formel (8.4) motivierte Lebendigkeitseigenschaft für einzelne Operationen entspricht der Beweisverpflichtung 11. Schließlich wird die Beweisverpflichtung 12. benötigt, damit die globale Umgebung nicht willkürlich die Datenstruktur ändert und somit einen Fortschritt der einzelnen Komponenten verhindert. Die gezeigte Gesamteigenschaft entspricht der mit Formel (8.1) (siehe Seite 143) motivierten Eigenschaft für *Lock-Freedom*.

8.4 Beweis des *Lock-Free*-Theorems

In diesem Abschnitt wird der Beweis des *Lock-Freedom*-Theorems beschrieben. Wie die anderen Theoreme wurde das *Lock-Freedom*-Theorem formal in KIV bewiesen.

8.4.1 Grundidee

Alle in Kapitel 7 gezeigten Sicherheitseigenschaften für *CSeq* und *CSpawn* lassen sich auch für die um das *Act*-Flag erweiterten Prozeduren $CSeq_l$ und $CSpawn_l$ zeigen. Alle Beweise können für diese beiden Prozeduren analog geführt werden. Im Folgenden werden daher die Lemmas aus Kapitel 7.4 auch für $CSeq_l$ und $CSpawn_l$ benutzt.

Konzeptionell ist die Beweisstruktur des *Lock-Freedom*-Theorems ähnlich wie der Beweis des Verfeinerungstheorems (siehe Kapitel 7.4). Anstatt der Verfeinerungseigenschaft mit einem abstrakten Programm werden hier Lebendigkeitseigenschaften für *COP*, $CSeq_l$ und $CSpawn_l$ benutzt. Die Lebendigkeitseigenschaft für *COP* ist bereits durch die Beweisverpflichtung 11. des *Lock-Freedom*-Theorems vorgegeben. Dazu müssen entsprechende Lebendigkeitseigenschaften für $CSeq_l$ und $CSpawn_l$ gefunden werden, die durch „lifting“ aus der Eigenschaft von *COP* gefolgert werden können und aus denen die Gesamteigenschaft des Theorems gefolgert werden kann. Für $CSeq_l$ wird dazu die folgende Eigenschaft benutzt:

$$\begin{aligned} \mathit{CSeq\text{-}Life}(n) \quad :\equiv \quad \Box\, (\quad & \neg\ \mathit{Uchg}(CS, CS') \vee (\mathit{Act}(n) \wedge \Box\ \mathit{Uchg}(CS', CS'')) \\ & \rightarrow \Diamond\ \mathit{Fin}(n)) \end{aligned}$$

Eine Operation terminiert garantiert, wenn sie aktiv ist und die Datenstruktur von der Umgebung nicht mehr geändert wird oder wenn die Operation die Datenstruktur selbst ändert (in diesem Fall ist sie implizit aktiv). Für $CSpawn_l$ wird diese Eigenschaft auf eine Menge von Prozessen mit den Prozessnummern 0 bis n erweitert:

$$\begin{aligned} \mathit{CSpawn\text{-}Life}(n) \quad :\equiv \quad \Box\, (\quad & \neg\ \mathit{Uchg}(CS, CS') \\ & \vee\ (\quad (\exists\ m.\ m \leq n \wedge \mathit{Act}(m)) \\ & \qquad \wedge\ \Box\ \mathit{Uchg}(CS', CS'')) \\ & \rightarrow \Diamond\ \exists\ m.\ m \leq n \wedge \mathit{Fin}(m)) \end{aligned}$$

Der Beweis, dass $CSpawn_l$ diese Eigenschaft erfüllt, wird wieder durch Induktion über die Anzahl der Komponenten geführt (siehe Lemma 15, Seite 151). Ähnlich wie beim Beweis des Verfeinerungstheorems wird im Induktionsschritt zuerst gezeigt, dass die zusätzlichen Voraussetzungen, die $CSeq_l$ und $CSpawn_l$ benötigen, um die Lebendigkeitseigenschaften zu zeigen, unter den Interleaving-operator gezogen werden können. Danach kann mittels der Kompositionalität des Interleavings die Induktionshypothese und die Lebendigkeitseigenschaft von $CSeq_l$ eingesetzt werden.

In Abschnitt 8.4.2 wird zunächst gezeigt, dass $CSeq_l$ und $CSpawn_l$ nicht die Activity-Flags anderer Komponenten ändern. Dass eine $CSeq_l$-Prozedur die Eigenschaft *CSeq-Life*(n) erfüllt, wird in Abschnitt 8.4.3 gezeigt. Um die Eigenschaft *CSpawn-Life*(n) für Spawn-Prozesse zeigen zu können, werden zunächst einige Hilfsaussagen benötigt. Diese werden in Abschnitt 8.4.4 gezeigt, die eigentliche Lebendigkeitsaussage für die Spawn-Prozedur wird dann in Abschnitt 8.4.5 erläutert. Mit diesen Voraussetzungen kann dann schließlich in Abschnitt 8.4.6 das *Lock-Freedom*-Theorem bewiesen werden.

8.4.2 Hilfssätze: Activity-Flags werden nicht von anderen Komponenten geändert

Eine für den Beweis des *Lock-Freedom*-Theorems benötigte Eigenschaft besagt, dass Prozesse nur ihr eigenes Activity-Flag verändern können. Damit kann ausgeschlossen werden, dass das Activity-Flag eines Prozesses gesetzt ist, obwohl keine Operation aktiv ist. Das folgende Lemma zeigt diese Eigenschaft für $CSeq_l$-Prozeduren:

Lemma 10 *(cseq preserves act).* Unter den Voraussetzungen des *Lock-Freedom*-Theorems (Satz 8) gilt für alle n:

$$[CSeq_l(n; Act, In, CS, Out)]_{Act,In,CS,Out}$$
$$\vdash \Box\ (\forall\ m.\ m \neq n \rightarrow Act(m) = Act'(m))$$

□

Beweis

Das Lemma kann einfach durch symbolische Ausführung und Induktion (über die Always-Formel im Sukzedent) gezeigt werden, da $CSeq_l(n)$ nur auf $Act(n)$ zugreift. □

Eine analoge Eigenschaft kann auch für $CSpawn_l$ gezeigt werden:

Lemma 11 *(cspawn preserves act).* Unter den Voraussetzungen des *Lock-Freedom*-Theorems (Satz 8) gilt für alle n:

$$[CSpawn_l(n; Act, In, CS, Out)]_{Act,In,CS,Out}$$
$$\vdash \Box\ (\forall\ m.\ m > n \rightarrow Act(m) = Act'(m))$$

□

Beweis
Das Lemma wird durch Induktion über die Anzahl der Komponenten n gezeigt. Der Induktionsanfang folgt direkt aus Lemma 10. Im Induktionsschritt wird $CSpawn_l$ expandiert und für die linke Komponente Lemma 10 und für die rechte Komponente die Induktionsvoraussetzung eingesetzt, so dass die folgende Sequenz zu zeigen ist:

$$\begin{aligned} & \Box\ (\forall\ m.\ m \neq n+1 \rightarrow Act(m) = Act'(m)) \\ \| & \Box\ (\forall\ m.\ m > n \rightarrow Act(m) = Act'(m)) \\ \vdash & \Box\ (\forall\ m.\ m > n+1 \rightarrow Act(m) = Act'(m)) \end{aligned}$$

Diese Sequenz kann ebenfalls einfach durch symbolische Ausführung und Induktion (über die Always-Formel im Sukzedent) gezeigt werden. □

8.4.3 Lebendigkeitseigenschaft von $CSeq_l$

In diesem Abschnitt wird die Lebendigkeitseigenschaft *CSeq-Life* für $CSeq_l$-Prozeduren gezeigt:

Lemma 12 *(cseq live).* Unter der Voraussetzung des *Lock-Freedom*-Theorems (Satz 8) gilt für alle n:

$$\begin{aligned} & [CSeq_l(n; Act, In, CS, Out)]_{Act,In,CS,Out}, \\ & \neg\ Act(n),\ \Box\ Act'(n) = Act''(n), \\ & \Box\ (R(n, CS', CS'') \wedge I(CS) \wedge I(CS')) \\ \vdash\ & CSeq\text{-}Life(n) \end{aligned}$$

□

Beweis

Zuerst wird der Prozeduraufruf von $CSeq_l$ mit der entsprechenden Spezifikation expandiert und der Aufruf von *COP* durch die entsprechende Lebendigkeitsaussage (siehe Beweisverpflichtung 11 des *Lock-Freedom*-Theorems) ersetzt. Das heißt, $CSeq_l$ im Antezedent wird durch die Formel

$$\begin{aligned} (\quad & \textbf{skip} \\ \vee\ & Act(n) := \text{true}; \\ & (\Box\ (\Box\ Uchg(CS', CS'') \vee \neg\ Uchg(CS, CS')) \rightarrow \Diamond\ \textbf{last}); \\ & Act(n) := \text{false})^* \end{aligned} \tag{8.7}$$

ersetzt. Durch symbolische Ausführung dieser Formel und Induktion kann nun die Gesamteigenschaft gezeigt werden. Falls $Act(n)$ gilt, kann der Seitenast, in dem $\Diamond\ Fin(n)$ zu zeigen ist, durch eine Induktion nach $\Diamond$ **last** aus Formel 8.7 gezeigt werden. □

8.4.4 Hilfssätze für den Beweis des *Lock-Freedom*-Theorems

In diesem Abschnitt werden einige für den Beweis der Lebendigkeit von *CSpawn* benötigte Hilfssätze bewiesen. Wichtige Voraussetzungen wie die lokalen Umgebungsannahmen oder das Activity-Flag dürfen nicht von der Umgebung geändert werden. Diese bedeutsamen Voraussetzungen können unter den Interleaving-Operator „gezogen“ werden. Dieses Lemma erfüllt damit eine ähnliche Funktion wie Lemma 7 (CSpawn-has-rely) für das Verfeinerungstheorem aus Kapitel 7.

Lemma 13 *(CSpawn-has-rely-and-act).* Unter der Voraussetzungen des *Lock-Freedom*-Theorems (Satz 8) gilt für alle n:

$$
\begin{array}{ll}
 & [\mathit{CSpawn}_l(n+1; \mathit{In}, \mathit{CS}, \mathit{Out})]_{\mathit{In},\mathit{CS},\mathit{Out}}, \\
 & \forall\, m.\ m \leq n+1 \rightarrow \neg\ \mathit{Act}(m), \\
 & \Box\ \forall\, m.\ (m \leq n+1 \rightarrow \mathit{Act}'(m) = \mathit{Act}''(m)), \\
 & \Box\ (\mathit{Rto}(n+1, \mathit{CS}', \mathit{CS}'') \wedge I(\mathit{CS}) \wedge I(\mathit{CS}'')) \\
\vdash & \quad [\mathit{CSeq}_l(n+1; \mathit{In}, \mathit{CS}, \mathit{Out})]_{\mathit{In},\mathit{CS},\mathit{Out}} \\
 & \wedge\ \neg\ \mathit{Act}(n+1) \\
 & \wedge\ \Box\ \mathit{Act}'(n+1) = \mathit{Act}''(n+1) \\
 & \wedge\ \Box\ (R(n+1, \mathit{CS}', \mathit{CS}'') \wedge I(\mathit{CS}) \wedge I(\mathit{CS}'')) \\
\| & \quad [\mathit{CSpawn}_l(n; \mathit{In}, \mathit{CS}, \mathit{Out})]_{\mathit{In},\mathit{CS},\mathit{Out}} \\
 & \wedge\ \forall\, m.\ m \leq n \rightarrow \neg\ \mathit{Act}(m) \\
 & \wedge\ \Box\ \forall\, m.\ (m \leq n \rightarrow \mathit{Act}'(m) = \mathit{Act}''(m)) \\
 & \wedge\ \Box\ (\mathit{Rto}(n, \mathit{CS}', \mathit{CS}'') \wedge I(\mathit{CS}) \wedge I(\mathit{CS}''))
\end{array}
$$

□

Beweis

Der Beweis, dass die lokalen Rely-Bedingungen innerhalb des Interleavings nie verletzt sind, wurde schon in Lemma 7 für *CSpawn* ohne *Act*-Variable gezeigt. Dieser Beweis kann für das hier verwendete CSpawn_l analog geführt werden. Dass das entsprechende *Act*-Flag zu Beginn des Ablaufs jeder Komponente nicht gesetzt ist und dass das Flag von der Umgebung der Komponente niemals verändert wird, kann auf die gleiche Weise durch Fallunterscheidung und Beweis durch Widerspruch bewiesen werden. □

Eine weitere wichtige Eigenschaft ist, dass die Lebendigkeitseigenschaften „geliftet“ werden können: Wenn zu einem Spawn-Prozess, der *CSpawn-Life*(n) erfüllt, noch ein weiterer *CSeq* Prozess mit *CSeq-Life*$(n+1)$ hinzugenommen wird, erfüllt das Gesamtsystem *CSpawn-Life*$(n+1)$. Diese Eigenschaft wird für den Induktionsschritt im Beweis, dass der Spawn-Prozess *CSpawn-Life*(n) erfüllt, benutzt.

Lemma 14 *(Lifting lemma für lock-freedom).* Unter den Voraussetzungen des *Lock-Freedom*-Theorems (Satz 8) gilt für alle n:

$$\begin{array}{l} \quad \Box\ (\forall\ m.\ m \neq n+1 \rightarrow \mathit{Act}(m) = \mathit{Act}'(m)) \\ \quad \wedge\ \mathit{CSeq\text{-}Life}(n+1) \\ \parallel \quad \Box\ \forall\ m.\ m > n \rightarrow \mathit{Act}(m) = \mathit{Act}'(m) \\ \quad \wedge\ \mathit{CSpawn\text{-}Life}(n), \\ \Box\ \forall\ m.\ (m \leq n+1 \rightarrow \mathit{Act}'(m) = \mathit{Act}''(m)), \\ \Box\ (\mathit{Rto}(n+1, \mathit{CS}', \mathit{CS}'') \wedge I(\mathit{CS}) \wedge I(\mathit{CS}'')) \\ \vdash\ \mathit{CSpawn\text{-}Life}(n+1) \end{array}$$

□

Beweis

Im Folgenden wird die Always-Formel *CSpawn-Life*$(n+1)$ mit $\Box\ \delta$ abgekürzt. Der Beweis wird durch Induktion und symbolische Ausführung geführt. Aus dem Always-Operator $\Box\ \delta$ im Sukzedent kann zunächst eine Induktionshypothese generiert werden. Dabei wird die Formel $\Box\ \delta$ aus dem Sukzedent zu $N = N'' + 1$ **until** $\neg\ \delta$ im Antezedent umgeschrieben (siehe Kapitel 2.5.1). Nun kann mit der Normalformregel *untl* (siehe Tabelle 2.3 auf Seite 27) eine Fallunterscheidung durchgeführt werden, ob δ im aktuellen Zustand verletzt ist oder ob mit $\circ\ N = N'' + 1$ **until** $\neg\ \delta$ die Until-Formel im nächsten Zustand erfüllt ist. Die letztere Formel kann mit einem symbolischen Ausführungsschritt gezeigt werden. Dabei entstehen durch das Interleaving sehr viele unterschiedliche Fälle, die jedoch alle durch Anwendung der Induktionshypothese geschlossen werden können. Falls δ im aktuellen Zustand verletzt ist, muss gezeigt werden, dass dies zu einem Widerspruch führt (die vorher generierte Induktionshypothese kann in diesem Fall nicht benutzt werden, da keinerlei Wissen über N mehr vorhanden ist). Damit δ verletzt sein kann, muss entweder

$$\neg\ \mathit{Uchg}(\mathit{CS}, \mathit{CS}') \tag{8.8}$$

oder

$$(\exists\ m.\ m \leq n+1 \wedge \mathit{Act}(m)) \wedge \Box\ \mathit{Uchg}(\mathit{CS}', \mathit{CS}'') \tag{8.9}$$

gelten. In beiden Fällen muss nachgewiesen werden, dass

$$\Diamond\ \exists\ m.\ m \leq n+1 \wedge \mathit{Fin}(m) \tag{8.10}$$

gilt, um den Widerspruch zu zeigen:

- Falls (8.8) gilt:
  Die Komponente, die den nächsten Schritt vollzieht, ändert die Datenstruktur und daher wird die Operation irgendwann terminieren. Das heißt, für diese Komponente gilt $\Diamond\ \mathit{Fin}(m)$ für ein entsprechendes m. Aus dieser Eventually-Formel kann eine Induktionshypothese generiert werden und damit kann $\Diamond\ \exists\ m.\ m \leq n+1 \wedge \mathit{Fin}(m)$ durch symbolische Ausführung und Induktion gezeigt werden.

- Falls (8.9) gilt:
  In diesem Fall muss zuerst unterschieden werden, ob eine Operation in der linken Komponente (d. h. $Act(n+1)$) oder der rechten Komponente (d. h. mit $Act(m)$ für ein $m \leq n$) aktiv ist. Hier wird nur der Fall $Act(n+1)$ für die linke Komponente behandelt, der symmetrische Fall für die rechte Komponente kann analog bewiesen werden. Um zu zeigen, dass die linke Komponente irgendwann $Fin(n+1)$ erreicht, ist noch eine weitere Fallunterscheidung notwendig, da die rechte Komponente die Datenstruktur CS nicht ändern darf:

  - Falls die rechte Komponente irgendwann einen Schritt macht, in dem $\neg\ Uchg(CS, CS')$ gilt, wird danach irgendwann auch

    $$\exists\ m.\ m \leq n+1 \wedge Fin(m)$$

    erfüllt sein. Damit ist auch Formel (8.10) erfüllt, womit der Widerspruch gezeigt wird.

  - Falls für die rechte Komponente $\Box\ Uchg(CS, CS')$ gilt, kann mit (8.9) gezeigt werden, dass für die Umgebungsschritte der linken Komponente $\Box\ Uchg(CS', CS'')$ erfüllt ist. Da auch $Act(n+1)$ gilt, erfüllt die linke Komponente irgendwann $Fin(n+1)$, also gilt die Formel (8.10), womit der Widerspruch gezeigt ist.

□

Dieser Beweis ist der Schlüsselbeweis für den Beweis des Lock-Freedom-Theorems. Er benötigt über 1200 Beweisschritte, wovon etwa 500 interaktiv anzuwenden sind. Wegen dieser Größe wurde der Beweis nur knapp skizziert und die wesentlichsten Schritte erläutert. Die Hauptschwierigkeit dabei ist die richtige Vorgehensweise. Mit einer falschen Generalisierung kann ein einzelner symbolischer Ausführungsschritt über 100 Prämissen erzeugen (z. B. wenn die Fallunterscheidung, ob δ verletzt ist, nicht vor dem Schritt getroffen wird). Dabei ist die Beweisstrategie der symbolischen Ausführung eine große Hilfe, da sie die Beweisschritte verständlicher macht, so dass es leichter ist, die passenden Generalisierungen zu finden.

8.4.5 Lebendigkeitseigenschaft von CSpawn

Mit den im vorhergehenden Abschnitt gezeigten Lemmas kann nun bewiesen werden, dass ein CSpawn-Prozess die Eigenschaft *CSpawn-Life* erfüllt:

Lemma 15 *(cspawn live).* Unter den Voraussetzungen des *Lock-Freedom*-Theorems (Satz 8) gilt für alle n:

$$
\begin{aligned}
& [\mathit{CSpawn}_l(n; \mathit{Act}, \mathit{In}, \mathit{CS}, \mathit{Out})]_{\mathit{Act},\mathit{In},\mathit{CS},\mathit{Out}}, \\
& \forall\, m.\ m \leq n \rightarrow \neg\ \mathit{Act}(m), \\
& \Box\ \forall\, m.\ (m \leq n \rightarrow \mathit{Act}'(m) = \mathit{Act}''(m)), \\
& \Box\ (\mathit{Rto}(n, \mathit{CS}', \mathit{CS}'') \wedge \mathit{I}(\mathit{CS}) \wedge \mathit{I}(\mathit{CS}'')) \\
\vdash\ & \mathit{CSpawn\text{-}Life}(n)
\end{aligned}
$$

□

Beweis

Der Beweis wird durch Induktion über die Anzahl der Komponenten n geführt. Der Induktionsanfang kann direkt mit Lemma 12 gezeigt werden. Zum Beweis des Induktionsschritts wird in einem ersten Schritt Lemma 13 angewandt, um CSpawn_l dementsprechend umzuschreiben. Damit lässt sich Lemma 12 und Lemma 10 auf die $(n+1)$-te Komponente, sowie die Induktionsvoraussetzung und Lemma 11 auf das Restsystem anwenden, so dass die Sequenz die gleiche Form hat wie der Antezedent von Lemma 14. Mit diesem lässt sich dann die zu zeigende Eigenschaft beweisen. □

8.4.6 Beweis des *Lock-Freedom*-Theorems

Schließlich kann das *Lock-Freedom*-Theorem von Seite 145 selbst bewiesen werden. Die zu beweisende Sequenz lautet:

$$
\begin{aligned}
& [\mathit{CSpawn}_l(n; \mathit{Act}, \mathit{In}, \mathit{CS}, \mathit{Out})]_{\mathit{Act},\mathit{In},\mathit{CS},\mathit{Out}}, \\
& \mathit{Init}(\mathit{CS}),\ \Box\ R(\mathit{CS}', \mathit{CS}''), \\
& \forall\, m.\neg\ \mathit{Act}(m),\ \Box\ \forall\, m.\mathit{Act}'(m) = \mathit{Act}''(m) \\
\vdash\ & \Box\ (\exists\, i.\mathit{Act}(i) \rightarrow \Diamond\ \exists\, j.\ \mathit{Fin}(j))
\end{aligned}
$$

Beweis

Zum Beweis dieser Sequenz wird in einem ersten Schritt Lemma 15 angewandt, um die Eigenschaft $\mathit{CSpawn\text{-}Life}(n)$ aus CSpawn_l zu folgern. Dabei werden die Beweisverpflichtungen 8. und 6. des *Lock-Freedom*-Theorems benutzt, um nachzuweisen, dass die Invariante $\mathit{I}(\mathit{CS})$ im ersten Zustand gilt und dass immer die lokalen Annahmen der Komponenten von der globalen Umgebung erfüllt sind: $\Box\ \mathit{Rto}(n, \mathit{CS}', \mathit{CS}'')$. Dass die Invariante immer erfüllt ist, kann dann mit Lemma 6 (siehe Seite 117) gezeigt werden. Da mit Beweisverpflichtung 12. des *Lock-Freedom*-Theorems aus $\Box\ R(\mathit{CS}', \mathit{CS}'')$ die Formel

$$\Box\ \mathit{Uchg}(\mathit{CS}', \mathit{CS}'')$$

folgt, kann aus der Formel $\mathit{CSpawn\text{-}Life}(n)$ nun die zu zeigende Eigenschaft mit Induktion und symbolischer Ausführung bewiesen werden. □

8.5 Fallstudie: *Lock-Freedom* des „Treiber-Stack“- Algorithmus

Die Anwendung des *Lock-Freedom*-Theorems wird in diesem Abschnitt anhand des „Treiber Stack“-Algorithmus demonstriert. Die globalen und lokalen Rely- bzw. Guarantee-Bedingungen sowie die Invariante und die Initialbedingung können alle unverändert aus dem Korrektheitsbeweis (siehe Kapitel 7.6) des Stack Algorithmus übernommen werden. Damit müssen auch die Beweisverpflichtungen 1. - 8. nicht erneut bewiesen werden.

Da bei jeder Änderung des Stacks durch die Operationen *CPush* oder *CPop* der *Top*-Zeiger modifiziert wird, kann das Prädikat *Uchg* durch

$$Uchg(CS, CS') \quad :\leftrightarrow \quad Top' = Top \tag{8.11}$$

definiert werden. Die Beweisverpflichtungen 9. und 10. können mit dieser Definition leicht gezeigt werden. In der letzten Beweisverpflichtung (11.) des *Lock-Freedom*-Theorems muss gezeigt werden, dass eine *COP*-Operation immer terminiert, wenn sie die Datenstruktur ändert oder die Datenstruktur von ihrer Umgebung nicht mehr geändert wird. Der Beweis kann in die folgenden zwei Teilbeweise aufgespalten werden:

$$\begin{aligned} & [COP(i, In; CS, Out)]_{In,CS,Out}, \\ & \Box\ I(CS) \wedge I(CS') \wedge R_i(CS', CS'') \\ \vdash\ & \Box\ (\neg\ Uchg(CS, CS') \rightarrow \Diamond\ \mathbf{last}) \end{aligned} \tag{8.12}$$

$$\begin{aligned} & [COP(i, In; CS, Out)]_{In,CS,Out}, \\ & \Box\ I(CS) \wedge I(CS') \wedge R_i(CS', CS'') \\ \vdash\ & \Box\ (\Box\ Uchg(CS', CS'') \rightarrow \Diamond\ \mathbf{last}) \end{aligned} \tag{8.13}$$

Diese beiden Sequenzen entsprechen den beiden Sequenzen (8.2) und (8.3) aus Abschnitt 8.3. Die beiden Beweise dieser Eigenschaften werden nochmals in jeweils einen Beweis für *CPush* und für *CPop* aufgespalten. In allen vier Beweisen lässt sich aus der Always-Formel im Sukzedent eine Induktionshypothese generieren, so dass der Beweis durch symbolische Ausführung und Induktion geführt werden kann.

In den beiden *CPush*- und *CPop*- Beweisen der Eigenschaft (8.13) muss nach einem geglückten CAS-Kommando gezeigt werden, dass die Operation in endlich vielen Schritten terminiert. Dies gelingt einfach, da in diesem Fall die Schleife terminiert und so das Operationsende nach wenigen Schritten erreicht ist.

Bei den beiden Beweisen der Eigenschaft (8.12) muss in jedem Zustand zusätzlich gezeigt werden, dass die Operation terminiert, wenn ab diesem Zustand die Umgebung die Variable *Top* nicht mehr ändert. Falls der Zustand, ab dem dies gezeigt werden muss, nach der Momentaufnahme der Variable *Top* (Zeile 7 bei der Operation *CPush*, bzw. Zeile 5 bei Operation *CPop*) liegt, kann die nächste CAS-Operation fehlschlagen und die Schleife muss noch einmal komplett durchlaufen werden. Bei diesem erneuten Durchlauf ist die CAS-Operation

aber immer erfolgreich, da die Variable *Top* nicht mehr geändert wird. Nach dem erfolgreichen CAS-Kommando terminieren beide Operation nach wenigen Schritten.

Bis auf die Generierung der Induktionshypothese lassen sich alle vier Beweise nahezu vollständig automatisieren. Nur zu Beginn muss der erste symbolische Ausführungsschritt manuell ausgeführt und die Induktionshypothese interaktiv ausgewählt werden. Damit ist nahezu der gesamte Beweisaufwand für die Lebendigkeit in die generische Theorie verschoben. Zusätzlich wird ausgenutzt, dass eine Theorie für Korrektheit und Lebendigkeit benutzt wird, so dass sich einige der für die Korrektheit bewiesenen Eigenschaften auch für die Lebendigkeit nutzen lassen

8.6 Andere Arbeiten zur *Lock-Freedom*-Eigenschaft

Für die Verifikation der Lebendigkeit von lock-freien Algorithmen und zum Nachweis der *Lock-Freedom*-Eigenschaften existieren bisher nur zwei andere Ansätze: Colvin & Dongol beschreiben in [CD07, CD09] die Verifikation einiger lock-freier Algorithmen, unter anderem auch eine Implementierung eines lock-freien Queue-Algorithmus basierend auf Arrays [CG05]. Bei diesem Verifikationsansatz ist die explizite Konstruktion einer wohlfundierten Ordnung auf den Programmzählern notwendig. Zusätzlich müssen einige Anweisungen als „progress actions“ ausgezeichnet werden (diese entsprechen den Anweisungen, in denen *Uchg* falsch ist). Zum Nachweis der *Lock-Freedom*-Eigenschaft wird gezeigt, dass jeder Schritt entweder einen Fortschritt garantiert (d. h., er führt eine progress action aus) oder der Zustand nimmt bezüglich der wohlfundierten Ordnung weiter ab. Im Vergleich zum Ansatz von Colvin & Dongol benötigt die in dieser Arbeit beschriebene Technik nicht die explizite Konstruktion einer wohlfundierten Ordnung. Diese wird implizit durch die Induktion während der symbolischen Ausführung konstruiert.

Ein zweiter Ansatz, der eine hohe Automatisierung erreicht, wurde von Gotsman et al. [GCPV09] beschrieben. Dieser Ansatz benutzt neben der Rely-Guarantee-Methode noch die Techniken *Separation Logic* [ORY01] und *Shape-Analysis* [SRW02]. Durch die Kombination mehrerer Werkzeuge können einige lock-freie Algorithmen einschließlich der „Michael-Scott“-Queue (siehe Kapitel 9) automatisch verifiziert werden. Die Theorie wurde dabei allerdings per Hand bewiesen. Weitere Unterschiede zu dem in der vorliegenden Arbeit vorgestellten Ansatz sind: Bei Gotsman et al. wird keine Folge von *COP*-Aufrufen betrachtet (wie *CSeq*), sondern es werden nur jeweils einzelne *COP*-Aufrufe analysiert. Dazu müssen einige Annahmen über die Symmetrie getroffen werden. Zusätzlich wird eine Beweisverpflichtung benötigt, dass kein Prozess die Datenstruktur unendlich oft verändert. Bei dem in der vorliegenden Arbeit vorgestellten Ansatz muss dagegen, nachdem *Uchg* verletzt wurde, nur die Terminierung eines Prozesses gezeigt werden. Das benutzte Shape-Analysis Werkzeug des Ansatzes von Gotsman et al. unterstützt momentan nur Datenstrukturen, die auf verketteten Listen basieren. Die Autoren glauben aber, dass diese Einschränkung mit spezialisierten Werkzeugen behoben werden kann. Eine solche

prinzipielle Einschränkung besteht bei dem hier beschriebenen Ansatz nicht.

Diese beiden Ansätze gehen von einem möglicherweise nicht-fairen Ablauf aus, was eher zutrifft als die hier gemachte Annahme von schwacher Fairness. Eine eingehende Analyse zeigt, dass Fairness nur benötigt wird, um zu beweisen, dass ein Prozess nicht verhungern wird zugunsten eines anderen Prozesses, der nur skip-Schritte ausführt. Dieser Fall kann in den beiden anderen Ansätze nicht auftreten, da nur Prozesse berücksichtigt werden, die in einer Endlos-Schleife nur Aufrufe von *COP* ohne weitere Instruktionen ausführen. Wenn die Implementierung von *CSeq* aus dieser Arbeit durch eine derartige Schleife ersetzt wird, kann der faire Interleaving-Operator durch einen nicht-fairen ersetzt werden. Allerdings ist die generische Formulierung von *CSeq* realistischer, da ein Prozess normalerweise auch andere Anweisungen durchführt oder terminiert anstatt nur wiederholt *COP* aufzurufen. Trotzdem kann auch für die einfachere while-Schleife mit nicht-fairem Interleaving das Lock-Freedom Theorem gezeigt werden. Allerdings wurde der formale Beweis bisher nur für Algorithmen ohne blockierende Schritte gezeigt. Das Lock-Freedom Theorem selbst kann aber für einen solchen Ansatz mit nicht-fairem Interleaving unverändert übernommen werden.

Der Beweis verläuft ganz ähnliche wie der ursprüngliche, zumal die Regeln der symbolischen Ausführung für nicht-faires Interleaving die gleichen sind wie für faires. Der wesentliche Unterschied besteht darin, dass ohne schwache Fairness nicht länger garantiert werden kann, dass der erste von zwei parallelen Komponenten irgendwann einen Schritt ausführen wird. Statt dessen ist eine zusätzlichen Fallunterscheidung notwendig, dass entweder die erste Komponente irgendwann einen Schritt ausführt oder dass nur noch die zweite Komponente Schritte durchführt.

8.7 Zusammenfassung

Insgesamt wurde mit ITL$^+$ ein Ansatz zum Nachweis einer globalen *Lock-Freedom*-Eigenschaft gezeigt. Dabei müssen zum Beweis nur prozess-lokale Eigenschaften nachgewiesen werden. Bei diesem Ansatz kann sowohl die Theorie als auch die Fallstudie mit dem ITL$^+$-Kalkül bewiesen werden. Anhand des „Treiber-Stack“-Algorithmus wurde die Anwendung dieser Technik demonstriert. Demnach können die Rely- und Guarantee-Bedingungen aus dem Korrektheitsbeweis direkt übernommen werden. Insgesamt wurde damit eine intuitive Technik entwickelt, mit der die Lebendigkeit von lock-freien Algorithmen nachgewiesen werden kann.

Kapitel 9

Der „Michael-Scott“-Queue Algorithmus

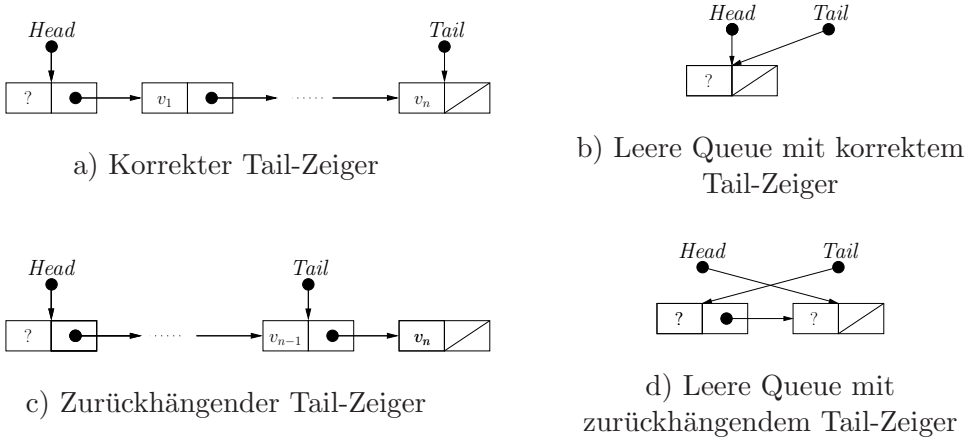
In den vorangehenden drei Kapiteln wurden lock-freie Algorithmen zusammen mit modularen Verifikationstechniken vorgestellt, um deren Korrektheit und Lebendigkeit nachzuweisen. Der „Treiber-Stack“-Algorithmus wurde als Fallstudie benutzt, um diese Verifikationstechniken zu erläutern. Dieser Algorithmus ist ein relativ einfacher lock-freier Algorithmus und daher als einführendes Beispiel geeignet.

In diesem Kapitel wird mit einem lock-freien Queue-Algorithmus eine weitere Fallstudie beschrieben. Dabei handelt es sich um eine von Doherty et al. [DGLM04] optimierte Version einer von Michael & Scott vorgestellten lock-freien Queue-Implementierung [MS96]. Dieser Algorithmus ist deutlich komplexer als der „Treiber“-Stack-Algorithmus. Zusätzlich lässt sich an diesem Beispiel eine Besonderheit unserer Verifikationsmethode darstellen, da eine Verifikation des Queue-Algorithmus mittels Verfeinerung normalerweise Forward- und Backwardsimulation benötigt. Dies ist beim hier beschriebenen Ansatz nicht der Fall.

Ein Teil der hier beschriebenen Ergebnisse wurde vom Autor bereits in [BSTR09, TBSR10] veröffentlicht.

9.1 Spezifikation der Queue

Der konkrete Queue Algorithmus wird mit einer einfach verketteten Liste spezifiziert. Diese wird im Heap *Hp* gespeichert. Der Anfang der Queue wird über den Zeiger *Head* markiert. Dabei ist der erste Knoten, auf den *Head* zeigt, ein Pseudoknoten, dessen Wert für die Queue irrelevant ist. Das heißt, erst der Nachfolgeknoten dieses Pseudoknotens enthält den ersten Wert der Queue. Durch Verwendung des Pseudoknotens lassen sich Fallunterunterscheidungen, die für die leere Queue notwendig wären, vermeiden. Das Ende der Queue wird durch den Nullzeiger repräsentiert. Zusätzlich wird das Ende mit dem Zeiger *Tail* referenziert, der einen direkten Zugriff auf das Ende der Queue ermöglicht. Es ist jedoch erlaubt, dass *Tail* „zurückhängt“, das heißt, der *Tail*-Zeiger verweist nicht auf den letzten Knoten, sondern auf einen vorhergehenden Knoten.



a) Korrekter Tail-Zeiger

b) Leere Queue mit korrektem Tail-Zeiger

c) Zurückhängender Tail-Zeiger

d) Leere Queue mit zurückhängendem Tail-Zeiger

Abbildung 9.1: Queue Representationen

Abbildung 9.1 zeigt die verschiedenen Queue-Representationen mit korrektem oder zurückhängendem *Tail*-Zeiger.

Zum Zugriff auf die Queue werden zwei Operationen benutzt. Die *Enqueue*-Operation fügt einen Datenwert an das Ende der Queue an. Um Elemente aus der Queue zu entfernen, wird die *Dequeue*-Operation benutzt. Falls die Queue nicht leer ist, entfernt diese Operation das vorderste Element und gibt dessen Datenwert aus. Andernfalls wird nur der spezielle Rückgabewert $\emptyset$ ausgegeben.

9.1.1 Enqueue-Operation

In den Zeilen *E2* und *E3* von Abbildung 9.2 wird eine neue Zelle *ENew* alloziert, die an die Queue angehängt werden soll. Zusätzlich wird die Variable *ESucc* mit false initialisiert. Diese Variable wird wie die Variable *USuccess* und *OSuccess* beim Stack-Algorithmus dazu benutzt, um festzustellen, ob die *CAS*-Operation am Linearisierungspunkt erfolgreich war oder nicht. In Zeile *E4* wird der in die Queue einzufügende Datenwert in die Zelle *ENew* geschrieben. Danach werden die folgenden Schritte in einer Schleife ausgeführt bis die Zelle erfolgreich an die Queue angehängt worden ist. Zunächst werden Momentaufnahmen des *Tail*-Zeigers in der Variable *ETl* und des *.nxt*-Zeigers der Zelle, auf die *Tail* verweist in der Variable *ENxt* gespeichert (Zeile *E6* und *E7*). Die nächsten Schritte werden nur ausgeführt, falls sich bis dahin der *Tail*-Zeiger nicht verändert hat. Ansonsten wird die Schleife erneut ausgeführt. Falls die Momentaufnahme *ENxt* der Nullzeiger ist, verweist *Tail* auf das Ende der Queue. In diesem Fall wird versucht, mittels einer *CAS*-Operation die neu allozierte Zelle an das Ende der Queue anzuhängen (Zeile *E10*). Falls der *Tail*-Zeiger zurückhängt, wird mittels einer *CAS*-Operation versucht, *Tail* auf die nächste Zelle zu verschieben (Zeile *E12*). Dies ist gleichzeitig die letzte Zelle, da keine weiteren Enqueue-Operationen erfolgreich sein können, solange der *Tail*-Zeiger zurückhängt. Nachdem eine Zelle erfolgreich an die Queue angehängt wurde, wird in Zeile *E16* versucht, den korrekten *Tail*-Zeiger wieder herzustellen. Diese Operation muss nur einmal aufgerufen werden, da im Falle eines Fehlschlags

```
E1   CEnq(v; Hp, Tail, ENew, ETl, ENxt, ESucc) {
E2       choose RefNew with RefNew ≠ null ∧ ¬ RefNew ∈ Hp in {
E3           Hp := Hp ∪ {RefNew}, ENew := RefNew, ESucc := false;
E4           Hp[ENew] := v × null;
E5           while ¬ ESucc do {
E6               ETl := Tail;
E7               ENxt := Hp[ETl].nxt;
E8               if ETl = Tail then {
E9                   if ENxt = null then {
E10                      CAS(ENxt, ENew; Hp[ETl].nxt, ESucc);
E11                  } else {
E12                      CAS(ETl, ENxt; Tail);
E13                  }
E14              }
E15          }
E16          CAS(ETl, ENew; Tail);
E17      }
E18  }
```

Abbildung 9.2: Formale Spezifikation des konkreten Enqueue-Algorithmus

schon ein anderer Prozess *Tail* korrigiert hat.

9.1.2 Dequeue-Operation

Der konkrete Dequeue-Algorithmus besteht im Wesentlichen aus einer Schleife (Zeile *D4* bis *D22*), die solange ausgeführt wird, bis entweder eine leere Queue angetroffen wird oder die Dequeue-Operation erfolgreich ausgeführt wurde. Am Anfang der Schleife werden in den Zeilen *D5* und *D6* zunächst Momentaufnahmen vom *Head*-Zeiger und dem Zeiger auf die erste Zelle der Queue (das ist der *.nxt*-Zeiger der Zelle, auf die *Head* verweist) gespeichert. Der Test in Zeile *D7* prüft, ob sich die Momentaufnahme von *Head* geändert hat. Er dient zur Effizienzsteigerung. In Zeile *D8* wird getestet, ob die Momentaufnahme *DNxt* aus Zeile *D6* einen Nullzeiger enthält. Falls *DNxt* den Nullzeiger enthält, war die Queue zum Zeitpunkt der Momentaufnahme leer und die Dequeue-Operation kann mit dem Rückgabewert $\emptyset$ beendet werden (Zeile *D9* und *D10*). Falls *DNxt* nicht den Nullzeiger enthält, wird der Datenwert der vordersten Queue-Zelle gesichert (Zeile *D12*) und versucht, diese mit einer *CAS*-Operation aus der Queue zu entfernen. Falls die Operation erfolgreich war und sich danach *Head*- und *Tail*-Zeiger überlappen (dies entspricht der Situation „leere Queue mit zurückhängendem Tail-Zeiger“ aus Abbildung 9.1) wird durch eine zusätzliche *CAS*-Operation versucht, diese Situation zu beheben (Zeile *D14* bis *D17*). Zuletzt wird in Zeile *D23* der zuvor gespeicherte Rückgabewert an die Ausgabevariable *O* übergeben.

```
D1   CDeq(; Hp, Head, Tail, DHd, DNxt, DSucc, O) {
D2     let Lo = ∅, DTl = NULL in {
D3       DSucc := false;
D4       while ¬ DSucc do{
D5         DHd := Head;
D6         DNxt := Hp[DHd].nxt;
D7         if DHd = Head then {
D8           if DNxt = NULL then {
D9             Lo := ∅;
D10            DSucc := true;
D11          } else {
D12            Lo := Hp[DNxt].val;
D13            CAS(DHd, DNxt; Head, DSucc);
D14            if DSucc then {
D15              DTl := Tail;
D16              if DTl = DHd then {
D17                CAS(DTl, DNxt; Tail);
D18              }
D19            }
D20          }
D21        }
D22      }
D23      O := Lo;
D24    }
D25  }
```

Abbildung 9.3: Formale Spezifikation des konkreten Dequeue-Algorithmus

```
AE1  AEnq(v; Queue) {
AE2    skip*;
AE3    Queue := enq(Queue, v);
AE4    skip*
AE5  }
```

Abbildung 9.4: Abstrakte enqueue Operation

```
AD1   ADeq(; Queue, O) {
AD2       let Lo = empty in {
AD3           skip* :
AD4           if* Queue ≠ emptyq then{
AD5               Lo := head(Queue),
AD6               Queue := deq(Queue);
AD7           }
AD8           skip*;
AD9           O := Lo
AD10      }
AD11  }
```

Abbildung 9.5: Abstrakte dequeue Operation

9.2 Abstrakte Queue Operationen

Wie bei der „Treiber-Stack“-Fallstudie muss für die „Michael-Scott“-Queue eine abstrakte Datenstruktur sowie die entsprechenden atomaren abstrakten Operationen definiert werden. Die abstrakte Datenstruktur *Queue* kann wie der Stack algebraisch von Listen abgeleitet werden. Dabei bezeichnet $\emptyset$ die leere Liste/Queue, die Operation $enq(Queue, v)$ fügt das Element v hinten an die Liste *Stack* an, $head(Queue)$ gibt das erste Element der *Queue* aus und $rest(Queue)$ entfernt das erste Element der Liste *Queue*.

Mit diesen atomaren Operationen kann die zu verfeinernde abstrakte Operation *AOP* spezifiziert werden:

$$
\begin{array}{l}
AOP(i, In; Queue, Out)\ \{ \\
\quad AEnq(In(i); Queue) \\
\quad\ \vee \\
\quad APop(; Queue, Out(i)) \\
\}
\end{array}
$$

Die Operation *AOP* ruft entweder eine *AEnq*- oder eine *APop*-Operation auf. Die Spezifikation dieser Operationen ist in Abbildung 9.4 und 9.5 dargestellt. Beide Operationen können nach dem in Kapitel 7 dargestellten Schema spezifiziert werden.

9.3 Linearisierungspunkte des Queue-Algorithmus

Wie in Kapitel 7 beschrieben, wird die Korrektheit einer lock-freien Operation nachgewiesen, indem man zeigt, dass an einem bestimmten Punkt der Operation eine äquivalente atomare abstrakte Operation ausgeführt werden kann. D.h. der abstrakte Effekt der Operation wird in diesem wirksam. Dieser Punkt wird Linearisierungspunkt genannt (da er aus dem Korrektheitskriterium „Linearisierbarkeit“ abgeleitet ist). Beim „Treiber-Stack“-Beispiel kann der Linearisierungspunkt ohne weiteres angegeben werden. In beiden Operationen ist

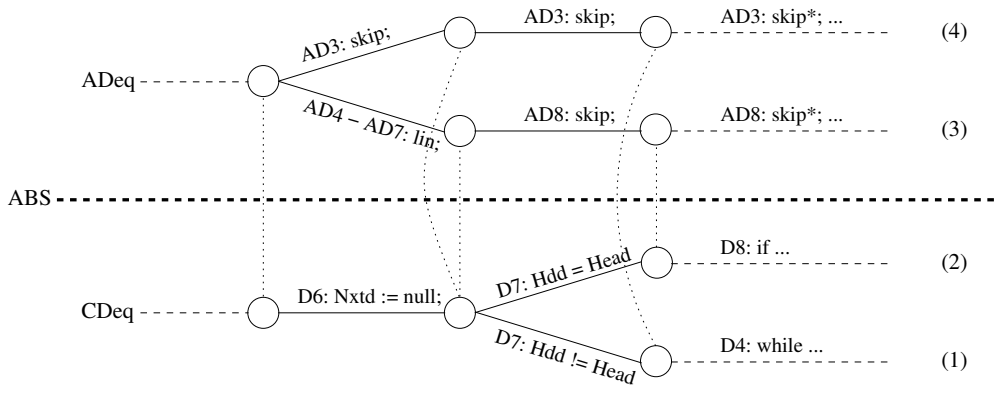


Abbildung 9.6: Dequeue Empty – Zusammenhang zwischen konkretem und abstraktem Ablauf

der Linearisierungspunkt die CAS-Operation. Lediglich bei einem leeren Stack liegt der Linearisierungspunkt der Pop-Operation am Zeitpunkt der Momentaufnahme von *Top*. Aber in all diesen Fällen steht der Linearisierungspunkt zum Zeitpunkt der Ausführung der entsprechenden Anweisung fest.

Für die Enqueue-Operation des „Michael-Scott-Queue“-Algorithmus stimmt der Linearisierungspunkt ebenfalls mit der *CAS*-Operation in Zeile *E10* überein. Genau in diesem Schritt wird die abstrakte Repräsentation der Queue verändert. Zwar wird in der Enqueue-Operation noch eine zweite *CAS*-Operation in Zeile *E12* aufgerufen, diese sorgt aber lediglich dafür, dass ein zurückhängender *Tail*-Zeiger wieder auf die letzte Zelle der Queue zeigt. Bei dieser Operation wird die abstrakte Repräsentation der Queue nicht verändert.

Bei der Dequeue-Operation ist der Linearisierungspunkt jedoch nicht so einfach zu bestimmen, da er nicht für alle Abläufe gleich ist. Der einfachere Fall tritt auf, wenn die Momentaufnahme, die vom *.nxt*-Zeiger der von *Head* referenzierten Zelle in Zeile *D6* gemacht wird, nicht der Null-Zeiger ist. In diesem Fall führt der Algorithmus den Else-Zweig in Zeile *D11* aus. Wenn die CAS-Operation in Zeile *D13* erfolgreich ist, wird in diesem Schritt die abstrakte Repräsentation der Queue verändert und damit ist dieser Schritt der Linearisierungspunkt. Falls jedoch die Momentaufnahme *DNxt* aus Zeile *D6* den Null-Zeiger (also die Repräsentation für die leere Queue) enthält, wird entweder der Schleifendurchlauf in Zeile *D7* abgebrochen oder der if-Fall in Zeile *D8* gewählt und die Operation mit dem Ergebnis $\emptyset$ in Zeile *D10* beendet. Im zweiten Fall muss also ein Linearisierungspunkt gewählt werden, bei dem die Queue sicher leer ist. Dies kann man nur zum Zeitpunkt der Momentaufnahme in Zeile *D6* sicherstellen. Damit ist diese Zeile der Linearisierungspunkt, falls die Dequeue-Operation den Wert $\emptyset$ ausgibt.

Genau dieser Fall ist allerdings problematisch bei der Verifikation der Dequeue-Operation. Eine naive – und falsche – Vorgehensweise wäre, in Zeile *D6* eine Fallunterscheidung durchzuführen, ob die Queue aktuell leer ist und in diesem Fall die abstrakte Operation auszuführen. Diese Vorgehensweise wäre

fehlerhaft, da es möglich ist, dass der aktuelle Schleifendurchlauf in Zeile *D7* abgebrochen wird und im nächsten Schleifendurchlauf ein anderer Linearisierungspunkt gewählt werden muss. Tatsächlich kann man erst in Zeile *D7* entscheiden, ob *D6* der Linearisierungspunkt war oder nicht. Die Abbildung 9.6 stellt diesen Sachverhalt dar: Für den abstrakten Ablauf muss in Zeile *D6* entschieden werden, ob der Linearisierungspunkt erreicht ist (Ablauf 3) oder nicht (Ablauf 4). Beim konkreten Ablauf wird diese Entscheidung einen Schritt später getroffen (Ablauf 1 und 2).

Da die Entscheidung, ob der Linearisierungspunkt erreicht ist, vom zukünftigen Ablauf abhängt, haben klassische Verifikationstechniken Probleme, die jeden einzelnen Schritt eines konkreten Algorithmus separat verfeinern. Sie benötigen eine Rückwärtssimulation der Schritte oder eine sogenannte *Prophecy*-Variable zur Verifikation. Die Lösung mit dem in Kapitel 7 vorgestellten Verfeinerungstheorem ist dagegen deutlich einfacher, da dieses auf Verfeinerung von Intervallen, also vollständigen Abläufen basiert.

9.4 Korrektheit des „Michael-Scott Queue“- Algorithmus

In diesem Abschnitt wird der Korrektheitsbeweis für die „Michael-Scott Queue“ beschrieben. Dazu wird das in Kapitel 7.3 beschriebene Verfeinerungstheorem verwendet. In den folgenden drei Abschnitten werden dann die dazu benötigte Abstraktionsfunktion, Invariante und die Rely-/Guarantee-Bedingungen behandelt. Damit kann der eigentliche Verfeinerungsbeweis für die „Michael-Scott Queue“ geführt werden. Dieser wird in Abschnitt 9.4.4 beschrieben.

9.4.1 Abstraktionsfunktion

Ähnlich wie bei der Stack Fallstudie wird die generische Zustandsvariable *CS* durch einen Produktdatentyp instanziert. Dieser besteht aus den Variablen *Hp*, *Head* und *Tail* und einer Abbildung von *Nat* auf den jeweils entsprechenden Datentyp für jede lokale Variable. Damit kann auf die lokalen Variablen eines Prozesses i mit $ENew(i), ETl(i), ENxt(i)$ und $ESucc(i)$ für einen Enqueue-Prozess, bzw. mit $DHd(i), DNxt(i)$ und $DSucc(i)$ für einen Dequeue-Prozess zugegriffen werden.

Zur Definition der Abstraktionsfunktion wird wie bei der „Treiber-Stack“-Fallstudie zunächst rekursiv eine Abstraktionsrelation *Abs* zwischen der abstrakten und der konkreten Queue definiert. Diese wird zur Definition der Abstraktionsfunktion und des *valid*-Prädikats, mit dem die erlaubten Heap-Konfigurationen definiert werden, benutzt.

Zunächst wird *Abs* für die leere Queue definiert:

$$
\begin{aligned}
& Abs(emptyq, Hp, Head, Tail) \\
\leftrightarrow \quad & Head \neq \textsc{Null} \wedge Head \in Hp \wedge Hp[Head].\mathrm{nxt} = \textsc{Null} \\
& \wedge\ Tail \neq \textsc{Null} \wedge Tail \in Hp \\
& \wedge\ (Tail = Head \vee Tail \neq Head \wedge Hp[Tail].\mathrm{nxt} = Head)
\end{aligned}
$$

a) Korrekter Tail-Zeiger b) Zurückhängender Tail-Zeiger

Abbildung 9.7: Representationen der einelementigen Queue

Die Definition der *Abs*-Relation besteht aus drei Teilen: Der *Head*-Zeiger muss auf eine im Heap allozierte Zelle zeigen, die keinen Nachfolger hat (also NULL als .nxt-Zeiger). Ebenso muss der *Tail*-Zeiger auf eine im Heap allozierte Zelle verweisen. Zusätzlich muss entweder *Tail* selbst oder der .nxt-Zeiger der von *Tail* referenzierten Zelle gleich dem *Head*-Zeiger sein. Der zweite Fall spiegelt dabei einen zurückhängenden *Tail*-Zeiger wieder (wie in Abbildung 9.1d dargestellt).

Wegen des zurückhängenden *Tail*-Zeigers muss noch die Abstraktionsrelation für die einelementige Queue spezifiziert werden, bevor der rekursive Fall spezifiziert werden kann. Die verschiedenen Konfigurationen einer einelementigen Queue sind in Abbildung 9.7 graphisch dargestellt. Die *Abs*-Relation für eine Queue mit genau einem Element v wird wie folgt definiert:

$$\begin{aligned} & \mathit{Abs}(v + \mathit{emptyq}, \mathit{Head}, \mathit{Tail}, \mathit{Hp}) \\ \leftrightarrow \quad & \mathit{Head} \neq \text{NULL} \wedge \mathit{Head} \in \mathit{Hp} \wedge \mathit{Hp}[\mathit{Hp}[\mathit{Head}].\text{nxt}].\text{val} = v \\ & \wedge\ (\mathit{Tail} \neq \mathit{Head} \wedge \mathit{Tail} = \mathit{Hp}[\mathit{Head}].\text{nxt} \vee \mathit{Tail} = \mathit{Head}) \\ & \wedge\ \mathit{Abs}(\mathit{emptyq}, \mathit{Hp}[\mathit{Head}].\text{nxt}, \mathit{Tail}, \mathit{Hp}) \end{aligned}$$

Die Bedingungen für den *Head*-Zeiger sind ähnlich wie bei der leeren Queue. Die von *Head* referenzierte Zelle muss nun aber nicht den NULL-Zeiger enthalten, sondern auf eine zweite Zelle mit dem Wert v zeigen. Für *Tail* muss wieder unterschieden werden, ob der *Tail*-Zeiger zurückhängt. Wenn das nicht der Fall ist, muss er auf die zweite Zelle von *Head* aus zeigen (dabei ist *Head* ungleich *Tail*). Falls der *Tail*-Zeiger zurückhängt, muss er gleich dem *Head*-Zeiger sein. Zusätzlich müssen rekursiv die Bedingungen für die leere Queue gelten, wenn der *Head*-Zeiger eine Zelle weiter geschoben wird.

Schließlich kann der rekursive Fall mit einem ersten Element v und einer restlichen Queue $Q \neq \mathit{emptyq}$ definiert werden:

$$\begin{aligned} & \mathit{Abs}(v + Q, \mathit{Head}, \mathit{Tail}, \mathit{Hp}) \\ \leftrightarrow \quad & \mathit{Head} \neq \text{NULL} \wedge \mathit{Head} \in \mathit{Hp} \wedge \mathit{Hp}[\mathit{Hp}[\mathit{Head}].\text{nxt}].\text{val} = v \\ & \wedge\ \mathit{Tail} \neq \mathit{Head} \wedge (\quad \mathit{Hp}[\mathit{Tail}].\text{nxt} = \text{NULL} \\ & \qquad\qquad \vee \quad \mathit{Hp}[\mathit{Tail}].\text{nxt} \neq \text{NULL} \wedge \mathit{Hp}[\mathit{Tail}].\text{nxt} \in \mathit{Hp} \\ & \qquad\qquad\ \wedge\ \mathit{Hp}[\mathit{Hp}[\mathit{Tail}].\text{nxt}].\text{nxt} = \text{NULL}) \\ & \wedge\ \mathit{Abs}(Q, \mathit{Hp}[\mathit{Head}].\text{nxt}, \mathit{Tail}, \mathit{Hp})) \end{aligned}$$

Da in diesem Fall das erste Element der Queue ebenfalls v ist, können die Bedingungen von *Head* genau wie im Fall der einelementigen Queue definiert

werden. Wie bei den beiden vorhergehenden Definitionen wird für den *Tail*-Zeiger unterschieden, ob er zurückhängt oder nicht. Falls er nicht zurückhängt, muss die referenzierte Zelle NULL als .nxt-Zeiger enthalten. Andernfalls muss es eine Nachfolgerzelle im Heap geben, die NULL als .nxt-Zeiger enthält. Schließlich müssen wieder rekursiv die Bedingungen für die Queue Q gelten, wenn der *Head*-Zeiger eine Zelle weiter geschoben wird. Die Abstraktionsrelation *Abs* ist eindeutig definiert, d.h. es gilt:

$$Abs(Q_1, Head, Tail, Hp) \wedge Abs(Q_2, Head, Tail, Hp) \rightarrow Q_1 = Q_2$$

Wie bei der Stack-Fallstudie wird ein Prädikat *valid* benötigt, um die gültigen Repräsentationen der Queue zu charakterisieren. Dies kann wieder über die Abstraktionsrelation definiert werden.

$$valid(Head, Tail, Hp) \quad :\leftrightarrow \quad \exists\ Queue.\ Abs(Queue, Head, Tail, Hp)$$

Eine Konfiguration der konkreten Datenstruktur *Head*, *Tail* und *Hp* ist gültig, wenn es eine abstrakte *Queue* gibt, die bezüglich *Abs* mit der konkreten Datenstruktur in Relation steht.

Die partielle Abstraktionsfunktion *absf* wird mit Hilfe von *Abs* und dem Prädikat *valid* definiert:

$$valid(Head, Tail, Hp) \ \rightarrow \ Abs(absf(Head, Tail, Hp), Head, Tail, Hp) \qquad (9.1)$$

Falls die konkrete Repräsentation der Queue gültig ist, so ist das Ergebnis der Abstraktionsfunktion *absf* ein zu *Abs* passender Wert. Damit die Funktion *absf* in jedem erreichbaren Zustand definiert ist, wird die Formel $valid(Head, Tail, Hp)$ in die Invariante integriert (siehe nächster Abschnitt).

9.4.2 Invariante

Die Invariante besteht aus drei Teilen:

$$I(CS) \quad :\leftrightarrow \quad valid(CS) \wedge okrep(CS) \wedge disj(CS)$$

Die Repräsentation der Queue muss auf dem Heap gültig sein. Das Prädikat *okrep* beschreibt für jeden Prozess m die zulässige Zeigerstruktur. Dabei kann dieses Prädikat in einen ersten Teil Inv_{enq} für die Enqueue-Operation und einen zweiten Teil Inv_{deq} für die Dequeue-Operation aufgeteilt werden:

$$okrep(CS) \quad :\leftrightarrow \quad \forall\ m.(Inv_{enq}(m, CS) \wedge Inv_{deq}(m, CS))$$

Für die meisten lokalen Zeiger gilt, dass sie auf im Heap allozierte Zellen zeigen oder der Null-Zeiger sind. Um dies für eine Variable P sicherzustellen, wird das Prädikat *InHp* benutzt:

$$InHp(P, Hp) \quad :\leftrightarrow \quad P \neq \text{NULL} \rightarrow P \in Hp$$

Das Prädikat Inv_{enq} kann wie folgt definiert werden:

$$
\begin{aligned}
&Inv_{enq}(m, CS) \quad :\leftrightarrow \\
&\quad InHp(ENew(m), Hp) \wedge InHp(ETl(m), Hp) \wedge InHp(ENxt(m), Hp) \\
&\wedge\ ETl(m) \neq \textsc{Null} \rightarrow InHp(Hp[ETl(m)].\mathrm{nxt}, Hp) \\
&\wedge\ (\quad \neg\ ESucc(m) \wedge ENew(m) \neq \textsc{Null} \\
&\quad\ \rightarrow\ \neg\ reachable(Head, ENew(m), Hp) \wedge Hp[ENew(m)].\mathrm{nxt} = \textsc{Null}) \\
&\wedge\ (ETl(m) \neq \textsc{Null} \wedge Hp[ETl(m)].\mathrm{nxt} = \textsc{Null} \rightarrow ETl(m) = Tail)
\end{aligned}
$$

In der ersten Zeile der Konjunktion wird das Prädikat *InHp* benutzt, um sicherzustellen, dass die lokalen Zeiger auf gültige Zellen verweisen. Diese Bedingung muss auch für $Hp[ETl(m)].\mathrm{nxt}$ gelten, falls $ETl(m)$ nicht der Null-Zeiger ist (zweite Zeile). In der dritten Zeile wird sichergestellt, dass die neu allozierten Zellen nicht Teil der Queue sind. Das Prädikat *reachable* definiert die Erreichbarkeit der Zelle B von der Zelle A aus. Die Definition findet sich in Kapitel 7.6 (siehe Formel (7.24), Seite 132). In der letzten Zeile wird der Zusammenhang zwischen dem globalen *Tail*-Zeiger und dessen lokaler Momentaufnahme beschrieben. Falls die Momentaufnahme auf das Ende der Queue verweist, muss sie identisch mit dem *Tail*-Zeiger sein.

Die entsprechenden Bedingungen für eine Dequeue-Operation werden wie folgt definiert:

$$
\begin{aligned}
&Inv_{deq}(m, CS) \quad :\leftrightarrow \\
&\quad InHp(DHd(m), Hp) \wedge InHp(DNxt(m), Hp) \\
&\wedge\ DHd(m) \neq \textsc{Null} \rightarrow InHp(Hp[DHd(m)].\mathrm{nxt}, Hp) \\
&\wedge\ (DHd(m) \neq \textsc{Null} \wedge Hp[DHd(m)].\mathrm{nxt} = \textsc{Null} \rightarrow DHd(m) = Head)
\end{aligned}
$$

Die ersten beiden Zeilen der Konjunktion stellen sicher, dass die von lokalen Variablen referenzierten Zellen alloziert sind. Diese Bedingungen sind analog zu den ersten beiden Zeilen von Inv_{deq} definiert. In der letzten Zeile wird der Zusammenhang zwischen *Head* und dessen lokaler Momentaufnahme spezifiziert. Sie entspricht der letzten Zeile von Inv_{deq}.

In *disj* wird für alle m spezifiziert, dass alle neu allozierten Zellen *ENew* disjunkt sind:

$$disj(CS) \ :\leftrightarrow\ \forall\, m, n.\ (m \neq n \wedge ENew(m) \neq \textsc{Null} \rightarrow ENew(m) \neq ENew(n))$$

Eine analoge Bedingung gibt es auch für die *UNew*-Zellen in der Stack Fallstudie (siehe das *disj*-Prädikat in Kapitel 7.6.2).

9.4.3 Rely- und Guarantee-Bedingungen

Wie bei der Stack-Fallstudie kann die lokale Umgebungsbedingung R_m durch drei Teile definiert werden:

$$
\begin{aligned}
R_m(CS', CS'') \ :\leftrightarrow \quad & (I(CS') \rightarrow I(CS'')) \\
& \wedge\ Enqlocal_m(CS', CS'') \wedge Deqlocal_m(CS', CS'')
\end{aligned}
$$

Der erste Teil stellt sicher, dass die Invariante erhalten bleibt. Der zweite Teil $Enqlocal_m$ beinhaltet alle Annahmen, die eine Enqueue-Operation benötigt, während $Deqlocal_m$ alle Annahmen für eine Dequeue-Operation enthält.

Die Umgebungsannahme der Enqueue Operation wird wie folgt definiert:

$$
\begin{aligned}
&Enqlocal_m(CS', CS'') \ :\leftrightarrow \\
&\qquad\qquad Unchg_{env}(ENew(m), ETl(m), ENxt(m), ESucc(m)) \\
&\qquad \wedge \ (\quad ENew'(m) \neq \textsc{Null} \wedge \neg\, ESucc'(m) \\
&\qquad\quad \rightarrow \ Hp''[ENew''(m)] = Hp'[ENew'(m)]) \\
&\qquad \wedge \ (\quad ETl'(m) \neq \textsc{Null} \wedge Hp'[ETl'(m)].\text{nxt} \neq \textsc{Null} \\
&\qquad\quad \rightarrow \ Hp''[ETl''(m)].\text{nxt} = Hp'[ETl'(m)].\text{nxt})
\end{aligned}
$$

Die erste Zeile der Konjunktion beschreibt, dass lokale Variablen nicht durch die Umgebung geändert werden (die Definition von $Unchg_{env}$ ist in Kapitel 4.6.3 beschrieben). Im zweiten Teil wird sichergestellt, dass die neuallozierte Zelle nicht von der Umgebung verändert wird. Der dritte Teil besagt, dass diese Referenz nicht geändert wird, wenn der .nxt-Zeiger der Momentaufnahme des *Tail*-Zeigers nicht Null ist. Der erste und zweite Teil findet sich in analoger Form auch in der Umgebungsannahme für die Push-Operation wieder. Lediglich die dritte Bedingung wird zusätzlich benötigt, damit sichergestellt ist, dass die zweite CAS-Operation in Zeile *E12* bzw. Zeile *E16* korrekt ausgeführt wird.

Für die Dequeue-Operation wird die folgende Umgebungsannahme benötigt:

$$
\begin{aligned}
&Deqlocal_m(CS', CS'') \ :\leftrightarrow \\
&\qquad\qquad Unchg_{env}(DSucc(m), DHd(m), DNxt(m)) \\
&\qquad \wedge \ (\quad DHd'(m) \neq \textsc{Null} \wedge Hp'[DHd'(m)].\text{nxt} \neq \textsc{Null} \\
&\qquad\quad \rightarrow \quad Hp''[DHd''(m)].\text{nxt} = Hp'[DHd'(m)].\text{nxt} \\
&\qquad\qquad \wedge \ Hp''[Hp''[DHd''(m)].\text{nxt}].\text{val} = \\
&\qquad\qquad\quad Hp'[Hp'[DHd'(m)].\text{nxt}].\text{val})
\end{aligned}
$$

Hier muss ebenfalls sichergestellt werden, dass die Umgebung die lokalen Variablen nicht verändert. Zusätzlich darf weder der Zeiger auf die zweite Zelle von der Momentaufnahme von *Head* aus noch deren Wert verändert werden, falls die Referenz auf die zweite Zelle nicht Null ist. Eine ähnliche Bedingung findet sich auch in der Umgebungsannahme der Pop-Operation der Stack-Fallstudie; allerdings ist die Formalisierung bei der Queue wegen des Pseudoknotens komplexer.

Die lokalen Guarantee-Bedingungen bestehen analog zur „Treiber-Stack“-Fallstudie aus den lokalen Rely-Bedingungen aller anderen Prozesse:

$$G_i(CS, CS') \ :\leftrightarrow \ \forall\, j.\ j \neq i \rightarrow R_j(CS, CS')$$

D.h. jeder Prozess erfüllt die Rely-Bedingungen der anderen Prozesse.

Zusätzlich wird eine globale Umgebungsannahme benötigt. Auch diese wird wie bei der Stack-Fallstudie definiert:

$$R(CS', CS'') \;:\leftrightarrow\; (I(CS') \rightarrow I(CS'')) \wedge \forall i.R_i(CS', CS'')$$

Als Initialbedingung wird die Invariante angenommen, d.h.

$$Init(CS) \quad :\leftrightarrow \quad I(CS)$$

Alternativ könnte wie bei der Stack-Fallstudie auch die leere Queue initial angenommen werden. Damit wäre auch die Beweisverpflichtung 8. des Verfeinerungstheorems erfüllt. Auf die restlichen Beweise hat diese Änderung keine Auswirkung.

9.4.4 Beweis

Mit der beschriebenen Invariante und den Rely- und Guarantee-Bedingungen können die prädikatenlogischen Beweisverpflichtungen *2.* bis *8.* des Verfeinerungstheorems gezeigt werden. Auch der Beweis, dass die Operationen ihre Rely-Guarantee-Bedingung aufrecht erhalten (Beweisverpflichtung *1.*), lässt sich mit wenigen Interaktionen zeigen. Der Beweis besteht im Wesentlichen aus symbolischer Ausführung und Induktion über die Unless-Formel des Sustain-Operators.

Die eigentlichen Verfeinerungsbeweise (Beweisverpflichtung *9.*) sind etwas aufwändiger. Zum Beweis der Enqueue-Operation werden die im vorhergehenden Abschnitt erläuterten Rely-Bedingungen benötigt. Zum einen ist es wichtig, dass die neu allozierte Zelle nicht von der Umgebung verändert wird, bis sie von der Enqueue-Operation in die Queue integriert wurde. Ohne diese Bedingung könnte ein ungültiger Wert zur Queue hinzugefügt werden. Außerdem darf die .nxt-Referenz der Zelle, auf welche die Momentaufnahme $ETl(m)$ verweist, von der Umgebung nicht verändert werden. Diese wird für den Nachweis einer korrekten Änderung des *Tail*-Zeiger in Zeile *E12* bzw. *E16* benötigt. Insgesamt ähnelt der Verfeinerungsbeweis der Enqueue-Operation den Verfeinerungsbeweisen für die Stack-Operationen. Allerdings verzweigt dieser Beweis stärker und es sind auch mehr Interaktionen notwendig. Dies resultiert aus der höheren Komplexität der Queue-Operation.

Der Verfeinerungsbeweis der Dequeue-Operation ist noch komplexer. Während der Linearisierungspunkt bei der Enqueue-Operation eindeutig ist, gibt es bei der Dequeue-Operation mehrere mögliche Linearisierungspunkte. Wie in Abschnitt 9.3 bereits beschrieben, kommt erschwerend hinzu, dass der Linearisierungspunkt bei einer leeren Queue erst im Nachhinein bestimmt werden kann. Dies stellt für viele Ansätze eine besondere Schwierigkeit dar. Mit der in dieser Arbeit vorgestellten Verfeinerungstechnik lässt sich dieses Problem lösen, indem mehrere mögliche abstrakte Abläufe gleichzeitig berücksichtigt werden. Im Folgenden werden die wesentlichen Beweisschritte für einen Prozess i beschrieben. Die kritischen Schritte, in denen mehrere Linearisierungspunkte berücksichtigt werden müssen, sind die beiden Zeilen *D6* und *D7* der konkreten Dequeue-Operation. Abbildung 9.8 zeigt die Beweisschritte für diesen Abschnitt. Dabei

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\mathbf{D8}, RI, \varphi_2 \vdash \dots}}{\dots \vdash \mathbf{AD8}}\,(6)}{\dots, \varphi_1, DHd(i) = Head \vdash \mathbf{AD3}, \mathbf{AD8}}\,(5) \quad \dfrac{\dfrac{\dfrac{}{\mathbf{D4}, RI, \varphi_3 \vdash \dots}\,(9)}{\dots \vdash \mathbf{AD3}}\,(8)}{\dots, \varphi_1, DHd(i) \neq Head \vdash \mathbf{AD3}, \mathbf{AD8}}\,(7)}{\mathbf{D7}, RI, \varphi_1 \vdash \mathbf{AD3}, \mathbf{AD8}}\,(4)}{\mathbf{D6}, RI, \varphi_0, Hp[DHd(i)].\text{nxt} = \text{Null} \vdash skip; \mathbf{AD3}, \mathbf{AD4}}\,(3)}{\mathbf{D6}, RI, \varphi_0, Hp[DHd(i)].\text{nxt} = \text{Null} \vdash \mathbf{AD3}}\,(2) \quad \dots \neq \text{Null} \vdash \dots}{\mathbf{D6}, RI, \varphi_0 \vdash \mathbf{AD3}}\,(1)$$

$$\begin{aligned}
RI &:\equiv \Box\,(R_i(CS', CS'') \wedge Inv(CS) \wedge Inv(CS')) \\
\varphi_0 &:\equiv \neg\, DSucc(i) \wedge DHd(i) \neq \text{Null} \wedge abs(queue_0, CS) \\
\varphi_1 &:\equiv \neg\, DSucc(i) \wedge DHd(i) \neq \text{Null} \wedge DNxt(i) = \text{Null} \wedge abs(queue_1, CS) \\
\varphi_2 &:\equiv \neg\, DSucc(i) \wedge DHd(i) \neq \text{Null} \wedge DNxt(i) = \text{Null} \wedge abs(queue_2, CS) \\
\varphi_3 &:\equiv \neg\, DSucc(i) \wedge DHd(i) \neq \text{Null} \wedge DNxt(i) = \text{Null} \wedge abs(queue_3, CS)
\end{aligned}$$

Abbildung 9.8: Beweisskizze von Dequeue Empty

steht **Dk** für das verbleibende Programm ab Zeile *Dk* (siehe Abbildung 9.9). So steht zum Beispiel **D6** für die Instruktionen von *D6* bis *D21*, gefolgt von einem Restprogramm von *D4* bis *D23*, welches im Wesentlichen die While-Schleife enthält. Ebenso steht die Abkürzung **ADk** für das verbleibende Restprogramm von *ADeq* ab Zeile *ADk*. Die anderen benutzten Abkürzungen sind in Abbildung 9.8 erläutert. Die Konklusion von (1) beschreibt den Zustand, nachdem die Zeilen *D1* bis *D5* von *CDeq* ausgeführt wurden. Das noch auszuführende konkrete Programm ist **D6**. Prozess i hat einen Zustand erreicht, in dem die Formel φ_0 gilt. Mit der Annahme der Umgebungs- und Invariantenbedingung (abgekürzt durch *RI*) muss gezeigt werden, dass es einen bezüglich *absf* passende abstrakten Ablauf von **AD3** gibt.

Im Beweisschritt (1) wird zunächst eine Fallunterscheidung durchgeführt, ob die von $DHd(i)$ referenzierte Zelle den Null-Zeiger als .nxt-Referenz hat. Der unkritische Fall liegt vor, wenn diese Referenz nicht Null ist. In dieser Situation liegt der Linearisierungspunkt entweder bei der CAS-Operation in Zeile *D13* oder die CAS-Operation schlägt fehl und die While-Schleife wird erneut durchlaufen. Beide Fälle können mit den bisher benutzten Techniken – symbolische Ausführung und Induktion – bewiesen werden. Der kritische Fall tritt jedoch ein, falls die Referenz Null ist und damit die aktuell repräsentierte Queue leer ist.[1] In diesem Fall ist noch nicht klar, ob der aktuelle Systemschritt der Linearisierungspunkt ist.

Mit der Regel *star* aus Tabelle 2.4 (siehe Seite 28 in Kapitel 2.4) und den Regeln *chp lem* und *chp dis* aus Tabelle 2.5 (siehe Seite 29) kann das abstrakte Programm **AD3** wie folgt umgeschrieben werden:

$$\mathbf{AD3} \quad \leftrightarrow \quad \mathbf{AD4} \vee \mathbf{skip}; \mathbf{AD3} \tag{9.2}$$

[1] Dass die Queue in diesem Fall leer ist, kann mit Inv_{deq} aus der Invariante gezeigt werden.

```
D6      DNxt := Hp[DHd].nxt;
D7      if DHd = Head then {
D8          if DNxt = NULL then {
D9              Lo := ∅;
D10             DSucc := true;
D11         } else {
D12             Lo := Hp[DNxt].val;
D13             CAS(DHd, DNxt; Head, DSucc);
D14             if DSucc then {
D15                 DTl := Tail;
D16                 if DTl = DHd then {
D17                     CAS(DTl, DNxt; Tail);
D18                 }
D19             }
D20         }
D21     }
D4      while ¬ DSucc do {
D5-D21  ...
D22     }
D23     O := Lo;
```

Abbildung 9.9: **D6** – der verbleibende Dequeue-Algorithmus ab Zeile *D6*

Mit der Disjunktion auf der rechten Seite wird eine Fallunterscheidung durchgeführt. Im Fall **AD4** wird im nächsten Schritt die abstrakte Operation ausgeführt; die Operation befindet sich demnach am Linearisierungspunkt. Im anderen Fall wird ein weiterer **skip**-Schritt ausgeführt, d.h. der aktuelle Schritt ist nicht der Linearisierungspunkt. In Beweisschritt (2) wird die Äquivalenz (9.2) benutzt, um beide möglichen Fälle zu berücksichtigen. Die eigentliche Entscheidung, ob der aktuelle Schritt der Linearisierungspunkt ist oder nicht, wird erst später gefällt. Allerdings müssen bis zu dem Zustand, an dem die Entscheidung getroffen wird, beide Formeln **AD4** und **skip**; **AD3** vom konkreten Programmablauf erfüllt werden. Dieses Vorgehen ist möglich, da die beiden Formeln (disjunkt verknüpft) im Sukzedent stehen, d.h. es muss nur eine der beiden Formeln bewiesen werden.

In Beweisschritt (3) wird zunächst ein symbolischer Ausführungsschritt von **D6** nach **D7** vollzogen. Dabei wird der momentane Zustand vom .nxt-Zeiger der ersten Zelle kopiert (der den Wert NULL hat). Die konkrete Queue wird in diesem Schritt nicht geändert. Im gleichen Schritt werden auch die beiden möglichen abstrakten Restprogramme **skip**; **AD3** und **AD4** ausgeführt. Im ersten Fall wird das **skip** ausgeführt, d.h. der abstrakte Zustand wird nicht verändert. Das Restprogramm ist in diesem Fall **AD3**. Im anderen Fall wird die eigentliche abstrakte Operation durchgeführt. Das Restprogramm nach der abstrakten Operation ist dann **AD8**. Da die Queue leer ist, wird auch in diesem Fall der abstrakte Zustand nicht verändert. Damit machen die beiden abstrakten Restprogramme einen zum konkreten Systemschritt (bzgl. der Abstraktionsfunktion) „passenden“ Schritt.

Nun kann entschieden werden, ob der letzte Systemschritt der Linearisierungspunkt war. Dazu wird in Beweisschritt (4) eine Fallunterscheidung durchgeführt, ob die in Zeile *D5* gemachte Momentaufnahme $DHd(i)$ noch identisch mit *Head* ist. Falls dies der Fall ist, war *D6* der Linearisierungspunkt, d.h. als abstraktes Restprogramm wird **AD8** gewählt (Beweisschritt (5) eliminiert **AD3** durch Abschwächung). Nun kann der Beweis mit symbolischer Ausführung fortgesetzt werden (Beweisschritt (6)). Falls jedoch nach Beweisschritt (4) $DHd(i)$ nicht identisch mit *Head* ist, wurde der Linearisierungspunkt noch nicht erreicht. In diesem Fall muss das abstrakte Restprogramm **AD3** gewählt werden (Beweisschritt (7)). Da die if-Anweisung in Zeile *D7* nicht erfüllt ist, bleibt nach einem symbolischen Ausführungsschritt als konkretes Restprogramm *D4* (Beweisschritt (8)). Dieser Fall kann durch Induktion geschlossen werden (Beweisschritt (9)).

9.5 Lebendigkeit des „Michael-Scott“-Queue- Algorithmus

Um das in Kapitel 8 vorgestellte Theorem zu benutzen und die Lebendigkeit für die „Michael-Scott“-Queue zu zeigen, muss zusätzlich zu der in Abschnitt 9.4 gegebenen Rely-Bedingung und der Invariante ein passendes Prädikat *Uchg* spezifiziert werden. Im nächsten Abschnitt wird das *Uchg* Prädikat für die „Michael-Scott“-Queue erläutert. Der Beweis der Prämissen des *Lock-Freedom*-Theorems wird dann in Abschnitt 9.5.2 beschrieben.

9.5.1 Das *Uchg*-Prädikat

Mit dem *Uchg*-Prädikat wird ausgedrückt, welche Änderungen im Heap eine erfolgreiche Enqueue- bzw. Dequeue-Operation verhindern. Da die Enqueue- und Dequeue-Operation unterschiedliche Zeiger zur Detektion von Veränderungen der Queue benutzen, müssen Änderungen des *Head*- als auch des *Tail*-Zeigers berücksichtigt werden.

Die Dequeue-Operation benutzt in der entscheidenden CAS-Operation den *Head*-Zeiger zum Detektieren von relevanten Änderungen. Damit eine solche Operation erfolgreich ist, darf sich der *Head*-Zeiger nicht ändern:

$$Id_H \quad :\equiv \quad Head'' = Head'$$

Für die Enqueue-Operation reicht es nicht aus, dass der *Tail*-Zeiger nicht verändert wird, um eine erfolgreiche Operation zu garantieren. Zum Beispiel würde ein Prozess m nie eine erfolgreiche Operation durchführen, wenn ein anderer Prozess mit einer Enqueue-Operation immer neue Zellen an die von *Tail* referenzierte Zelle hinzufügen, sobald der *Tail*-Zeiger nicht zurückhängt. Dabei wird der .nxt-Zeiger der von *Tail* referenzierten Zelle verändert, so dass die CAS-Operation des Prozesses m in Zeile *E10* fehlschlägt.

Um festzustellen, dass kein anderer Prozess eine Zelle zur Queue hinzugefügt hat, muss daher sowohl der *Tail*-Zeiger selbst als auch der .nxt-Zeiger der von *Tail* referenzierten Zelle betrachtet werden. Formal wird dabei unterschieden,

ob *Tail* zurückhängt (d.h. ob $Hp[Tail].\text{nxt} \neq \text{NULL}$). Falls der *Tail*-Zeiger nicht zurückhängt, darf weder *Tail* noch der .nxt-Zeiger der von *Tail* referenzierten Zelle verändert werden:

$$Id_T \quad :\equiv \quad Tail'' = Tail' \wedge Hp''[Tail''].\text{nxt} = Hp'[Tail'].\text{nxt}$$

Bei einem zurückhängenden *Tail*-Zeiger, muss das folgende Verhalten angenommen werden:

$$Id_{LT} \quad :\equiv \quad Id_T \vee Tail'' = Hp'[Tail'].\text{nxt} \wedge Hp''[Tail''].\text{nxt} = \text{NULL}$$

Entweder die Umgebung lässt sowohl *Tail* als auch die .nxt-Referenz unverändert, oder *Tail* wird um eine Zelle verschoben, so dass sie auf die letzte Zelle zeigt.

Damit kann das *Uchg*-Prädikat definiert werden:

$$\begin{aligned} Uchg(CS', CS'') \quad :\leftrightarrow \quad & Id_H \\ & \wedge\ (Hp'[Tail'].\text{nxt} = \text{NULL} \rightarrow Id_T) \\ & \wedge\ (Hp'[Tail'].\text{nxt} \neq \text{NULL} \rightarrow Id_{LT}) \end{aligned}$$

9.5.2 Beweis

Die Initialbedingung, die Invariante sowie die Rely- und Guarantee-Bedingung können direkt aus dem Korrektheitsbeweis der Queue übernommen werden. Daher müssen die Beweisverpflichtungen 1. bis 8. des *Lock-Freedom*-Theorems nicht erneut bewiesen werden. Das im vorhergehenden Abschnitt beschriebene *Uchg*-Prädikat ist offenbar reflexiv und transitiv (entspricht Beweisverpflichtungen 9. und 10.).

Wie beim Lebendigkeitsbeweis der „Treiber-Stack“-Fallstudie wird der Beweis von Prämisse 11. in vier Teilbeweise aufgespalten. Für die Enqueue-Operation sind die folgenden beiden Sequenzen zu beweisen:

$$\begin{aligned} & [CEnq(i, In; CS, Out)]_{In,CS,Out}, \\ & \Box\ (I(CS) \wedge I(CS') \wedge R_i(CS', CS'')) \\ \vdash\ & \Box\ (\Box\ Uchg(CS', CS'') \rightarrow \Diamond\ \mathbf{last}) \end{aligned} \tag{9.3}$$

$$\begin{aligned} & [CEnq(i, In; CS, Out)]_{In,CS,Out}, \\ & \Box\ (I(CS) \wedge I(CS') \wedge R_i(CS', CS'')) \\ \vdash\ & \Box\ (\neg\ Uchg(CS, CS') \rightarrow \Diamond\ \mathbf{last}) \end{aligned} \tag{9.4}$$

Der zweite Beweis ist relativ einfach. Die Formel kann durch Induktion über die Always-Formel im Sukzedenten bewiesen werden. Im Wesentlichen besteht der Beweis aus schrittweisem Ausführen von *CEnq*. Die Enqueue-Operation ändert die Datenstruktur nur an einer einzige Stelle, nämlich wenn die CAS-Operation in der Zeile *E10* erfolgreich ausgeführt wird. In diesem Fall muss gezeigt werden, dass die Operation terminiert. Da die Enqueue-Operation in

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{E8}\ldots,\varphi_1 \vdash \ldots}{\cdots \neq \textsc{Null},\ \mathit{Tail} = \mathit{ETl}(m) \vdash \ldots}\ (4) \quad \cdots = \textsc{Null} \vdash \ldots}{\mathbf{E7},\ldots,\varphi_0 \vdash \ldots}\ (3)}{\ldots\, \mathit{Hp}[\mathit{Tail}].\mathrm{nxt} \neq \textsc{Null} \vdash \ldots}\ (2) \quad \cdots = \textsc{Null} \vdash \ldots}{\mathbf{E6},\ \mathit{VRU} \vdash \Diamond\ \mathbf{last}}\ (1)$$

$$\begin{aligned}
\mathit{VRU} &:\equiv \Box\,(\quad \mathit{valid}(\mathit{Head}, \mathit{Tail}, \mathit{Hp}) \\
&\qquad \wedge\ \mathit{Enqlocal}_m(\mathit{CS}', \mathit{CS}') \wedge \mathit{Uchg}(\mathit{CS}', \mathit{CS}'')) \\
\varphi_0 &:\equiv \mathit{ETl}(m) \neq \textsc{Null} \\
\varphi_1 &:\equiv \quad \mathit{ETl}(m) \neq \textsc{Null} \wedge \mathit{ENxt}(m) \neq \textsc{Null} \\
&\qquad \wedge\ \mathit{Hp}[\mathit{ETl}(m)].\mathrm{nxt} = \mathit{ENxt}(m)
\end{aligned}$$

Abbildung 9.10: Beweisskizze zu Formel (9.5)

diesem Fall immer nach zwei weiteren Schritten terminiert, ist dafür keine neue Induktion notwendig.

Der erste Beweis ist etwas komplexer. Die Sequenz wird ebenfalls durch Induktion über den führenden Always-Operator im Sukzedent bewiesen. Danach muss die Operation ausgeführt werden bis sie entweder terminiert oder durch Induktion geschlossen werden kann. Dabei entsteht jedoch bei jedem Schritt ein weiteres Beweisziel, in dem gezeigt werden muss, dass die Operation terminiert, wenn ab dem aktuellen Zustand $\Box\ \mathit{Uchg}(\mathit{CS}', \mathit{CS}'')$ gilt, d.h. wenn die Umgebung ab diesem Zustand die Queue nicht mehr verändert. Diese Seitenbeweise können gezeigt werden, indem die Enqueue-Operation solange ausgeführt wird, bis das Programm entweder direkt terminiert oder sich wieder am Schleifenanfang in Zeile *E5* befindet. Dieser Fall kann dann durch das folgende Lemma bewiesen werden:

$$\begin{aligned}
&\mathbf{E5},\ \Box\ I(\mathit{CS}) \wedge I(\mathit{CS}') \wedge R_i(\mathit{CS}', \mathit{CS}'') \\
&\vdash \Box\ \mathit{Uchg}(\mathit{CS}', \mathit{CS}'') \rightarrow \Diamond\ \mathbf{last}
\end{aligned} \tag{9.5}$$

Eine Enqueue-Operation am Anfang der Schleife (durch **E5** dargestellt) terminiert, falls die Umgebung die Datenstruktur nicht mehr ändert. Für den Beweis des Lemmas ist keine Induktion notwendig, da die Enqueue-Operation immer nach endlich vielen Schritten terminiert, falls $\mathit{Uchg}(\mathit{CS}', \mathit{CS}'')$ gilt. Im ersten Schritt wird ausgewertet, ob die Schleife betreten wird. Falls *ESucc* nicht gilt, terminiert die Operation im übernächsten Schritt. Andernfalls befindet sich die Operation in Zeile *E6*. Diese Situation ist in der Conclusio der Beweisskizze in Abbildung 9.10 dargestellt. Das noch auszuführende Programm ist **E6**. Die Formel *VRU* (siehe Abbildung 9.10) kann aus der Invariante, der lokalen Rely-Bedingung und der Formel $\Box\ \mathit{Uchg}(\mathit{CS}', \mathit{CS}'')$ hergeleitet werden. Außer *VRU* werden keine weiteren Annahmen für den restlichen Beweis benötigt.

Zunächst wird in Schritt (1) eine Fallunterscheidung durchgeführt, ob der *Tail* zurückhängt. Falls *Tail* nicht zurückhängt (rechte Prämisse von Beweis-

schritt (1)), wird die Umgebung weder *Tail* noch die .nxt-Referenz der letzten Zelle ändern (dies wird durch $\Box\ Uchg(CS', CS'')$ garantiert). Daher terminiert die Enqueue-Operation nach dem aktuellen Schleifendurchlauf, d.h. $\Diamond$ **last** kann durch mehrfaches symbolisches Ausführen gezeigt werden. Falls *Tail* zurückhängt (die linke Prämisse von Beweisschritt (1)), wird zunächst ein weiterer symbolischer Ausführungsschritt durchgeführt (Beweisschritt (2)). Der Programmschritt macht eine Momentaufnahme von *Tail* und das danach verbleibende Programm ist **E7**.

Im Beweisschritt (3) wird überprüft, ob inzwischen ein Prozess in der Umgebung *Tail* verschoben hat, so dass er nicht mehr zurückhängt. In der rechten Prämisse von (3) wurde *Tail* verschoben. In diesem Fall wird die aktuelle Schleifeniteration in Zeile *E8* abgebrochen. Die nächste Iteration kann nach einem Schritt durch die Beweislemma Regel (siehe Kapitel 2.5.2) mit der rechten Prämisse von Beweisschritt (1) geschlossen werden. Falls *Tail* nach dem Beweisschritt (3) immer noch zurückhängt (linke Prämisse), kann der Beweis analog durch symbolische Ausführung und Fallunterscheidung weiter geführt werden, bis die Enqueue-Operation in Zeile *E12* schließlich den *Tail* selbst verschiebt. Auch dieser Fall kann durch die Beweislemma Regel und der rechten Prämisse von Beweisschritt (1) geschlossen werden.

Für die Dequeue-Operation müssen analog zu (9.3) und (9.4) zwei entsprechende Eigenschaften bewiesen werden. Diese Beweise sind einfacher, da nur der *Head*-Zeiger relevant ist, für den keine Fallunterscheidungen durchgeführt werden müssen.

9.6 Fazit

Anhand der Fallstudie der „Michael-Scott“-Queue lassen sich zwei Dinge gut untersuchen. Zum einen kann durch einen Vergleich mit der „Treiber“-Stack-Fallstudie die Wiederverwendbarkeit der Prädikate (z.B. der Invariante) und der Beweise analysiert werden. Durch eine hohe Wiederverwendbarkeit kann der Verifikationsaufwand bei neuen Fallstudien deutlich reduziert werden. In Abschnitt 9.6.1 wird auf diesen Punkt genauer eingegangen.

Der „Michael-Scott“-Queue Algorithmus wurde auch in anderen Arbeiten verifiziert. In Abschnitt 9.6.2 werden die verschiedenen Verifikationsansätze kurz verglichen.

9.6.1 Vergleich zur „Treiber“-Stack-Fallstudie

Wie bei der „Treiber“-Stack-Fallstudie liegt der hauptsächliche Aufwand bei der Verifikation der „Michael-Scott“-Queue in der Spezifikation der Abstraktionsfunktion, der richtigen Invariante und geeigneter Rely-Bedingungen.

Die Abstraktionsfunktionen für Stack und Queue müssen für jede Datenstruktur separat definiert werden. Jedoch können beide nach dem gleichen Schema spezifiziert werden. Die zusätzliche Komplexität der Abstraktionsfunktion der Queue resultiert im Wesentlichen daraus, dass der *Tail*-Zeiger zurückhängen kann. Auch die Invarianten für beide Algorithmen können im Wesentlichen analog spezifiziert werden. Beide sind durch die drei Formeln *valid*, *okrep* (beim

Stack wird dies durch $I_{push} \wedge I_{pop}$ ausgedrückt) und *disj* spezifiziert, die jeweils analog aufgebaut sind. Der wesentliche Unterschied zwischen den beiden Fallstudien sind zusätzliche Bedingungen in *okrep* beim Queue-Algorithmus. Auch diese sind durch den zurückhängenden *Tail*-Zeiger bedingt. Bei der Spezifikation der lokalen Rely-Bedingungen ergibt sich ein ähnliches Bild. Die Unterschiede bei der Queue resultieren aus dem zurückhängenden *Tail*-Zeiger und der Tatsache, das die Queue mit einem Pseudoknoten beginnt.

Bei beiden Fallstudien sind die temporallogischen Beweise die einzigen Beweise von nennenswerter Komplexität. Diese können im Wesentlichen durch die beiden Beweisprinzipien der symbolischen Ausführung und der Induktion bewiesen werden. Die wesentlichen Interaktionen sind dabei Generalisierungen, d.h. das Finden einer Hoare-ähnlichen Invariante. Im Vergleich sind bei den Queue-Beweisen als zusätzliche Komplexität noch Fallunterscheidungen notwendig, die wegen eines zurückhängenden *Tail*-Zeigers oder der leeren Queue getroffen werden müssen. Einen weiteren wesentlichen Unterschied bilden die zusätzlichen Beweisschritte, die wegen des nicht eindeutigen Linearisierungspunktes der Dequeue-Operation durchgeführt werden müssen.

Damit lassen sich die wesentlichen Unterschiede bei der Verifikation der beiden Algorithmen auf konzeptionelle Unterschiede bei beiden Algorithmen zurückführen. Dies lässt erwarten, dass viele Aspekte auch bei der Verifikation von anderen lock-freien Algorithmen wiederverwendbar sind, womit der Verifikationsaufwand deutlich reduziert werden kann.

9.6.2 Andere Arbeiten zur „Michael-Scott“-Queue

In den Kapiteln 7.7 und 8.6 wurde bereits der in dieser Arbeit vorgestellte modulare Ansatz zur Verifikation von lock-freien Algorithmen mit verwandten Arbeiten verglichen. Der „Michael-Scott“-Queue Algorithmus gehört jedoch zu einer komplexeren Klasse von lock-freien Algorithmen als der „Treiber“-Stack Algorithmus. Wie in Abschnitt 9.3 beschrieben wurde, kann die Position des Linearisierungspunkts in bestimmten Fällen nicht während eines Schritts, sondern erst im Nachhinein ermittelt werden. Verifikationsansätze, die nur die einzelnen Schritte verfeinern, benötigen in solchen Fällen zusätzliche Techniken, da ein abstraktes V-förmiges Diagramm durch ein Y-förmiges Diagramm verfeinert werden muss. Dabei wird der Nichtdeterminismus zurück verschoben.

Der automatenbasierte Ansatz von Doherty et al. [DGLM04] benötigt zur Verifikation der Korrektheit des Queue-Algorithmus einen Zwischenautomaten, der Rückwärtssimulation anwendet. Vafeiadis schlägt in seiner Dissertation [Vaf07] eine „*prophecy*“-Variable vor, um dieses Problem zu lösen. Prophecy-Variablen haben eine ähnliche Mächtigkeit wie Rückwärtssimulation. Auch sie benötigen eine Zwischenebene, die den Algorithmus mit den entsprechenden Variablen ergänzt.

Der in dieser Arbeit vorgestellte Ansatz der Verfeinerung über Intervalle (d.h. mittels „trace inclusion“) ist einfacher. Wie in Abschnitt 9.4.4 beschrieben, erlaubt es dieser Ansatz, mehrere mögliche abstrakte Abläufe gleichzeitig zu betrachten. Damit kann man sich erst einige Schritte später, wenn der Linearisierungspunkt feststeht, für den Ablauf mit dem korrekten Linearisierungs-

punkt entscheiden. Daher werden in diesem Ansatz keine neuen Techniken zum Beweis des Queue Algorithmus benötigt.

Kapitel 10

Zusammenfassung

In der vorliegenden Arbeit wird die Integration einer auf dem Rely-Guarantee-Paradigma basierenden Beweistechnik in die Logik ITL$^+$ beschrieben. Diese Logik besitzt eine natürliche Spezifikationssprache und mit der symbolischen Ausführung eine intuitive Beweismethodik (Kapitel 2). Es konnte gezeigt werden, dass sich der Sustain-Operator, der von vielen modularen Ansätzen zur Vermeidung eines Zirkelschlusses benutzt wird, direkt in der Logik ausdrücken lässt. Damit konnte zunächst eine den bekannten Ansätzen vergleichbare Rely-Guarantee Methodik in die Logik ITL$^+$ eingebettet werden (Kapitel 3 und 4). Ein Alleinstellungsmerkmal dieses Rely-Guarantee-Ansatzes ist es, dass das entsprechende Modularisierungstheorem direkt in der Logik ITL$^+$ formalisiert und mit dem Kalkül bewiesen werden kann. Dadurch ergibt sich ein einheitlicher und flexibler Ansatz zur Verifikation von nebenläufigen Systemen, mit dem sich die Vorteile von ITL$^+$ nutzen lassen.

Das entwickelte Rely-Guarantee-Prinzip lässt sich auch auf synchrone Parallelität übertragen (Kapitel 5). Die praktische Anwendung wurde anhand der PROTOCURE-Fallstudie belegt.

Ein aktuelles und wichtiges Forschungsgebiet ist die Analyse und Verifikation von lock-freien Algorithmen (Kapitel 6). Diese Klasse von parallelen Algorithmen ist äußerst komplex und erfordert ganz neue Verifikationstechniken. In der vorliegenden Arbeit wurden der Rely-Guarantee-Ansatz zur Verifikation solcher Algorithmen erweitert. Dabei wurden Methoden zum Nachweis der Korrektheit und Lebendigkeit von lock-freien Algorithmen entwickelt. Zum Nachweis der Korrektheit wird die Rely-Guarantee-Methode so erweitert, dass Verfeinerung zwischen lock-freien und abstrakten Operationen bewiesen werden kann (Kapitel 7). Zur Verifikation dieser Eigenschaft sind dabei nur Beweise über einzelne Komponenten notwendig. Anhand der „Michael-Scott“-Queue wurde demonstriert, dass im Gegensatz zu klassischen Verfeinerungstechniken und anderen Arbeiten zu lock-freien Algorithmen kein Zwischenschritt mit Rückwärtssimulation notwendig ist. Statt dessen kann die Verfeinerung mit dem hier entwickelten Verfahren bewiesen werden, indem mehrere abstrakte Abläufe gleichzeitig betrachtet werden.

Um die Lebendigkeit von lock-freien Algorithmen zu verifizieren, wurde ebenfalls eine modulare Technik zum Nachweis der so genannten Lock-Freedom-

Eigenschaft entwickelt. Dabei ist keine explizite Konstruktion einer wohlfundierten Ordnung notwendig (Kapitel 8). Die Anwendung dieser Technik wurde anhand des „Treiber“-Stack- und des „Michael-Scott“-Queue-Algorithmus demonstriert (Kapitel 9).

Die Ergebnisse dieser Arbeit bilden eine wichtige Grundlage für das gerade anlaufende DFG-Forschungsprojekt „VeriCAS“. Mit diesem Projekt soll die Verifikation von lock-freien Algorithmen gefördert und verbessert werden. Als Zielsetzung sollen zum einen die hier entwickelten Techniken mit automatischen Techniken, wie z. B. Shape-Analysis, kombiniert werden. Eine solche Kombination verspricht eine deutliche Reduzierung des Beweisaufwandes der Prämissen des Modularisierungstheorems. Als zusätzliche Zielvorgabe sollen mit diesen neu entwickelten Methoden komplexere lock-freie Algorithmen verifiziert werden. Der in dieser Arbeit dargestellte flexible Rely-Guarantee-Ansatz bietet dafür zusammen mit dem intuitiven ITL$^+$-Beweiskalkül eine gute Grundlage.

Literaturverzeichnis

[AAD+93] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[AC05] J.-R. Abrial and D. Cansell. Formal Construction of a Non-blocking Concurrent Queue Algorithm (a Case Study in Atomicity). *Journal of Universal Computer Science*, 11(5):744–770, 2005.

[AFV01] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2001.

[AL89] M. Abadi and L. Lamport. Composing Specifications. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, volume 430, pages 1–41, Berlin, Germany, 1989. Springer-Verlag.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, pages 507 – 534, 3 Nov 1995.

[ARR+07] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer, 2007.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. Technical report, Cornell University, Ithaca, NY, USA, 1985.

[AS87] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3), 1987.

[Ash75] E. A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135, 1975.

[Bal05] M. Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction*. PhD thesis, University of Augsburg, Augsburg, Germany, 2005.

[BBC+00] N. Bjørner, A. Brown, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A step tutorial. In *Formal Methods in System Design*, pages 227–270, 2000.

[BBN+10] S. Bäumler, M. Balser, F. Nafz, W. Reif, and G. Schellhorn. Interactive verification of concurrent systems using symbolic execution. *AI Communications*, 23(2,3):285–307, 2010.

[BBR08] M. Balser, S. Bäumler, and W. Reif. An interval temporal logic with compositional interleaving. Technical Report 2008-20, University of Augsburg, 2008.

[BDRS06] M. Balser, C. Duelli, W. Reif, and J. Schmitt. Formal semantics of Asbru – v2.12. Technical Report 2006-16, University of Augsburg, June 2006. URL: www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. Representation and Mind Series. The MIT Press, 2008.

[BLAM+08] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *CAV'08*. Springer, 2008.

[BNBR08] S. Bäumler, F. Nafz, M. Balser, and W. Reif. Compositional Proofs with Symbolic Execution. In *Proceedings of the 5th International Verification Workshop in connection with IJCAR 2008*, volume 372 of *CEUR Workshop Proceedings*, 2008.

[BP99] M. Breitling and J. Philipps. Black-box views of state machines. Sfb-bericht nr. 342/07/99, TU München, 1999.

[BPS01] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science, 2001.

[BRSS99] M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. KIV 3.0 for Provably Correct Systems. In *Current Trends in Applied Formal Methods*, LNCS 1641. Boppard, Germany, Springer-Verlag, 1999.

[BSTR09] S. Bäumler, G. Schellhorn, B. Tofan, and W. Reif. Proving linearizability with temporal logic. *Formal Aspects of Computing (FAC)*, 2009. appeared online-first.

[CC96] A. Cau and P. Collette. Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. *Acta Informatica*, 33(2):153–176, 1996.

[CD07] R. Colvin and B. Dongol. Verifying lock-freedom using well-founded orders. In *Theoretical Aspects of Computing - ICTAC*

2007, volume 4711 of *LNCS*, pages 124–138. Springer-Verlag, 2007.

[CD09] R. Colvin and B. Dongol. A general technique for proving lock-freedom. *Sci. Comput. Program.*, 74(3):143–165, 2009.

[CDG05] R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *ENTCS*, 137:93–110, 2005.

[CG05] R. Colvin and L. Groves. Formal verification of an array-based nonblocking queue. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 507–516, Washington, DC, USA, 2005. IEEE Computer Society.

[CGP00] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[CH95] P. Clayton and G. Hripcsak. Decision support in healthcare. *International Journal of Bio-Medical Computing*, 39:59–66, 4 1995.

[CK95] P. Collette and E. Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. In *Algebraic Methodology and Software Technology*, number 936 in LNCS, pages 353 – 367, 1995.

[CL98] E. Cohen and L. Lamport. Reduction in tla. In *CONCUR '98: Proceedings of the 9th International Conference on Concurrency Theory*, pages 317–331, London, UK, 1998. Springer-Verlag.

[Cli08] C. Click. Towards a scalable non-blocking coding style. Talk at JavaOne conference. Available at www.azulsystems.com/events/javaone_2008/2008_CodingNonBlock.pdf, 2008. Code available at http://high-scale-lib.git.sourceforge.net/.

[CMZ02] A. Cau, B. Moszkowski, and H. Zedan. *ITL – Interval Temporal Logic*. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK, 2002. URL: http://www.cms.dmu.ac.uk/ cau/itlhomepage.

[CPV07] C. Calcagno, M. J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2007.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Springer, May 2001.

[DDG+04] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *SPAA '04:*

Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, pages 216–224, New York, NY, USA, 2004. ACM.

[DFG+00] D. L. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele Jr. Even better dcas-based concurrent deques. In *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73, London, UK, 2000. Springer-Verlag.

[DGLM04] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE 2004*, volume 3235 of *LNCS*, pages 97–114, 2004.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[Dij02] E. W. Dijkstra. Cooperating sequential processes. In Per Brinch Hansen, editor, *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[DMMJ01] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr. Lock-free reference counting. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 190–199, New York, NY, USA, 2001. ACM.

[Don06] B. Dongol. Formalising progress properties of non-blocking programs. In *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 284–303. Springer Berlin / Heidelberg, 2006.

[DOY06] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920, pages 287–302. Springer, 2006.

[dRdBH+01] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[dRE98] W.-P. de Roever and K. Engelhardt. *Data Refinement – Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[DSW07] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In J. Davies and J. Gibbons, editors, *IFM*, volume 4591 of *Lecture Notes in Computer Science*, pages 195–214. Springer, 2007.

[DSW08] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanising a correctness proof for a lock-free concurrent stack. In *Prooceedings of FMOODS 2008, Oslo*, volume 5051 of *LNCS*, pages 78–95, 2008.

[DSW10] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM transactions on Programming Languages and Systems*, 2010. (to appear).

[Dur98] Tom Durkin. What the media couldn't tell you about mars pathfinder. *Robot, Science & Technology*, 1, 1998.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *ACM*, 33(1):151–178, 1986.

[ehc94] Implementing clinical practise guidelines. Effective Health Care Bulletin, 1994. York: University of York.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

[FL92] M.J. Field and K.N. Lohr. *Clinical Practice Guidelines: Directions for a New Program.* National Academy Press, 1992.

[FOL01] D. Fober, Y. Orlarey, and S. Letz. Optimised lock-free fifo queue. Technical report, GRAME - Computer Music Research Lab., 2001.

[GC96] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 123–136, 1996.

[GC07] L. Groves and R. Colvin. Derivation of a scalable lock-free stack algorithm. *Electron. Notes Theor. Comput. Sci.*, 187:55–74, 2007.

[GC09] L. Groves and R. Colvin. Trace-based derivation of a scalable lock-free stack algorithm. *Formal Aspects of Computing (FAC)*, 21(1–2):187–223, 2009.

[GCPV09] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that nonblocking algorithms don't block. In *Principles of Programming Languages*, pages 16–28. ACM, 2009.

[Gen35] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39, 1935.

[GGH05] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distrib. Comput.*, 18(1):21–42, 2005.

[GGH07] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free parallel and concurrent garbage collection by mark&sweep. *Sci. Comput. Program.*, 64(3):341–374, 2007.

[GH07] H. Gao and W. H. Hesselink. A general lock-free algorithm using compare-and-swap. *Inf. Comput.*, 205(2):225–241, 2007.

[GL94] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[GTB+06] L. Giordano, P. Terenziani, A. Bottrighi, S. Montani, and L. Donzella. Model Checking for Clinical Guidelines: an Agent-based Approach. In *AMIA Annu Symp Proc.*, pages 289–293, 2006.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.

[Har84] David Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.

[Har01] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2001.

[Her03] M. Herlihy. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE Computer Society, 2003.

[HH01] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proc. of USENIX Annual Technical Conference*, pages 217 – 230, Boston, MA, USA, 2001. USENIX Association.

[HHL+07] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.

[HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. of the European symposium on programming on ESOP 86*, pages 187–196. Springer-Verlag New York, Inc., 1986.

[HLM02] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 339–353, London, UK, 2002. Springer-Verlag.

[HLMM05] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst*, 23(2):146–196, 2005.

[HSY04] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM Press.

[HW90] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83a] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon83b] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[JT96] B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theor. Comput. Sci.*, 167(1-2):47–72, 1996.

[Kal95] S. Kalvala. A formulation of TLA in Isabelle. In *In Higher Order Logic Theorem Proving and Its Applications (HOL'95), volume 971 of Lect. Notes in Comp. Sci*, pages 46–57. Springer-Verlag, 1995.

[KIV] Web presentation of the composition theorem and the lock-free stack and queue case study in KIV. URL: www.informatik.uni-augsburg.de/swt/projects/lock-free.html.

[KL93] R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In *Fifth International Workshop on Computer Aided Verification, Proceedings*, April 1993.

[Lam79] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, July 2002.

[Lam06a] L. Lamport. The +CAL algorithm language. Technical report, Microsoft, 2006.

[Lam06b] L. Lamport. Checking a multithreaded algorithm with +CAL. In *Distributed Computing*, pages 151–163. Springer Verlag, 2006.

[Leo07] T. Leonard. Dragged kicking and screaming: Source multicore. Talk at the Game Developers Conference, San Francisco, March 2007.

[Lip75] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LS85] L. Lamport and F.B. Schneider. *Distributed Systems: Methods and Tools for Specification*, volume 190 of *LNCS*, chapter 5, pages 203–285. Springer, 1985.

[LS89] L. Lamport and F. B. Schneider. Pretending atomicity. Technical Report TR89-1005, SRC Digital, 1989.

[Luc03] P. Lucas. Quality Checking of Medical Guidelines through Logical Abduction. In *Proc. of AI-2003, volume XX*, pages 309–321. Springer, 2003.

[LV95] N. Lynch and F. Vaandrager. Forward and Backward Simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995. also: Technical Memo MIT/LCS/TM-486.b, Laboratory for Computer Science, MIT.

[MBtTH] M. Marcos, M. Balser, A. ten Teije, and F. Van Harmelen. From informal knowledge to formal logic: a realistic case study in medical protocols.

[MC81] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE, Trans. on Software Eng.*, 1981.

[Mer95] S. Merz. Mechanizing TLA in Isabelle. In R. Rodošek, editor, *Workshop on Verification in New Orientations*, pages 54–74, Maribor, July 1995. Univ. of Maribor.

[Mic02] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA 2002*, pages 73–82. ACM, 2002.

[Mic03] M. M. Michael. CAS-based lock-free algorithm for shared deques. In *Euro-Par 2003*, volume 2790, pages 651–660. Springer LNCS, 2003.

[Mic04] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[Mis03] J. Misra. A reduction theorem for concurrent object-oriented programs. In A. McIver and C. Morgan, editors, *Programming methodology*, pages 69–92. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[Mos85] B. C. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[Mos86] B. C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.

[Mos00] B. C. Moszkowski. A complete axiomatization of interval temporal logic with infinite time. In *LICS 2000: Proc. of the 15th IEEE Symposium on Logic in Computer Science*, pages 241–252. IEEE Computer Society Press, 2000.

[MP91a] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1991.

[MP91b] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[MRtTvH02] M. Marcos, H. Roomans, A. ten Teije, and F. van Harmelen. Improving medical protocols through formalisation: a case study, 2002.

[MS96] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[OG76] S. S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.*, 6:319–340, 1976.

[ORY01] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, 2142, 2001.

[PA03] A. Pnueli and T. Arons. TLPVS: A PVS-based LTL verification system. In *Verification–Theory and Practice: Proceedings of an International Symposium in Honor of Zohar Manna's 64th Birthday*, Lect. Notes in Comp. Sci., pages 84–98. Springer-Verlag, 2003.

[PCMS+05] C. Polo-Conde, M. Marcos, A. Seyfang, J. Wittenberg, S. Miksch, and K. Rosenbrand. Assessment of mhb: an intermediate language for the representation of medical guidelines. In *Proc. of the 10th Conference of the Spanish Association for Artificial Intelligence (CAEPIA-05)*, pages I–19 – I–28, 2005.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. pages 123–144, 1985.

[Pre03] L. Prensa Nieto. The rely-guarantee method in Isabelle /HOL. In P. Degano, editor, *European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003.

[Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[Sch08] J. Schmitt. *Modellierung und Verifikation medizinischer Leitlinien.* PhD thesis, Universität Augsburg, Universitätsstr. 22, 86159 Augsburg, 2008. URL: http://opus.bibliothek.uni-augsburg.de/volltexte/2009/1355/.

[Sis94] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.

[SKM02] A. Seyfang, R. Kosara, and S. Miksch. Asbru 7.3 reference manual. Technical report, Vienna University of Technology, 2002.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[TBSR10] B. Tofan, S. Bäumler, G. Schellhorn, and W. Reif. Temporal logic verification of lock-freedom. In *Proc. MPC 2010*, Springer LNCS, 2010. (to appear).

[Tre86] R. K. Treiber. System programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[tTMB+06] A. ten Teije, M. Marcos, M. Balser, J. van Croonenborg, C. Duelli, F. van Harmelen, P. Lucas, S. Miksch, W. Reif, K. Rosenbrand, and A. Seyfang. Improving medical protocols by formal methods. *Artificial Intelligence in Medicine*, 36(3):193–209, 2006.

[Vaf07] V. Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2007.

[vdPW08] J. van de Pol and M. Weber. A multi-core solver for parity games. *Electron. Notes Theor. Comput. Sci.*, 220(2):19–34, 2008.

[VHHS06] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on*

Principles and practice of parallel programming, pages 129–136, New York, NY, USA, 2006. ACM.

[VY08] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 125–135, New York, NY, USA, 2008. ACM.

[XCC94] Q. Xu, A. Cau, and P. Collette. On unifying assumption-commitment style proof rules for concurrency. In *CONCUR*, volume 836 of *Lecture Notes in Computer Science*, pages 267–282, 1994.

[XdRH97] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. In *Formal Asp. Comput.*, 1997.

Anhang A

Präzedenzregeln

Präzedenzregeln	
höchste	$=$
Präzedenz	$:=$
	*
	$\Box,\ \Diamond,\ \circ,\ \bullet$
	until, **unless**
	if, **if***, **while**, **while***, **var**, **choose** , **await**
	$\|\|,\ \|\|\!\lfloor,\ \|\|\!\lfloor^b$
	;
	$\mathbf{A}, \mathbf{E}$
	$\neg$
	$\wedge$
	$\vee$
	$\supset$
	λ
	$\rightarrow$
niedrigste	$\leftrightarrow$
Präzedenz	$\forall,\ \exists$

Anhang B

Freie Variablen

Definition 37 *(Freie Variablen).* Sei $V \in \mathbf{V}$. Die Funktion

$$\text{free} : \mathbf{E} \to \mathcal{P}(\mathbf{V}),$$

berechnet für jeden Ausdruck die Menge der *freien Variablen*. Die Funktion ist wie folgt definiert:

$$\begin{aligned}
\text{free}(V) &:= \{V\} \\
\text{free}(V') &:= \{V\} \\
\text{free}(v'') &:= \{v\} \\
\text{free}(e'(e_1, \ldots, e_n)) &:= \text{free}(e_1) \cup \cdots \cup \text{free}(e_n) \\
\text{free}(\lambda \underline{x}.e) &:= \text{free}(e) \setminus \{\underline{x}\} \\
\text{free}(e \supset \varphi\ ;\ \psi) &:= \text{free}(e) \cup \text{free}(\varphi) \cup \text{free}(\psi) \\
\text{free}(f) &:= \emptyset \\
\text{free}(e_1 = e_2) &:= \text{free}(e_1) \cup \text{free}(e_2) \\
\text{free}(\exists\ V.\ \varphi) &:= \text{free}(\varphi) \setminus \{V\} \\
\text{free}(\varphi_1; \varphi_2) &:= \text{free}(\varphi_1) \cup \text{free}(\varphi_2) \\
\text{free}(\varphi^*) &:= \text{free}(\varphi) \\
\text{free}(\mathbf{step}) &:= \emptyset \\
\text{free}(\circ\ \varphi) &:= \text{free}(\varphi) \\
\text{free}(\varphi_1\ \mathbf{until}\ \varphi_2) &:= \text{free}(\varphi_1) \cup \text{free}(\varphi_2) \\
\text{free}([\varphi]_{\underline{V}}) &:= \text{free}(\varphi) \cup \{\underline{V}\} \\
\text{free}(\varphi \parallel \psi) &:= \text{free}(\varphi) \cup \cup\text{free}(\psi) \cup \{blk\} \\
\text{free}(\mathit{proc}(e_1, \ldots, e_n; \underline{V})) &:= \text{free}(e_1) \cup \cdots \cup \text{free}(e_n) \cup \{\underline{V}\} \\
\text{free}(\mathbf{A}\ \psi) &:= \text{free}(\varphi)
\end{aligned}$$

□

Die Freien Variablen der abgeleiteten Operatoren werden über deren Semantik bestimmt.

Anhang C

Lebenslauf

Persönliche Daten

	Geboren am 09. 05. 1974 in München
	Familienstand: ledig
	Staatsangehörigkeit: deutsch

Schulbildung

09/1981–07/1986	Grundschule Grafrath
09/1986–06/1995	Viscardi Gymnasium Fürstenfeldbruck

Zivildienst

01/1997–02/1998	Zivildienst im Kreiskrankenhaus München-Pasing

Studium

11/1995–12/1996	Studium der Mathematik an der Ludwig-Maximilians-Universität München
04/1998–03/2004	Studium der Informatik (Diplom) an der Ludwig-Maximilians-Universität München

Beruf

04/2004–10/2010	Wissenschaftlicher Mitarbeiter an der Universität Augsburg, Lehrstuhl für Softwaretechnik und Programmierung
seit 02/2011	Software-Ingenieur bei QAware GmbH

München, 15. April 2012