Entwurf einer formalen Semantik für Estelle unter Verwendung von TLA mit Prädikatentransformatoren

Diplomarbeit Nr. 858
Universität Hamburg
Fachbereich Informatik

Jan Bredereke

Bauernvogtkoppel 65a
D–2000 Hamburg 65

Betreuer: *Dr.-Ing. R. Gotzhein*
Zweitbetreuer: *Prof. Dr. F. H. Vogt*

Zusammenfassung

Die formale Beschreibungstechnik Estelle wird in einem internationalen Standard definiert. Ein Hauptnutzen einer formalen Semantik für eine Beschreibungssprache besteht darin, daß sie die formale Verifikation von Systembeschreibungen ermöglicht. Leider ist die im Standard enthaltene Semantikdefinition für Estelle nicht formal (und verständlich) genug, um formale Verifikation zu ermöglichen. Daher wird in dieser Arbeit ein Ansatz entwickelt, um die Semantik von Estelle vollständig formal und in einer für die Verifikation geeigneten Weise zu definieren. Für diesen Ansatz werden ausführliche Untersuchungen angestellt, insbesondere über die Methoden der Verifikation, die unterstützt werden müssen, und über eine geeignete Darstellung der sogenannten „Transitionen“ von Estelle. Um die hieraus resultierenden Forderungen zu erfüllen, wird ein neuer Formalismus entworfen, in dem Lamports temporale Logik der Aktionen und Dijkstras Prädikatentransformatoren vereinigt werden. Anschließend wird die Definition der gesamten Semantik von Estelle skizziert und die Definition des „Kerns von Estelle“, des sogenannten Ausführungsmodells, in diesem Formalismus vollständig ausgeführt. Es zeigt sich, daß der neue Ansatz die formale Verifikation von Estelle-Spezifikationen - bei mechanischer Unterstützung - nun möglich erscheinen läßt. Eine Ausarbeitung der Details des zum Formalismus gehörigen Schlußsystems und der skizzierten Gesamt-Semantik verbleibt allerdings zukünftigen Arbeiten.

English abstract

The formal description technique Estelle is defined by an international standard. A main advantage of a formal semantics for a description language is that it allows the formal verification of a system description. Unfortunately, the semantics definition in the Estelle standard is not formal (and intelligible) enough to render possible formal verification. So, this paper developes a way of how one can define a totally formal semantics for Estelle, suitable for verification. For this, extensive investigations are made, especially on the methods of verification that have to be supported, and on a suitable representation of the so-called “transitions” of Estelle. To meet the resulting requirements, a new formalism is designed by merging Lamport’s Temporal Logic of Actions and Dijkstra’s predicate transformers. Then, the definition of the entire semantics of Estelle is outlined and the central part of the Estelle definition, the so-called execution model, is carried out in this formalism completely. Our new way of defining the semantics showed that the formal verification of Estelle specifications can become feasible (provided mechanical support is available). However, the details of the proof system belonging to the formalism and the outlined entire semantics will have to be elaborated in future work.

Erklärung nach §23(9) DPO

Hiermit erkläre ich, daß ich diese Diplomarbeit selbständig durchgeführt habe und daß ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Jan Bredereke)

© 1992 Jan Bredereke

Vorwort

Danken möchte ich dem Betreuer meiner Arbeit, Herrn Dr.-Ing. R. Gotzhein, für die hervorragende Betreuung und die viele Zeit, die er dafür verwendet hat. Die vielen Diskussionen und Hinweise trugen wesentlich zum Gelingen der Arbeit bei. Das äußerst gründliche und sorgfältige Durcharbeiten der Zwischenversionen und die ausführlichen, konstruktiven Kommentare dazu verbesserten das Ergebnis wesentlich, trotz der Widrigkeiten unserer räumlichen Trennung in der zweiten Hälfte der Bearbeitungszeit. Gleichfalls gilt mein Dank dem Zweitbetreuer, Herrn Prof. Dr. F. H. Vogt, für das ständige, große Interesse, daß er für meine Arbeit zeigte, und für etliche entscheidende Anregungen.

Leslie Lamport gebührt Dank für die fruchtbaren Diskussionen über meine Arbeit und für die freigibigen Informationen über die aktuelle weitere Entwicklung der TLA. Peter Ladkin danke ich für hilfreiche Diskussionen über die TLA und über meine Arbeit und für seine Ermutigung. Der Estelle-Arbeitsgruppe am Fachbereich Informatik verdanke ich Austausch und Information über aktuelle Estelle-Aktivitäten. Meinem Vater möchte ich danken für das aufmerksame Korrekturlesen dieser Arbeit und für die Hinweise zu formalen Aspekten des Textes.

Hamburg, im Mai 1992

Jan Bredereke

Inhalt

Zusammenfassung / English abstract i

Vorwort iii

Inhaltsverzeichnis iv

1. Einleitung 1

2. Was ist Estelle

2.1 Ein kurzer Überblick 4
2.2 Kleine Einschränkungen des Sprachumfangs 12

3. Grundlagen: Semantikdefinitionstechniken

3.1 Methoden zur Definition der Semantik formaler Sprachen 13
3.2 Abstrakte Automaten 15
3.3 TLA: Die temporale Logik der Aktionen 17
3.4 Prädikatentransformatoren 26
3.5 Einige neue, nützliche Prädikatentransformatoren 36

4. Wahl einer Semantikdefinitionstechnik für Estelle

4.1 Auswahl einer grundsätzlichen Semantikdefinitionsmethode für Estelle 38
4.2 Welche Beweismethoden müssen unterstützt werden 42
4.3 Verschiedene prädikatenlogische Darstellungsformen für Estelle-Transitionen 47
4.4 Das untersuchte Beispiel: Der Euklidische Algorithmus 48
4.5 Untersuchung der prädikatenlogischen Darstellungsformen für Estelle-Transitionen 52
4.6 Zusammenfassung der Diskussion 68

5. TLA/PT: Erweiterung der TLA um Prädikatentransformatoren

5.1 Syntax und Semantik der TLA/PT 70
5.2 Strukturierte Variablen in der TLA/PT 78
5.3 Schlußregeln der TLA/PT 81
5.4 Weiteres aus der TLA^+ für die TLA/PT 83

6. Beschreibung der Semantik von Estelle-Texten in TLA/PT

6.1 Konzept der Semantikbeschreibung 84
6.2 Ein Demonstrationsbeispiel 89
6.3 Beschreibung von Typvereinbarungen 91
6.4 Beschreibung der Modulstruktur 94
6.5 Beschreibung von Interaktionspunkten 101
6.6 Beschreibung der Kommunikationsstruktur 109
6.7 Beschreibung des Transitionsteils 112

7. Definition des Ausführungsmodells

7.1 Das Ausführungsmodell ohne quantitative Zeitaspekte 120
7.2 Das Ausführungsmodell mit quantitativen Zeitaspekten 133

8. Ergebnisse 142

9. Ausblick 148

10. Literaturverzeichnis 151

11. Abbildungsverzeichnis 158

Anhang: Gesammelte Definition der TLA/PT 159

1. Einleitung

Estelle ist eine formale Beschreibungstechnik (FDT) für die Spezifikation von verteilten informationsverarbeitenden Systemen, insbesondere von Kommunikationsdiensten und -protokollen des OSI-Basisreferenzmodells ([ISO81]).[1]

Ganz allgemein werden Systeme mit einer (textuellen) Beschreibungstechnik beschrieben, indem eine Zeichenkette angegeben wird. Die Syntax der Beschreibungstechnik legt fest, welche Zeichenketten zur Beschreibungstechnik gehören, und ihre Semantik legt die Bedeutung der Zeichenketten fest, also welche Zeichenketten welche Systeme beschreiben.

Es ist schon länger üblich, die Syntax einer Beschreibungstechnik mit Hilfe einer geeigneten Grammatik formal zu definieren. (Genauer gesagt: Den kontextfreien Anteil der Syntax.) Für die Semantik der Beschreibungstechnik ist eine Formalisierung dagegen nicht so einfach wie für ihre Syntax.

Es gibt aber eine ganze Reihe von Gründen, auch die Semantik einer Beschreibungstechnik formal zu definieren. (Diese gelten sowohl für herkömmliche Programmiersprachen wie auch für abstraktere Spezifikationstechniken.)

1) Eine solche Definition der Semantik ist eine Präzisierung der weiterhin vorhandenen natürlichsprachlichen Beschreibung. Der Anwender kann mit ihrer Hilfe Zweifelsfälle entscheiden.
2) Falls die Konstruktion von automatischen Übersetzern für diese FDT sinnvoll ist, das heißt falls effiziente Verfahren existieren, dann gibt es für die Konstruktion hinreichend genaue Vorgaben (einschließlich aller bewußt gelassenen Freiheiten für den Implementator.)
3) Die Eigenschaften eines beschriebenen Systems lassen sich analysieren, und die Beschreibung kann gegen eine (noch) abstraktere formale Beschreibung verifiziert[2] werden.
4) Die Generierung und Validierung von Testfällen wird erleichtert, mit denen Implementationen des Systems getestet werden können. Und es wird bei der Entwicklung der Testfälle mehr maschinelle Unterstützung möglich.
5) Man erhält theoretische Hilfsmittel, um bestehende Beschreibungstechniken zu vergleichen und neue zu entwerfen.[3]

Je nach dem Ziel, mit dem eine Beschreibungstechnik entworfen wird, haben die einzelnen Punkte ein unterschiedliches Gewicht. Darauf abgestimmt wird dann auch die Methode der Formalisierung sein.

Im Falle von Estelle sind der erste und der dritte Punkt die wichtigsten, da Estelle nicht in erster Linie als ausführbare Sprache entworfen wurde, sondern als Spezifikationssprache. Der vierte Punkt ist für Estelle ebenfalls wichtig, aber er ist ohnehin mit der Eignung der FDT für Verifikation verbunden, da er ebenfalls mit einer Überprüfung von Korrektheit zu tun hat. Daher ist das Augenmerk bei einer formalen Beschreibung von Estelle insbesondere auch auf die Eignung der Methode für Verifikationsverfahren zu richten.

Wir werden hierauf zurückkommen.

Für Estelle wurde von der International Standardization Organization (ISO) ein

[1] Ein kurzer Überblick über Estelle findet sich in Kapitel 2.1.

[2] Allerdings verhindert bisher der Arbeitsaufwand für größere Systeme die Durchführung in der Praxis. (Aber siehe dazu die Kapitel 8 und 9.)

[3] Hierzu sind denotationale Beschreibungen der Semantik (siehe dazu Kapitel 3.1) am besten geeignet ([Lam80]).

internationaler Standard beschlossen ([ISO89a]). Entsprechend zu unserer Argumentation heißt es in dessen Einleitung unter anderem:[4]

> „Mit Hilfe formaler Techniken werden Systembeschreibungen möglich, die vollständig, widerspruchsfrei, genau, knapp und unzweideutig sind. Dies setzt voraus, daß eine formale Beschreibungstechnik in sich abgeschlossen ist, so daß die in ihr abgefaßten Beschreibungen sich auf keinerlei informelles Wissen über das beschriebene System beziehen müssen. Ein wichtiger Aspekt eines formalen Systems ist, daß es Analysen durch formale Methoden zuläßt. Eine formale Beschreibungstechnik auf einer solchen Grundlage kann dazu verwendet werden, die Korrektheit einer Spezifikation zu zeigen.
>
> Die in dieser Beschreibungstechnik abgefaßten Spezifikationen und Beschreibungen werden in dem Sinne als formal verstanden, daß es möglich ist, sie unzweideutig zu analysieren und zu interpretieren“.

In den restlichen Kapiteln des Textes wird Estelle dann (nachdem es zuerst informell anschaulich gemacht worden ist) in einer Kombination aus mathematischer Notation und natürlichsprachlichem Text[5] definiert. Damit ist die Beschreibungstechnik Estelle tatsächlich *formal* in dem genannten Sinne, sofern die Analyse durch einen Menschen durchgeführt wird, der den natürlichsprachlichen Text verstehen kann.

Zu den Kriterien, die eine formale Beschreibungstechnik für die Zwecke von OSI erfüllen sollte, wird in [ISO89a] unter anderem die *Wohldefiniertheit* gezählt:

> „Eine formale Beschreibungstechnik sollte ein formales Modell besitzen, das für die Verifikation von Spezifikationen und Definitionen geeignet ist. Dasselbe Modell sollte die Validierung von Implementationen, die von den internationalen Standards der OSI zugelassen werden, unterstützen. Dieses Modell sollte ebenfalls einen Konformitätstest von Implementationen unterstützen.“

Die Forderung, eine Analyse durch formale Methoden zuzulassen, wird vom Estelle-Standard nur unzureichend erfüllt, da die jetzige Beschreibung von Estelle einem vollständig formalen Kalkül nicht zugänglich ist. Dies zeigte sich zum Beispiel in der Studienarbeit [Bre90], in der eine in Estelle abgefaßte Spezifikation gegen eine Beschreibung der gewünschten Eigenschaften in temporaler Logik verifiziert werden sollte. Nur die Formeln in temporaler Logik konnten direkt durch syntaktische Umformungen manipuliert werden. Sobald auf Eigenschaften von Estelle-Konstrukten für die Korrektheitsbeweise zurückgegriffen wurde, mußte natürliche Sprache verwendet werden.[6]

Die Beweise waren zwar durchaus noch überzeugend, aber sie waren nicht mehr maschinell nachprüfbar, wie dies etwa Beweise im Kalkül der Prädikatenlogik sind. Daher war auch keine gute maschinelle Unterstützung bei der Arbeit des Beweisens möglich.

So war es wünschenswert, Estelle auf eine formale Basis in dem Sinne zu stellen, daß die Bedeutung einer in Estelle abgefaßten Spezifikation in einem formalen Kalkül, das heißt durch syntaktische Umformungen, abgeleitet werden kann.

---

[4] Übersetzung des englischsprachigen Originaltextes

[5] Beispiele für nur natürlichsprachliche Definitionen: In Kapitel 7 wird zwar die kontextfreie Syntax durch eine Grammatik in erweiterter Backus-Naur-Form definiert, aber die kontextsensitive Syntax wird nur in mit „Constraints“ betitelten Unterkapiteln auf Englisch beschrieben. Auch die Definition des „Ausführungsmodells“, zu dem wir unten noch kommen werden, in den Kapiteln 9.6.2 bis 9.6.5 enthält zwar viel mathematische Notation, aber wesentliche Aussagen und der Zusammenhalt der Formeln sind nur in Englisch beschrieben.

[6] Bei dem Text des Estelle-Standards kommt erschwerend hinzu, daß er sehr unübersichtlich und schlecht lesbar geschrieben ist. Es werden Hunderte von oft wenig mnemonischen Begriffen definiert, was zusammen mit einem umständlichen Schriftstil das Verstehen für den Leser sehr mühsam macht.

Bei der Beschreibung der Bedeutung (Semantik) einer Beschreibungsform mit syntaktischen Hilfsmitteln gerät man natürlich immer in ein Dilemma: Um nicht nur sinnleere Zeichenketten auf dem Papier stehen zu haben, müssen auch diese neuen Hilfsmittel eine Bedeutung haben, die ebenfalls auf irgend eine Weise festgelegt sein muß.[7] Daher ist uns nur geholfen, wenn wir es unterlassen dürfen, diese Bedeutung explizit anzugeben.

Dies dürfen wir dann tun, wenn einerseits jeder menschliche Leser exakt die gleiche Vorstellung von der Bedeutung haben wird und wenn andererseits die Beschreibungsform so einfach ist, daß sie von einer Maschine zu handhaben ist.

Im Gegensatz zu den in wenigen Jahrzehnten zu Abertausenden eingeführten Notationsformen der Informatik sind die wenigen Grundbegriffe der Mathematik und der Logik in Jahrtausenden so gut gefestigt worden, und ihre Formeln sind der maschinellen Verarbeitung so gut zugänglich, daß einfache Begriffe der Prädikatenlogik und der Mengenlehre unsere Forderung ausreichend erfüllen.

In dieser Arbeit entwickeln wir nun einen Ansatz zur durchgehend formalen Beschreibung der vollständigen Semantik einer in Estelle abgefaßten Spezifikation.

Als Grundlage dazu ist zunächst ein geeigneter mathematischer Formalismus notwendig. Da sich der ursprünglich ins Auge gefaßte Formalismus als nicht geeignet erwies, untersuchen wir in Kapitel 4 ausführlich, aus welchen Elementen unser Formalismus aufgebaut sein muß, um die Semantik von Estelle und inbesondere die Semantik der sogenannten Estelle-„Transitionen“ beschreiben zu können. In Kapitel 5 entwerfen wir dann diesen Formalismus. Dafür legen wir die „Temporale Logik der Aktionen“ (TLA) von Lamport ([Lam91]) zugrunde, und erweitern sie um Prädikatentransformatoren, wie sie Dijkstra populär gemacht hat ([DiSc90]). Beide Beschreibungsformen sind auf der Basis der Prädikatenlogik definiert, und damit ist die oben genannte Forderung nach einer einfachen Grundlage erfüllt. Für unsere „TLA/PT“ skizzieren wir außerdem über die Definition der Syntax und Semantik hinaus auch das zugehörige Schlußsystem, das man benötigt, um Eigenschaften von Spezifikationen ableiten zu können.

Auf der Basis der TLA/PT definieren wir in Kapitel 7 formal diejenigen Anteile von Estelle, die unabhängig von einer bestimmten Spezifikation sind, das sogenannte **Ausführungsmodell**. Die zeitliche Entwicklung eines in Estelle spezifizierten Systems wird im wesentlichen durch die Estelle-Transitionen beschrieben, und das Ausführungsmodell bestimmt, welche der Estelle-Transitionen wann und unter welchen Bedingungen dazu beitragen. Die Definition des Ausführungsmodells war das gesteckte Ziel unserer Arbeit, und mit ihm formalisieren wir den „Kern von Estelle“.

In Kapitel 6 zeigen wir auf, wie der Text einer Spezifikation in die TLA/PT umgesetzt werden kann, um dieses Gerüst aufzufüllen. Die Details dieser Umsetzung und die Semantik der vielen einzelnen Teilkonstrukte von Estelle arbeiten wir dann allerdings nicht mehr aus, da eine vollständig ausgeführte Definition der Semantik von Estelle den Rahmen einer Diplomarbeit bei weitem sprengen würde.

Schließlich fassen wir zusammen, was wir in dieser Arbeit erreicht haben und welche Auswirkungen dies auf das Gebiet der Verifikation von Estelle-Spezifikationen hat, und wir beschreiben, was noch zu tun verbleibt, um unsere Ergebnisse bis zur Anwendung in der Praxis zu tragen. (Kapitel 8 und 9)

---

[7] Hier besteht eine Verwandschaft mit dem *hermeneutischen Zirkel* der Philosophie: Dieser besteht in der Tatsache, daß von dem, was Gegenstand des Verstehens werden soll, vorher bereits durch eigene innere Erfahrung Wissen vorhanden sein muß ([Mey90]). Es kann kein Verstehen geben ohne ein bereits vorhandenes Vorwissen.

2. Was ist Estelle

2.1 Ein kurzer Überblick

Estelle ist eine formale Beschreibungstechnik (FDT) für die Spezifikation von verteilten informationsverarbeitenden Systemen, insbesondere von Kommunikationsdiensten und -protokollen des OSI-Basisreferenzmodells ([ISO81]). Estelle wurde von der International Standardization Organization (ISO) entwickelt und besitzt seit 1989 den Status eines internationalen Standards ([ISO89a]).

Eine gute Einführung in Estelle stellt [BuDe87] dar. Im folgenden werden wir uns teilweise locker an diesen Artikel anlehnen. Wir werden allerdings die Syntax von Estelle nicht näher beleuchten, da sie für unsere weitere Arbeit nicht wichtig ist und die gesamte Beschreibung aus [BuDe87] den Rahmen dieses Kapitels bereits sprengen würde. Wir beschränken uns daher auf die Darstellung der Estelle-typischen semantischen Konzepte. Am Ende dieses Kapitels werden wir allerdings eine Estelle-Spezifikation eines kleinen Schreiber-Leser-Systems angeben, damit sich der Leser dieser Arbeit wenigstens eine Vorstellung von den Notationen für die Konzepte machen kann.

Ein mit einer Estelle-Spezifikation beschriebenes System ist aufgebaut aus einer dynamischen Struktur von erweiterten endlichen Automaten, die miteinander kommunizieren, und deren Zustandsübergänge im wesentlichen durch Pascal-artige Anweisungen beschrieben werden.

Die Komponenten der dynamischen Struktur werden *Modulinstanzen* genannt. Sie können Punkte besitzen, an denen sie Nachrichten annehmen und abgeben können, diese werden *Interaktionspunkte* genannt. Die möglichen Zustandsübergänge einer Modulinstanz werden durch sogenannte *Estelle-Transitionen*[8] festgelegt. Besitzt eine Modulinstanz mindestens eine Estelle-Transition, so wird sie *aktiv* genannt, sonst *inaktiv*.

Modulinstanzen können hierarchisch strukturiert sein, das heißt, eine Modulinstanz kann *Sohn-Modulinstanzen* enthalten. „Enthalten“ bedeutet, daß von der äußeren Schnittstelle her nur die Vater-Modulinstanz sichtbar ist, daß deren Funktionalität aber mit Hilfe einer Untergliederung in weitere Modulinstanzen spezifiziert werden darf. Es gibt genau festgelegte, recht differenzierte Regeln für das Zusammenspiel der Modulinstanzen in der Hierarchie.

Eine Grundregel lautet, daß alle Sohn-Modulinstanzen blockiert sind, solange die Vater-Instanz noch Aktionen durchführen kann. (Hierdurch werden Konflikte bei dem Zugriff auf gemeinsame Ressourcen vermieden.) Außer dieser *Vater-Sohn-Vorrangregel* gibt es die Möglichkeit, verschiedene Grade von Parallelität zwischen den Sohn-Modulinstanzen zu spezifizieren. Dies geschieht durch die Angabe von Klassenattributen für die Modulinstanzen. Eine Modulinstanz kann eines der folgenden Attribute besitzen:

---

[8] Wir verwenden in dieser Arbeit den Begriff „*Estelle*-Transition“, um diesen Teil einer Estelle-Spezifikation sprachlich von den „Transitionen“ eines endlichen Automaten zu unterscheiden. Estelle-Transitionen beschreiben zwar Transitionen, aber erstens besitzen sie noch einige zusätzliche Eigenschaften, und zweitens gibt es Transitionen von Modulinstanzen, die keine Estelle-Transitionen sind. Wir werden solche nicht explizit spezifizierten Transitionen im Zusammenhang mit dem Ausführungsmodell von Estelle kennenlernen. (Stichworte dort: „Management-Phase“, „Auswählen von Estelle-Transitionen“)

- **SystemProcess**
- **SystemActivity**
- **Process**
- **Activity**

Auf den obersten Hierarchie-Ebenen dürfen die Modulinstanzen ohne Attribut sein, sie müssen dann inaktiv sein. Diese Art von Modulinstanzen gibt sozusagen den statischen Grobaufbau des Systems an. Die Söhne solcher Modulinstanzen müssen entweder ebenfalls ohne Attribut oder aber *System-Modulinstanzen* sein.

Diese werden auch *Subsysteme* genannt und müssen der Klasse **SystemProcess** oder **SystemActivity** angehören. Sie arbeiten völlig unsynchronisiert. Die weiteren Nachfahren von System-Modulinstanzen müssen eines der beiden Attribute **Process** oder **Activity** besitzen.

Mit dem Attribut **Process** oder **SystemProcess** spezifiziert man für die Söhne dieser Modulinstanz eine synchrone Parallelität: Es wird für alle ihre Söhne jeweils eine durchzuführende Aktion ausgewählt, und dann wird gewartet, bis alle Söhne damit fertig sind. Anschließend beginnt der Vorgang von vorn.

Bei dem Attribut **Activity** oder **SystemActivity** wird dagegen nichtdeterministisch immer nur ein einziger der Söhne ausgewählt, um eine Aktion durchzuführen. Diese Modulinstanzen dürfen nur noch Söhne ebenfalls aus der Klasse **Activity** besitzen. So wird die Abwesenheit von Parallelität spezifiziert.

Nachdem wir die Hierarchie der Modulinstanzen erläutert haben, müssen wir noch ein Wort zur Erzeugung und Vernichtung von Modulinstanzen verlieren. Eine Estelle-Spezifikation spezifiziert immer genau eine sogenannte *Spezifikations-Modulinstanz*. Dies ist die oberste Modulinstanz der Hierarchie, und sie existiert von Anbeginn an und für immer. Jedoch für alle weiteren Modulinstanzen werden lediglich generische Vorlagen spezifiziert, die *Moduldefinitionen* oder kurz *Module*.

Aufgrund der syntaktischen Regeln muß die Hierarchie der Moduldefinitionen statisch und endlich sein. Entsprechend ist auch die maximale Tiefe des Baumes von Modulinstanzen in jeder Situation statisch festgelegt. Die Breite des Baumes dagegen wird von Estelle nicht begrenzt. Eine bereits existierende Modulinstanz, wie zum Beispiel die Spezifikations-Modulinstanz, kann mit Hilfe einer Moduldefinition eine beliebige Anzahl von (gleichartigen) Sohn-Modulinstanzen erzeugen.

Wo eine Vater-Moduldefinition also jeweils eine Sohn-Moduldefinition besitzt, kann eine zugehörige Vater-Modulinstanz also `n` Sohn-Modulinstanzen besitzen. Umgekehrt gibt es zu jeder Sohn-Modulinstanz stets genau einen Vater. (Da eine Sohn-Modulinstanz ihrerseits wiederum Söhne erzeugen kann, kann es also zu einer Moduldefinition nicht nur danach geschaffene „Brüder“ geben, sondern auch „Vettern“.)

Das Erzeugen einer Modulinstanz, ebenso wie das Vernichten, ist immer Teil einer Estelle-Transition der jeweiligen Vater-Modulinstanz. Auch inaktive Modulinstanzen können Söhne erzeugen, da jede Modulinstanz unabhängig von den gewöhnlichen Estelle-Transitionen eine besondere *Initialisierungstransition* besitzen darf, die gleichzeitig mit der Erzeugung dieser Modulinstanz zur Anwendung kommt. Die Initialisierungstransition darf ausdrücklich auch weitere Sohn-Modulinstanzen erzeugen. Falls eine Modulinstanz inaktiv ist, ist die Struktur ihrer (direkten) Söhne anschließend folglich statisch.

Die Definition eines Moduls zerfällt in zwei Teile. Einerseits wird die äußere Schnittstelle festgelegt, und andererseits wird, getrennt davon, eine innere Verhaltensbeschreibung dazu festgelegt. Erstere wird auch als Modulkopf, letztere auch als Modulrumpf bezeichnet. Durch die getrennte Definition ist es möglich, zu einem Modul-

kopf mehrere, verschiedene Modulrümpfe zu definieren, die wahlweise zur Erzeugung einer Modulinstanz verwendet werden können. Es ist auch möglich, die Definition des Modulrumpfes (vorläufig) offenzulassen, indem man ihn mit dem Schlüsselwort **`external`** deklariert.

Der Zugriff auf Modulinstanzen erfolgt über *Modulvariablen*. Bei jeder Erzeugung einer Modulinstanz muß eine Modulvariable angegeben werden, über deren Namen die Modulinstanz künftig angesprochen werden kann. Wird zu einer Modulvariablen eine neue Modulinstanz erzeugt, bevor die alte vernichtet worden ist, so existiert die alte Modulinstanz weiterhin, kann aber nur noch indirekt angesprochen werden (zum Beispiel über ihre Kommunikationsverbindungen). Auf diese Weise können dynamisch beliebig viele Modulinstanzen erzeugt werden, ohne daß die statische Zahl der Modulvariablen-Vereinbarungen ein Hinderungsgrund wäre.

Nachdem wir die Modulinstanzen eingeführt haben, kommen wir nun dazu, wie diese miteinander kommunizieren können. Abgesehen von einer recht eingeschränkten Möglichkeit des Zugriffs auf gemeinsame Variablen[9], kommunizieren die Modulinstanzen nur durch den Austausch von *Nachrichten* (auch *Interaktionen* genannt). Die Schnittstellen, an denen sie Nachrichten abgeben und annehmen können, heißen *Interaktionspunkte*. Die Kommunikation ist immer asynchron, die sendende Instanz muß niemals auf die Annahme durch den Empfänger warten. Die Nachricht wird an eine unbeschränkte FIFO-Warteschlange angehängt und kann später vom Empfänger dort abgeholt werden.

Die Weiterleitung der Nachrichten wird durch eine Kommunikationsstruktur bewirkt, die aus einer dynamisch veränderbaren Konfiguration von (Interaktions-)Punkt-zu-Punkt-Verbindungen besteht.

Mit der Vereinbarung sogenannter *Kanäle* werden die Eigenschaften der Interaktionspunkte festgelegt. Da in Estelle eine Kommunikationsverbindung immer zwischen genau zwei Punkten besteht, besitzt eine solche Verbindung immer zwei Richtungen, und für beide wird jeweils eine Menge von zulässigen Nachrichten spezifiziert. Zu jedem Kanal gibt es somit zwei mögliche *Rollen*, die ein Interaktionspunkt übernehmen kann. Für jeden Interaktionspunkt muß in der Spezifikation (statisch) festgelegt werden, welche Rolle er für welche Art von Kanal übernehmen soll.

Die eben erwähnte Menge von zulässigen Nachrichten wird jeweils durch eine Aufzählung von Bezeichnern deklariert, die die verschiedenen Nachrichten kennzeichnen. Zusätzlich ist es möglich, Nachrichten zu parametrisieren. Ein Beispiel: `request(x: Integer; y: Boolean)` Auf diese Weise wird gleich eine ganze Klasse von Nachrichten auf einmal deklariert.

Mit der **`Connect`**-Operation können zwei Interaktionspunkte derselben Art von Kanal, aber mit entgegengesetzten Rollen, einander zugeordnet werden. Die Nachrichten, die an dem einen abgesandt werden, werden ab dann am anderen empfangen und umgekehrt. Die Übertragung geschieht dabei augenblicklich, die Nachricht wird mit der Aktion des Absendens in die Warteschlange des Empfängers eingereiht. Mit der **`Disconnect`**-Operation kann die Verbindung wieder gelöst werden.

Ein Interaktionspunkt kann *extern* sein, das heißt, daß er für die übergeordnete Modulinstanz sichtbar ist, oder er kann *intern* sein, so daß er nur der eigenen Modulinstanz bekannt ist. Es dürfen nicht beliebige Interaktionspunkte verbunden werden, die Hierarchie der Modulinstanzen muß beachtet werden. Mit der **`Connect`**-Operation dürfen nur interne Interaktionspunkte derselben Modulinstanz verbunden werden, oder jeweils an deren Stelle auch externe Interaktionspunkte der direkten

[9] Eine Vater-Modulinstanz darf auf die als „exportiert“ deklarierten Variablen ihrer Sohn-Modulinstanzen zugreifen.

Sohn-Modulinstanzen.

Es ist möglich, eine solche Sohn-Modulinstanz in einer strukturierten Weise zu spezifizieren, so daß diese die Bearbeitung der Nachrichten nicht selbst übernehmen muß, die über ihren externen Interaktionspunkt übertragen werden. Sie kann die Bearbeitung auch einer ihrer eigenen Sohn-Modulinstanzen überlassen. Hierfür kann sie mit der **Attach**-Operation ihren externen Interaktionspunkt mit einem externen Interaktionspunkt ihrer Sohn-Modulinstanz verbinden. Voraussetzung dafür ist, daß beide Interaktionspunkte die gleiche Rolle für die gleiche Art von Kanal spielen. Eine solche Verbindung kann sich über mehrere Rekursionsstufen durch die Modulinstanz-Hierarchie fortsetzen, und sie ist natürlich ebenso auch für die zweite Seite der **Connect**-Verbindung möglich. Eine Kommunikation kann nur zwischen den beiden Endpunkten einer solchen Kette stattfinden; eine solche Kette wird auch *Link* genannt. Gelöst werden kann eine mit der **Attach**-Operation hergestellte Verbindung entsprechend mit der **Detach**-Operation.

Es wurde bereits erwähnt, daß die an einem Interaktionspunkt empfangenen Nachrichten in einer FIFO-Warteschlange zwischengespeichert werden. Hier sind zwei wichtige Fälle zu unterscheiden. Jedem Interaktionspunkt, der mit dem Attribut **Individual Queue** deklariert worden ist, wird eine eigene, getrennte Warteschlange zugeordnet. Ist ein Interaktionspunkt aber mit dem Attribut **Common Queue** deklariert worden, dann werden seine Nachrichten in eine FIFO-Warteschlange eingeordnet, die alle derartigen Interaktionspunkte dieser Modulinstanz gemeinsam benutzen. An jedem Interaktionspunkt können zwar weiterhin nur die für ihn bestimmten Nachrichten entnommen werden, aber die erste Nachricht in der Warteschlange blockiert jeweils so lange alle weiteren, bis sie entnommen wurde.

Kommen wir nun dazu, wie das aktive Verhalten einer Modulinstanz spezifiziert wird. Ihre Verhaltensbeschreibung basiert wie bereits erwähnt auf dem Modell des erweiterten endlichen Automaten. Jede Modulinstanz besitzt also einen (lokalen) Zustand, und sie besitzt eine endliche Menge von möglichen Zustandsübergängen, die (Estelle-)Transitionen. Der Zustand wird insbesondere gekennzeichnet durch den *Hauptzustand*, der aus einer besonders deklarierten, endlichen Menge von möglichen Werten stammen muß. Erweitert werden kann der Zustand dadurch, daß für die Modulinstanz auch gewöhnliche Variablen wie in Pascal deklariert werden können. Die Zustandsübergänge zerfallen in zwei Teile: Der erste Teil legt fest, unter welchen Bedingungen der Übergang stattfinden kann, und der zweite legt die Wirkungen dabei fest. Für jede Modulinstanz kann immer nur eine eigene Estelle-Transition zur Zeit den Zustand verändern.

Folgende Bedingungen können für eine Estelle-Transition angegeben werden: Mit einer sogenannten **From**-Klausel kann gefordert werden, daß sich die Modulinstanz in einem bestimmten Hauptzustand oder einer bestimmten Menge von Hauptzuständen befindet. Mit einer **When**-Klausel kann der Empfang einer bestimmten Nachricht an einem bestimmten Interaktionspunkt zur Bedingung gemacht werden. Mit einer **Provided**-Klausel können beliebige Bedingungen über den erweiterten Zustand aus den Pascal-artigen Variablen formuliert werden. Mit einer **Priority**-Klausel kann der Vorrang einer Estelle-Transition vor anderen spezifiziert werden, falls ansonsten bei mehreren Estelle-Transitionen alle anderen Klauseln erfüllt sind. Bei gleicher Priorität wird sonst eine nichtdeterministische Entscheidung getroffen. Mit einer **Delay**-Klausel kann eine Estelle-Transition für eine bestimmte Anzahl von Zeiteinheiten blockiert werden.

Alle diese Klauseln sind optional, bei ihrem Fehlen entfallen die daraus erwachsenden Einschränkungen[10]. Fehlt die **When**-Klausel, heißt die Estelle-Transition *spontan*. Nur für spontane Estelle-Transitionen darf mit einer **Delay**-Klausel ei-

ne Verzögerung spezifiziert werden.

Der Transitionsrumpf, mit dem die Wirkungen der Estelle-Transition beschrieben werden, enthält im wesentlichen Pascal-artige Anweisungen. Dabei treten allerdings alle Wirkungen einer Estelle-Transition *atomar* ein. Aufgrund der Atomizität ist es nicht möglich, daß sich die Wirkungen von zwei verschiedenen Estelle-Transitionen in irgend einer Weise „zeitlich verzahnen". Wir sprechen deshalb auch nicht von der „Ausführung" einer Estelle-Transition, sondern von ihrem *Schalten*. Außer den üblichen Pascal-Anweisungen gibt es noch einige neue Anweisungen zur Steuerung der Modulinstanz- und Kommunikationsstruktur. Die Anweisungen **Connect**, **Disconnect**, **Attach** und **Detach** hatten wir bereits erwähnt. Mit den Anweisungen **Init**, **Terminate** und **Release** wird die Erzeugung und Vernichtung von Sohn-Modulinstanzen gesteuert. Die **Output**-Anweisung schließlich dient dazu, eine Nachricht durch einen Interaktionspunkt abzusenden. Ferner gibt es einige kleine, nützliche Erweiterungen zu Pascal und einige Beschränkungen, zum Beispiel was die Verwendung von Zeigern und von Sprunganweisungen oder das Fehlen von Dateioperationen betrifft.

Eine Estelle-Transition kann eine **To**-Klausel besitzen. Diese beschreibt keine Bedingungen, sondern gibt eine weitere Wirkung an; sie legt einen neuen Hauptzustand fest. (Die oben bereits erwähnte **When**-Klausel beschreibt implizit ebenfalls eine Wirkung, und zwar die Entnahme der genannten Nachricht aus der Warteschlange.) Ferner kann jedem Transitionsrumpf mit dem **Name**-Konstrukt ein Name zugeordnet werden. Dieser trägt keinerlei Bedeutung und dient nur zu Dokumentationszwecken, genau wie ein (Programm)-Kommentar.

Um die Übersichtlichkeit zu erhöhen, ist es erlaubt, mehrere Estelle-Transitionen, die eine oder mehrere gleiche Klauseln besitzen, zusammenzufassen, so daß diese Klauseln nur noch einmal aufgeschrieben werden müssen. Mit einem einfachen, nur an der Syntax orientierten Algorithmus (ein Beispiel findet sich in [AmPrSc88]) lassen sich solche Blöcke jederzeit wieder in getrennte Estelle-Transitionen auflösen. Der ISO-Standard setzt für seine formale Definition von Estelle voraus, daß eine solche Auflösung bereits stattgefunden hat.

Im Zusammenhang mit der **Delay**-Klausel hatten wir bereits kurz das Konzept der „*Zeit*" erwähnt. Estelle-Spezifikationen sind im wesentlichen unabhängig von „Bearbeitungszeiten" formuliert, diese Informationen werden als implementationsabhängig betrachtet. Daher wird von einer Estelle-Spezifikation keinerlei Aussage über die relative oder absolute Geschwindigkeit von Vorgängen gemacht. Die einzige Ausnahme bildet die Möglichkeit, das Schalten einer Estelle-Transition um eine bestimmte Anzahl von Zeiteinheiten zu verzögern. (Wie schnell die Estelle-Transition nach Ablauf dieser Zeitspanne dann tatsächlich schaltet, hängt wie sonst auch von der Geschwindigkeit der Implementation ab.) Es wird die Existenz eines Zeit-Prozesses angenommen, aber er dient lediglich dazu, spezifizierte Zeitintervalle in der richtigen Reihenfolge ablaufen zu lassen. Über feste Zeitpunkte kann keine Aussage gemacht werden.

Die Definition der Semantik von Estelle erfolgt im ISO-Standard operational (siehe zu diesem Begriff auch Kapitel 3.1). Dies bedeutet, daß eine sogenannte *Nächster-Zustand-Relation* über einer Menge von *globalen Zuständen* definiert wird, die *globale Situationen* genannt werden. Die Relation gibt an, welche Situationen von einer gegebenen Situation aus erreicht werden können. Das Gesamtverhalten eines in Estelle spezifizierten Systems wird durch die Menge aller möglichen Sequenzen von globalen Situationen charaktierisiert, die von einer bestimmten *initialen* Situation aus erreicht werden können.

[10] Eine fehlende **Priority**-Klausel bedeutet allerdings die niedrigste Priorität.

Eine globale Situation besteht aus der aktuellen Information über die *globale Momentbeschreibung* `gid(SP)` einerseits und über die Estelle-Transitionen, die gerade in „paralleler (synchroner) Ausführung“[11] für jeweils ein Subsystem sind, andererseits. Für das `i`-te Subsystem wird deren Menge mit A_i bezeichnet. A_i kann eine Estelle-Transition dieses Subsystems enthalten oder auch mehrere Estelle-Transitionen aus dessen Sohn-Modulinstanzen. Die globale Momentbeschreibung `gid(SP)` enthält die aktuelle hierarchische Struktur der Modulinstanzen, die aktuellen Kommunikationsverbindungen und den lokalen Zustand jeder Modulinstanz. Ist eine der Mengen A_i leer, so sagt man, daß das zugehörige Subsystem in seiner *Managementphase* sei.

Die Nächste-Situation-Relation beschreibt, wie sich `gid(SP)` und die A_i von Situation zu Situation verändern können. Vieles davon hatten wir bereits zu Anfang im Zusammenhang mit der Modulinstanz-Struktur angedeutet. [BuDe87] skizziert recht detailliert und trotzdem übersichtlich, wie dies alles vom ISO-Standard formalisiert wird. Interessant ist insbesondere die Menge der gerade in „paralleler (synchroner) Ausführung“ befindlichen Estelle-Transitionen. Nur in der Managementphase eines Subsystems, also wenn seine Menge A_i leer ist, können Estelle-Transitionen dahinein aufgenommen werden. Danach verlassen sie eine nach der anderen diese Menge wieder, und dabei tritt jeweils die Wirkung der entsprechenden Estelle-Transition atomar ein. Sobald eine Estelle-Transition in diese Menge aufgenommen ist, ist sichergestellt, daß ihre Wirkung eintreten darf. So wird einerseits die konzeptuell wichtige Atomizität der Zustandsübergänge gewährleistet, und andererseits ist es einer Implementation trotzdem freigestellt, mit der Berechnung der Wirkung intern zu beginnen, sobald die Estelle-Transition in die Menge aufgenommen worden ist. Falls mehrere Estelle-Transitionen (aus verschiedenen Sohn-Modulinstanzen) in die Menge aufgenommen wurden, so ist nichtdeterministisch jede Permutation über die Reihenfolge des Eintretens der Wirkungen zulässig.

Auch wenn wir unvermeidlicherweise vieles nur sehr kurz anreißen konnten oder überhaupt auslassen mußten, wollen wir hiermit diesen Überblick beschließen. Für weitere Details verweisen wir auf [BuDe87] oder den ISO-Standard [ISO89a]. (Abgesehen natürlich von unseren Kapiteln 6 und 7, in denen wir unter anderem das Ausführungsmodell sogar noch wesentlich exakter beschreiben werden als der ISO-Standard.) Es folgt nun lediglich noch die versprochene kurze Beispiel-Spezifikation, um eine Vorstellung von der syntaktischen Struktur einer Estelle-Spezifikation zu vermitteln, und um eine Einordnung der vorangegangenen Ausführungen zu erleichtern.

[11] Dieser Begriff stammt aus [BuDe87]. Warum wir diese Charaktierisierung - wenn auch in Anführungsstrichen - an dieser Stelle übernommen haben, und warum wir die Menge nicht durch den Begriff „paralleles Schalten“ gekennzeichnet haben, wird bis zum Ende des nächsten Absatzes klar werden.

```
Specification schreiber_leser;
{ Dieses System besteht aus zwei Subsystemen. Eines davon
  produziert Daten, das andere verbraucht sie. Der Erzeuger
  wartet mit dem naechsten Datum, bis das vorige quittiert
  wurde.
}
  Channel kanal(schreiber, leser);  { Die Vereinbarung des Verbindungs-
                                        kanals zwischen beiden.
                                        Die moeglichen Rollen fuer IPe
                                        sind „schreiber“ und „leser“ }
    By schreiber:
      daten(x: integer);            { Eine Nachricht mit Parameter. }
    By leser:
      quittung;                     { Eine parameterlose Nachricht. }
  Module erzeuger_typ SystemProcess;{ Definition eines Modulkopfs. }
    Ip ausgabe: kanal(schreiber) Individual Queue;
  End;                              { Dieser IP ist extern sichtbar,
                                        da er im Modulkopf vereinbart
                                        wird. }
  Module verbraucher_typ SystemProcess;
                                    { Dito fuer das 2. Subsystem. }
    Ip eingabe: kanal(leser) Individual Queue;
  End;
  Body einfach_erzeuger For erzeuger_typ;
                                    { Ein Modulrumpf zum ersten
                                        Modulkopf. }
    State                           { Die Menge der moeglichen }
      ruhend,                       {    Hauptzustaende. }
      wartend;
                                    { Hier kein erweiterter Zustand. }
    Initialize                      { Die Initialisierungs-Transition. }
      To ruhend                     { Anfänglicher Hauptzustand. }
Name init_erzeuger:                 { Name dieser Estelle-Transition. }
        Begin                       { Hier keine weiteren Aktionen. }
        End;
    Trans                           { Eine Estelle-Transition. }
                                    {    Sie kann spontan schalten. }
      From ruhend                   { Bedingung fuer den Hauptzustand. }
        To wartend                  { Neuer Hauptzustand danach. }
Name sende_daten:                   { Name der Estelle-Transition. }
          Begin                     { Der Transitionsrumpf: }
            Output ausgabe.daten(42) { Ausgabe einer Nachricht. }
          End;
    Trans                           { Noch eine Estelle-Transition. }
      When ausgabe.quittung         { Bedingung: Empfang einer }
        From wartend                {    Nachricht. }
          To ruhend
Name empfange_quittung:
            Begin
            End;
  End; { Body einfach_erzeuger For erzeuger_typ }
```

```
  Body einfach_verbraucher For verbraucher_typ;
                                    { Ein Modulrumpf zum zweiten
                                        Modulkopf. }
                                    { Hier (implizit) nur ein
                                        einziger Hauptzustand. }
                                    { Hier keine Initialisierung
                                        notwendig. }
    Trans                           { Eine Estelle-Transition. }
      When eingabe.daten(zahl)      { Hiermit wird die Variable }
        Begin                       {   „zahl“ gemaess der }
          { Verarbeite "zahl". }    {   Parametervereinbarung }
                                    {   vom Anfang deklariert. }
          Output eingabe.quittung
        End;
  End; { Body einfach_verbraucher For verbraucher_typ }
  Modvar                            { Modulvariablen, fuer die }
    erzeuger: erzeuger_typ;         {   Modulinstanzen. }
    verbraucher: verbraucher_typ;
  Initialize                        { Initialisierung des }
Name init_system:                   {   (statischen) Rahmens. }
    Begin
      Init erzeuger With einfach_erzeuger;
                                    { Erzeugung einer Modul-
                                        instanz mit einem bestimmten
                                        Modulrumpf. }
      Init verbraucher With einfach_verbraucher;    { Dito. }
                                    { Verbinden der IPe: }
      Connect erzeuger.ausgabe To verbraucher.eingabe
    End;
End. { Specification schreiber_leser; }
```

Abbildung 2-1: Ein einfaches Schreiber-Leser-System

2.2 Kleine Einschränkungen des Sprachumfangs

Im letzten Kapitel wurde Estelle so beschrieben, wie es von der ISO definiert wurde. Mit unserem neuen Ansatz werden wir diesen Sprachumfang fast vollständig abdecken, einschließlich der schwierig zu definierenden Semantik zum Beispiel

- von Interaktionspunkten, die mit dem Attribut **`Common Queue`** deklariert sind,
- von quantitativen Zeitaspekten der Estelle-Transitionen mit **`Delay`**-Klauseln,
- von Transitionsrümpfen, die auch Pascal-artige **`While`**-Schleifen enthalten, deren Wirkung aber trotzdem atomar eintritt,
- ...

Einige kleine Einschränkungen des Sprachumfangs werden wir trotzdem machen:

- In Kapitel 6.7 werden wir fordern, daß jeder Estelle-Transition ein Name mit dem **`Name`**-Konstrukt zugeordnet werden *muß* (bisher war dies lediglich erlaubt).
- In Kapitel 6.4 werden wir fordern, daß auf jede Modulinstanz genau eine Modulvariable zeigen muß. (Womit Zuweisungsoperationen auf Modulvariablen untersagt werden, und womit eine neue Modulinstanz mit der **`Init`**-Anweisung nur dann erzeugt werden darf, wenn die dabei verwendete Modulvariable nicht bereits auf eine existierende Modulinstanz verweist.)
- Ebenfalls in Kapitel 6.4 schließen wir die Verwendung von Zeigervariablen aus.

Die erste Einschränkung, die obligatorische Namen fordert, ist eine rein syntaktische. Diese Namen können in jede Estelle-Spezifikation ohne Veränderung ihrer Bedeutung eingefügt werden. (Die Namen müssen für unsere Zwecke nicht einmal unbedingt global eindeutig sein.)

Diese Forderung erleichtert uns lediglich die Definition der Semantik, indem nicht erst Namen für anonyme Estelle-Transitionen generiert werden müssen. Man könnte diese Bedingung sicherlich auch eliminieren, aber für die Verifikation einer Estelle-Spezifikation sind solche Namen ohnehin sehr nützlich, so daß wir in einer Aufhebung der Forderung keinen Sinn sehen.

Die dritte Forderung bedeutet keine ernsthafte Einschränkung bei der Spezifikationen von Systemen in Estelle, da Zeiger dort kaum Anwendung finden. In der Definition der ISO wurde ohnehin bereits ihre Verwendung eingeschränkt.

Die zweite Forderung ergibt sich aus der dritten, wie wir in Kapitel 6.4 sehen werden. Dort werden wir auch beschreiben, wie beide Einschränkungen aufgehoben werden könnten. Aber wir werden auch erläutern, daß der zusätzliche Aufwand bei der Definition und vor allem bei der Nutzung eines Formalismus zur Semantikbeschreibung so hoch wäre, daß uns der Nutzen erst einmal nicht die Kosten wert erscheint.

Als tatsächliche Einschränkung der Beschreibungskraft von Estelle ergibt sich aus der zweiten Forderung, daß zu einer Moduldefinition nicht mehr beliebig viele Modulinstanzen erzeugt werden können, sondern höchstens noch so viele, wie Modulvariablen dazu vereinbart wurden. Solange man Systeme spezifizieren will, bei denen vorher eine obere Schranke für die Anzahl der zu erzeugenden Modulinstanzen angegeben werden kann, ist dies kein Hindernis, da man einfach entsprechend viele Modulvariablen vereinbaren kann (beispielsweise als Feld von Variablen).

3. Grundlagen: Semantikdefinitionstechniken

In Kapitel 3 wird zunächst ein Überblick über die grundsätzlichen Methoden zur Definition der Semantik formaler Sprachen gegeben, dann wird das operationale Konzept des abstrakten Automaten noch etwas ausführlicher erläutert, und schließlich werden zwei spezielle Semantikdefinitionstechniken vorgestellt, die in der weiteren Arbeit eine wichtige Rolle spielen werden.

3.1 Methoden zur Definition der Semantik formaler Sprachen

In diesem Kapitel wird zusammengetragen, welche grundsätzlichen Methoden existieren, um die Semantik einer Programmiersprache formal zu definieren, und welche Vor- und Nachteile die einzelnen Ansätze haben. Dies geschieht zunächst einmal ohne Bezug auf irgendeine bestimmte Beschreibungs*technik*. Basierend auf diesem Überblick wird dann später in Kapitel 4.1 die grundsätzliche Vorgehensweise bei der Neudefinition der Semantik für die formale Spezifikationstechnik Estelle festgelegt. Dies ist möglich, da sowohl Programmiersprachen als auch formale Spezifikationstechniken Spezialfälle von Sprachen allgemein sind und die vorhandenen Untersuchungen nicht die besonderen Eigenschaften von Programmiersprachen ausnutzen.

Kurz zusammengefaßt kann man eine Semantik (für sequentielle Programme) auf folgende Arten angeben (in der Einteilung von [Eng88]):

• **Übersetzersemantik**: Die Sprachkonstrukte werden auf der Basis einer anderen, einfacheren (Assembler-)Sprache definiert. Dies ist interessant für die Konstruktion von Übersetzern, ansonsten wird das Grundproblem aber nur verlagert.

• **operationale Semantik** (Interpretersemantik): Es wird ein abstrakter Interpreter definiert, der jeweils für bestimmte Eingabedaten durch schrittweise Abarbeitung des Programms die Ausgabedaten erzeugt. Die operationale Semantik ist nicht für Verifikation geeignet, da sie nur für feste und nicht für alle Eingabewerte arbeitet.

Der abstrakte Interpreter transformiert Paare aus je einem Programm und einem Zustand solange, bis das Programm leer ist. Bestehe die Eingabe am Anfang vereinfacht in einem Zustand, ebenso die Ausgabe am Schluß. Dann ist die Semantik der Sprache

$$f: P \times Z \rightarrow Z$$
$$f(p,z_e) = z_a$$

• **denotationale Semantik** (Funktionensemantik): Es wird vom konkreten Maschinenmodell abstrahiert und nur noch die Wirkung einer Anweisung auf die möglichen Zustände betrachtet. Eine Anweisung ordnet jedem Zustand einen Folgezustand zu (Systematische Funktion F). Man muß ein Gleichungssystem lösen.

Für ein bestimmtes Programm erhält man schließlich als Semantik die Lösung des Gleichungssystems, eine Funktion $f: Z \rightarrow Z$

Folglich liegt der Unterschied zwischen operationaler und denotationaler Semantikdefinition im Betrachtungsschwerpunkt und im Lösungsverfahren, das Grundprinzip

aber ist bei beiden gleich.

• **axiomatische Semantik** (Prädikatensemantik): Sie ist noch eine Abstraktionsebene höher angesiedelt. Es wird nicht mehr die Zuordnung von (Nach- zu Vor-)Zuständen durch eine Anweisung betrachtet, sondern es werden nur noch Eigenschaften von Zuständen vor und nach einer Anweisungsfolge betrachtet. Damit entfällt auch die Betonung der uninteressanten Zwischenzustände, wie sie in der operationalen Semantik notwendig ist. Die axiomatische Semantik ist gut für Verifikation geeignet.

Lamersdorf hat in [Lam80] die einzelnen Semantikdefinitionsmethoden ausführlich verglichen (wobei er die Übersetzersemantik allerdings nicht von der operationalen Semantik trennt), und kommt zu der folgenden prägnanten, aber auch nach eigener Aussage stark verkürzenden Liste von jeweils bevorzugten Einsatzgebieten:

• operationale Semantik: Für Sprachimplementatoren

• denotationale Semantik: Für den Neuentwurf von Programmiersprachen

• axiomatische Semantik: Für die Verifikation

Zu Beginn der achtziger Jahre wandelte sich das Paradigma der Informatik von der programmgesteuerten Rechenanlage, die Eingabedaten zu Ausgabedaten verarbeitet, hin zu einem System von intelligenten, interagierenden Individuen ([Bra91]). Auf letzterem Paradigma beruht auch Estelle. Damit verbunden gewannen die Konzepte der Nebenläufigkeit und des Nichtdeterminismus′ an Bedeutung. Daher wurden gleichzeitig die oben angeführten Semantikdefinitionsmethoden um Möglichkeiten zur Beschreibung von Nebenläufigkeit und von Nichtdeterminismus erweitert ([Bak89], [BoCa86], [BoCa88], [BrNe89], [Dij76], [DiSc90], [Har88], [Hoa85], [HoHe85], [HoRaRo90], [Lam91], [MaNe87], [Old87], [Plo82], [Tof90], [ZwRo89] und viele andere). Die jeweilige Grundidee der Beschreibungsform blieb jedoch unverändert, so daß die obige Einteilung weiterhin gültig ist.

Entsprechend den beiden von [Bra91] genannten Paradigmen teilt [Pnu86] rechnergesteuerte Systeme (Programme) in zwei Grundkategorien ein: In die *transformationellen* Systeme, die genau eine Eingabe bekommen und genau eine Ausgabe liefern, und in die *reaktiven* Systeme, die nicht terminieren (müssen), und die (viele) Interaktionen mit ihrer Umgebung haben können. Insbesondere dürfen dabei weitere Eingaben aus der Umgebung von früheren Zwischen-Ausgaben des Systemes abhängen.

3.2 Abstrakte Automaten

Der Vollständigkeit halber wollen wir in diesem Kapitel kurz erläutern, wie ein abstrakter Automat definiert ist, da er die theoretische Grundlage des operationalen Ansatzes darstellt. Anschließend werden wir sofort den Automaten vereinfachen und die Teile weglassen, die wir für unsere Zwecke nicht benötigen werden.

In [Eng88] wird der herkömmliche operationalen Ansatz folgendermaßen beschrieben:

Für sequentielle, deterministische Sprachen wird die operationale Semantik definiert durch den abstrakten Automaten M'' mit

$$M''=(I,O,K,\alpha,\omega,t',\pi)$$

Dabei ist

I	die Menge der Eingabedaten,
O	die Menge der Ausgabedaten,
$K=A\times Z$	die Menge der Konfigurationen,
A	die Menge der Anweisungsfolgen,
Z	die Menge der Zustände,
$\alpha\colon I\to K$	die Eingabefunktion,
$\omega\colon K\to O$	die Ausgabefunktion,
$t''\colon K\to K$	die Übergangsfunktion und
$\pi\colon K\to\{0,1\}$	das Halteprädikat

In diesem allgemeinen Ansatz ist vieles enthalten, was wir zur Beschreibung von Estelle nicht brauchen werden. Da es in Estelle keine Termination des Gesamtsystems gibt, entfällt das Halteprädikat π, und da das Gesamtsystem geschlossen ist, entfallen die Menge der Ausgabedaten O sowie die zugehörige Ausgabefunktion $\omega\colon K\to O$, und die Menge der Eingabedaten I und die Eingabefunktion $\alpha\colon I\to K$ können durch eine Startkonfiguration k_0 ersetzt werden. Da Nichtdeterminismus zugelassen werden muß, ersetzen wir die Übergangsfunktion $t''\colon K\to K$ durch eine Übergangsrelation $t'\subseteq K\times K$, und die Startkonfiguration k_0 ersetzen wir gleich wieder durch eine Menge von Startkonfigurationen K_0.

Der so abgewandelte Automat $M'=(K_0,Z,t')$ nimmt eine der Startkonfigurationen aus K_0 und transformiert sie gemäß der Übergangsrelation $t'\subseteq K\times K$. Dabei besteht eine Konfiguration aus einer Spezifikation a und einem Zustand z, $K=A\times Z$. Entsprechend besteht ein Element aus K_0 aus einer Startspezifikation a_0 und einem Startzustand z_0.

Der Automat zur Definition der Semantik sequentieller Programme modifizierte das Programm a aus zwei Gründen: Einmal mußte der „Befehlszähler“ des Interpreters dargestellt werden, und zweitens - damit geschickt verbunden - wurden Programmteile entfernt, die fertig bearbeitet waren. Sobald nichts mehr übrig geblieben war, terminierte der Automat.

Da es in Estelle keine „Befehlszähler“ gibt, sondern sich der Zustand der einzelnen Modulinstanzen durch atomares Schalten von z.B. Estelle-Transitionen ändert, und da Teile einer Spezifikation a niemals ausgedient haben, weil es keine Termination (des Gesamtsystems) gibt, ist es nicht sinnvoll, daß der Automat für Estelle die Spezifikation verändert. Daher lassen wir die Übergangsrelation nicht Konfigurationen transformieren, sondern nur Zustände, aber natürlich in Abhängigkeit von einer Spezifikation:

$t_a \subseteq Z \times Z$

Anstelle von Startkonfigurationen K_0 verwenden wir entsprechend Startzustände $Z_{0,a}$.
So ergibt sich schließlich als Definition der Semantik einer speziellen Spezifikation in Estelle der Automat

$M = (Z_{0,a}, Z, t_a)$

Dabei ist, wie bereits im Text erläutert,

A	die Menge der Spezifikationen,
$a \in A$	eine spezielle Spezifikation,
Z	die Menge der Zustände,
$Z_{0,a}$	die Menge der Startzustände der speziellen Spezifikation a und
$t_a \subseteq Z \times Z$	die Zustandsübergangsrelation für die spezielle Spezifikation a.

3.3 TLA: Die temporale Logik der Aktionen

In diesem Kapitel beschreiben wir die TLA, so wie sie von Lamport in [Lam91] eingeführt wurde.

Ziel und Einsatzgebiet

Die TLA ist insbesondere gedacht für das Gebiet der verteilten Systeme. Dort möchte man Algorithmen für reale Anwendungen schreiben, und man möchte die Eigenschaften aufschreiben, von denen man hofft, daß die Algorithmen sie besitzen. Dies führt zu drei verschiedenen Konzepten:

- Programm
- Eigenschaft
- erfüllt

Lamport schlägt vor, diese drei in *ein einziges Konzept* zu integrieren:

- logische Formel

Der Grund für diese Integration liegt darin, daß formales Schließen nur dann praktisch durchführbar ist, wenn der zugrundeliegende Formalismus *einfach* ist.

Es stellt sich natürlich die Frage, was die TLA nützt, wenn man einen Algorithmus immer noch in ein Programm kodieren muß, um ihn nutzen zu können. Aber nach Lamport sind es die Timing-abhängigen Synchronisationsfehler, die der Fluch der parallelen Programmierung sind. (Wir können dies bestätigen. In unserer Studienarbeit [Bre90] zeigte die Verifikation eines einfachen Protokolls, welch tückische, für einen normalen Leser nicht erkennbare Fehler in einer Estelle-Spezifikation noch vorhanden waren.) Aus diesem Grunde argumentiert man nach Lamport so gut wie immer über den abstrakten Algorithmus, nicht das kodierte Programm. Und abstrakte Algorithmen passen normalerweise selbst dann auf eine einzige Seite, wenn die Programme später sehr groß werden.

Die TLA ist also dazu gedacht, zuerst die Eigenschaften eines Algorithmus′ zu spezifizieren, ihn anschließend in Verfeinerungsschritten zu entwickeln und zum Schluß zu zeigen, daß das resultierende Programm die Eigenschaften erfüllt („$\Psi \Rightarrow \Phi$“).

Damit eine Logik praktisch anwendbar ist, muß sie sowohl einfach als auch ausdrucksfähig sein. Für den letzteren Punkt erreichte Lamport mit der TLA eine „Invarianz gegen Stottern“, auf die wir noch zurückkommen werden.

Bisher beschränkt sich Lamport auf abgeschlossene Systeme, da sie einfacher zu behandeln sind. (Notfalls muß die Umgebung als eine Komponente des Systems betrachtet werden.)

Die Grund-Bestandteile der TLA

Die TLA kombiniert zwei Logiken:

- Eine Logik der Aktionen
- Eine einfache temporale Logik

Die Definition der TLA

Die Semantik der Logik wird mit Hilfe von *Zuständen* definiert. (Diese sind nicht Teil der Logik selbst.) Ein Zustand ordnet einer Variablen einen Wert zu. Man schreibt:

Variable: `x` Bedeutung: $[\![x]\!]$ Wert für Zustand `s`: $s[\![x]\!]$

(Anmerkung: Man könnte genausogut auch sagen, daß es die Variable ist, die einem Zustand einen Wert zuordnet.)

Lamport führt auch sogenannte *starre Variablen* (englisch: rigid variables) ein, ein Programmierer würde sie „Konstanten“ nennen.

Eine *Zustandsfunktion* ist ein Ausdruck, der aus Variablen und Werten aufgebaut ist, zum Beispiel $x^2 + y - 3$.

Ihre Bedeutung ist eine Abbildung von Zuständen zu Werten:[12]

$$s[\![x^2 + y - 3]\!] \triangleq (s[\![x]\!])^2 + s[\![y]\!] - 3$$

Ein *Prädikat* ist eine boolwertige Zustandsfunktion, zum Beispiel: $x^2 = y - 3$

Eine *Aktion* ist ein boolwertiger Ausdruck aus Variablen, gestrichenen Variablen und Werten, zum Beispiel $x' + 1 = y$.

Ihre Bedeutung ist eine Relation zwischen (Vor- und Nach-)Zuständen:

$$s[\![x' + 1 = y]\!]\, t \triangleq t[\![x]\!] + 1 = s[\![y]\!]$$

Eine Aktion A beschreibt eine atomare Operation. Ein Paar von Zuständen s, t heißt *A-Schritt* genau dann, wenn $s[\![A]\!]\, t$ gleich dem Wahrheitswert `True` ist.

Das Prädikat *Enabled* A beschreibt, ob es zum aktuellen Zustand s einen Nachfolgezustand t gibt, so daß s, t ein A-Schritt ist. Oder mit anderen Worten, ob die Aktion A jetzt möglich ist.

Kommen wir nun zum Teil der temporalen Logik. Für Lamport liegt die Bedeutung eines Algorithmus′ in der Menge aller seiner möglichen Ausführungsfolgen. Dabei ist eine Ausführungsfolge eine Folge (im mathematischen Sinne) von Schritten, von denen jeder zu einem neuen Zustand führt. Um über Folgen von Zuständen schlußfolgern zu können, bietet sich eine temporale Logik an. Lamports temporale Logik besteht aus elementaren Formeln mit dem Immer-Operator „$\Box$“ und aus boolschen Operatoren. Die Semantikdefinition basiert auf unendlichen Zustandsfolgen. Die Bedeutung einer temporalen Formel ist eine Zusicherung über Zustandsfolgen.

Zuerst definiert Lamport eine sogenannte Roh-TLA (englisch: raw TLA). In ihr werden Formeln aufgebaut aus

- Aktionen und
- temporalen Operatoren

Die Bedeutung einer Aktion A für eine Zustandsfolge $\ll s_0, s_1, \ldots \gg$ ist:

$$\ll s_0, s_1, \ldots \gg [\![A]\!] \triangleq s_0 [\![A]\!]\, s_1$$

Aber leider hat die Roh-TLA einen Nachteil, sie ist *zu* ausdrucksfähig. Wenn wir die Formel $\Box(x' = x + 1)$ schreiben, dann muß x in *jedem* Schritt erhöht werden. Folglich wird die Geschwindigkeit des Systems durch die Granularität der Zeit bestimmt, und die Beschreibung ist nicht invariant gegen eine Veränderung der Granularität. Man kann keine Verfeinerung des Systems schreiben, die immer noch die

[12] Streng genommen ist ein Operator wie „`+`“ ein Wert, genauso wie „`2`“ und „`3`“, und die Notation `2+3` ist nur eine syntaktische Verschönerung des Tupels `+(2,3)`. Lamport setzt eine Evaluierungsfunktion `eval` voraus, die Tupel von Werten auf Werte abbildet. Zum Beispiel ist `eval(+,2,3)` gleich `5`. Die Funktion `eval` soll total sein, so daß selbst `eval(+,"abc",True)` ein Wert ist, auch wenn völlig offen bleibt, welcher Wert. Eine Zustandsfunktion ist entweder ein Wert, eine Variable oder ein Ausdruck der Form $f(f_1, \ldots, f_n)$, wobei f, f_1, ..., f_n Zustandsfunktionen sind. Lamport definiert $s[\![f(f_1, \ldots, f_n)]\!]$ als $\mathrm{eval}(s[\![f]\!], s[\![f_1]\!], \ldots, s[\![f_n]\!])$ für alle Zustände s. In Kapitel 5.1 werden wir auf die Thematik der Evaluierungsfunktion zurückkommen.

ursprüngliche Formel erfüllt. (Es handelt sich um das wohlbekannte Problem des Nächster-Zustand-Operators „$\circ$“ der temporalen Logik.)

Lamports Lösung besteht darin, daß er erstens in jedem Schritt erlaubt, daß gar nichts geschieht, und daß er zweitens Ausdrucksmittel für Liveness hinzufügt. Letzteres wird notwendig durch ersteres.

Es wird die folgende Notation definiert, wobei die Zustandsfunktion `f` gewöhnlich die relevanten Variablen des Systems beschreibt:

$$[A]_{\mathtt{f}} \triangleq A \vee (\mathtt{f}' = \mathtt{f})$$

Hiermit kann nun die (einfache) TLA definiert werden. Elementare Formeln sind nun (vergleiche mit der Roh-TLA oben):

- Prädikate und
- Formeln der Form $\Box[A]_{\mathtt{f}}$

Durch den erzwungenen Zusatz „$\vee\ (\mathtt{f}' = \mathtt{f})$“ erhält man eine Invarianz aller Formeln dagegen, daß vor Aktionen „Stotterschritte“ eingefügt werden, in denen gar nichts (mit den angegebenen Variablen) passiert.

Auch Liveness-Eigenschaften lassen sich bereits mit diesen Mitteln ausdrücken, denn es ist

$$\neg\Box[\neg A]_{\mathtt{f}} \equiv \Diamond(A \wedge (\mathtt{f}' \neq \mathtt{f}))$$

(Der Schließlich-Operator „$\Diamond$“ fordert, daß etwas schließlich zumindest einmal eintreten muß. In diesem Falle heißt dies, daß ein A-Schritt stattfinden muß, bei dem sich die Zustandsfunktion `f` tatsächlich verändert.)

Als Notation wird hierzu eingeführt:

$$<A>_{\mathtt{f}} \triangleq A \wedge (\mathtt{f}' \neq \mathtt{f})$$

Somit ist auch $\Diamond<A>_{\mathtt{f}}$ eine Formel der TLA, trotz der obigen Einschränkungen.

Nach Lamports Auffassung lassen sich Liveness-Eigenschaften verteilter Systeme durch Fairness-Eigenschaften ausdrücken. Es gibt nur zwei grundlegende Arten von Fairness:

- Die schwache Fairness (WF): ($\Box\Diamond$ ausgeführt) $\vee$ ($\Box\Diamond$ unmöglich)
  (Wenn etwas *lange* genug möglich ist, muß es schließlich geschehen.)
- Die starke Fairness (SF): ($\Box\Diamond$ ausgeführt) $\vee$ ($\Diamond\Box$ unmöglich)
  (Wenn etwas *oft* genug möglich ist, muß es schließlich geschehen.)

Formal definiert er:

$$\mathtt{WF}_{\mathtt{f}}(A) \triangleq (\Box\Diamond<A>_{\mathtt{f}}) \vee (\Box\Diamond\neg \mathit{Enabled}\ <A>_{\mathtt{f}})$$
$$\mathtt{SF}_{\mathtt{f}}(A) \triangleq (\Box\Diamond<A>_{\mathtt{f}}) \vee (\Diamond\Box\neg \mathit{Enabled}\ <A>_{\mathtt{f}})$$

Die TLA-Beschreibung eines Systems läßt sich immer in die folgende Grundform bringen:

$$\Phi \triangleq \mathtt{Init} \wedge \Box[N]_{\mathtt{f}} \wedge \mathtt{F}$$

Dabei spezifiziert das Prädikat `Init` die Anfangswerte der Variablen, N spezifiziert die Nächster-Zustand-Relation, `f` ein n-Tupel aller Programmvariablen und `F` eine Konjunktion aus $\mathtt{WF}_{\mathtt{f}}(A)$ und/oder $\mathtt{SF}_{\mathtt{f}}(A)$.

Die formale Definition der Syntax und Semantik der (einfachen) TLA findet sich in Abbildung 3-1; sie paßt, worauf Lamport Wert legt, vollständig auf eine einzige

Seite. Weiterhin benötigt eine Logik auch Schlußregeln, um Theoreme zu beweisen, sie finden sich in Abbildung 3-2. Dabei bilden die Regeln in der oberen Hälfte bereits ein relativ-vollständiges Schlußsystem, und damit es gut praktisch zu handhaben ist, hat Lamport die Regeln auf der unteren Hälfte der Seite hinzugefügt.

Zur Schlußregel STL1 ist allerdings ein Hinweis notwendig. Die Prämisse dort bedeutet, daß F eine aussagenlogische Tautologie ist, oder daß F mit Hilfe der Schlußregeln der Aussagenlogik (zum Beispiel des modus ponens) aus beweisbaren (TLA-)Formeln ableitbar ist.

In Abbildung 3-2 kommen gelegentlich Klammern „(: :)“ vor. Für den Moment können wir annehmen, daß sie einfach nicht da seien. Sie werden erst im Zusammenhang mit Verfeinerungen benötigt.

Verfeinerung

Im allgemeinen Fall konstruiert man eine Spezifikation Φ (mit einem höheren Abstraktionsgrad), indem man einige Hilfsvariablen $y_1, y_2, \ldots$ verwendet, und man konstruiert ein Programm Ψ (mit einem niedrigeren Abstraktionsgrad), indem man andere Hilfsvariablen $x_1, x_2, \ldots$ auf eine andere Weise verwendet. So kann man die Hilfsvariablen nicht direkt aufeinander abbilden. Um trotzdem zu zeigen, daß das Programm Ψ die Spezifikation Φ erfüllt, muß man beweisen:

$$(\exists x_1, x_2, \ldots : \Psi) \Rightarrow (\exists y_1, y_2, \ldots : \Phi)$$

Der Trick dafür ist, Zustandsfunktionen in Abhängigkeit von $(x_1, x_2, \ldots)$ zu finden, die die gesuchte Abbildung liefern. Um die obige Formel nachweisen zu können, beweist man

$$\Psi \Rightarrow (:\Phi:) \qquad .$$

Dabei bezeichnet $(:\Phi:)$ die Ersetzung $\Phi((:y_1:), \ldots / y_1, \ldots)$ für alle freien Vorkommen der Variablen $y_1, \ldots$ in Φ. Die verschiedenen „$(:y_i:)$“ wiederum sollen Zustandsfunktionen in Abhängigkeit von $(x_1, x_2, \ldots)$ sein.

Allerdings muß diese *Verfeinerungs-Abbildung* nicht unbedingt existieren. In diesem Falle muß man einige Dummy-Variablen an der richtigen Stelle einführen. Es wurde bewiesen, daß dann im Prinzip immer eine Verfeinerungs-Abbildung gefunden werden kann ([AbLa91]).

Um für Verfeinerungen mit der Existenz-Quantifikation arbeiten zu können, muß man die einfache TLA zuerst einmal um diese erweitern. Das geschieht in Abbildung 3-3.

Was Abbildung 3-2 betrifft, haben wir damit die Notation „(: :)“ in den Schlußregeln erklärt. Wenn man sie wie anfangs vorgeschlagen ignoriert, verwendet man einfach die Identität als Verfeinerungs-Abbildung.

Anmerkung: Lamport verwendet anstelle von (: :) einen Oberstrich über dem jeweiligen Ausdruck. Leider ist unser Textsystem nicht in der Lage, Oberstriche zu erzeugen (oder wenigstens Unterstreichungen, die echt unter tiefgestellten Indizes stehen). So haben wir die Notation ein wenig verändert. Aber dafür wird immerhin deutlicher, daß es sich bei (: :) um eine Funktion handelt.

Zusammenfassung der Eigenschaften der TLA

Die TLA basiert auf einfachen, grundlegenden Konstrukten der Mathematik. Die TLA kann Liveness ausdrücken, im Gegensatz zu (üblichen) Programmiersprachen. TLA-Formeln sind länger, da unveränderte Variablen jedesmal explizit genannt werden müssen, und da auch der Kontrollzustand („Programmzähler“) explizit an-

Syntax

$<\mathit{Formel}> \triangleq <\mathit{Prädikat}> \mid \Box[<\mathit{Aktion}>]_{<\mathtt{Zustandsfunktion}>} \mid \neg <\mathit{Formel}>$
$\mid <\mathit{Formel}> \wedge <\mathit{Formel}> \mid \Box <\mathit{Formel}>$

$<\mathit{Aktion}> \triangleq$ boolwertiger Ausdruck, der enthalten kann:
Konstanten, Variablen und gestrichene Variablen

$<\mathit{Prädikat}> \triangleq$ boolwertige $<\mathit{Zustandsfunktion}> \mid \mathit{Enabled}\ <\mathit{Aktion}>$

$<\mathit{Zustandsfunktion}> \triangleq$
Ausdruck, der enthalten kann:
Konstanten und Variablen

Semantik

$\mathtt{s}[\![\mathtt{f}]\!] \triangleq \mathtt{f}(\forall\text{"v"}: \mathtt{s}[\![\mathtt{v}]\!]/\mathtt{v})$

$\mathtt{s}[\![A]\!]\mathtt{t} \triangleq A(\forall\text{"v"}: \mathtt{s}[\![\mathtt{v}]\!]/\mathtt{v},\ \mathtt{t}[\![\mathtt{v}]\!]/\mathtt{v}')$

$\models A \triangleq \forall \mathtt{s},\mathtt{t} \in \mathbf{St}: \models \mathtt{s}[\![A]\!]\mathtt{t}$

$\sigma[\![\mathtt{F} \wedge \mathtt{G}]\!] \triangleq \sigma[\![\mathtt{F}]\!] \wedge \sigma[\![\mathtt{G}]\!]$

$\sigma[\![\neg\mathtt{F}]\!] \triangleq \neg\sigma[\![\mathtt{F}]\!]$

$\models \mathtt{F} \triangleq \forall \sigma \in \mathbf{St}^{\infty}: \models \sigma[\![\mathtt{F}]\!]$

$\mathtt{s}[\![\mathit{Enabled}\ A]\!] \triangleq \exists \mathtt{t} \in \mathbf{St}: \mathtt{s}[\![A]\!]\mathtt{t}$

$\ll \mathtt{s}_0,\mathtt{s}_1,\ldots \gg [\![\Box\mathtt{F}]\!] \triangleq \forall \mathtt{n} \in \mathbf{Nat}: \ll \mathtt{s}_n,\mathtt{s}_{n+1},\ldots \gg [\![\mathtt{F}]\!]$

$\ll \mathtt{s}_0,\mathtt{s}_1,\ldots \gg [\![A]\!] \triangleq \mathtt{s}_0[\![A]\!]\mathtt{s}_1$

zusätzliche Notation

$\mathtt{f}' \triangleq \mathtt{f}(\forall\text{"v"}: \mathtt{v}'/\mathtt{v})$

$[A]_{\mathtt{f}} \triangleq A \vee (\mathtt{f}' = \mathtt{f})$

$<A>_{\mathtt{f}} \triangleq A \wedge (\mathtt{f}' \neq \mathtt{f})$

$\mathit{Unchanged}\ \mathtt{f} \triangleq \mathtt{f}' = \mathtt{f}$

$\Diamond\mathtt{F} \triangleq \neg\Box\neg\mathtt{F}$

$\mathtt{F} \leadsto \mathtt{G} \triangleq \Box(\mathtt{F} \Rightarrow \Diamond\mathtt{G})$

$\mathtt{WF}_{\mathtt{f}}(A) \triangleq \Box\Diamond <A>_{\mathtt{f}} \vee \Box\Diamond\neg\mathit{Enabled}\ <A>_{\mathtt{f}}$

$\mathtt{SF}_{\mathtt{f}}(A) \triangleq \Box\Diamond <A>_{\mathtt{f}} \vee \Diamond\Box\neg\mathit{Enabled}\ <A>_{\mathtt{f}}$

wobei

$[\![\ ]\!]$ die zu definierende Bedeutungsfunktion ist,
$\mathtt{f}$ eine $<\mathit{Zustandsfunktion}>$ ist,
A eine $<\mathit{Aktion}>$ ist,
$\mathtt{F},\mathtt{G}$ jeweils $<\mathit{Formel}>$n sind,
$\mathtt{s},\mathtt{t},\mathtt{s}_0,\mathtt{s}_1,\ldots$ Zustände sind,
σ eine Zustandsfolge ist,
$\mathbf{St}$ die Menge aller Zustände ist und
$(\forall\text{"v"}: \ldots/\mathtt{v},\ \ldots/\mathtt{v}')$ die Ersetzung
für alle Variablen $\mathtt{v}$ bezeichnet.

Abbildung 3-1: Syntax und Semantik der einfachen TLA

Die Schlußregeln der einfachen temporalen Logik

STL1. $\dfrac{\text{F ist beweisbar durch Aussagenlogik}}{F}$

STL2. $\vdash \Box F \Rightarrow F$

STL3. $\vdash \Box\Box F \equiv \Box F$

STL4. $\dfrac{F \Rightarrow G}{\Box F \Rightarrow \Box G}$

STL5. $\vdash \Box(F \wedge G) \equiv (\Box F) \wedge (\Box G)$

STL6. $\vdash (\Diamond\Box F) \wedge (\Diamond\Box G) \equiv \Diamond\Box(F \wedge G)$

LATTICE.
$\succ$ ist eine Wohlordnung auf einer nichtleeren Menge S

$$\frac{F \wedge (c \in S) \Rightarrow (H_c \rightsquigarrow (G \vee (\exists d \in S\colon (c \succ d) \wedge H_d)))}{F \Rightarrow ((\exists c \in S\colon H_c) \rightsquigarrow G)}$$

Die grundlegenden Schlußregeln der TLA

TLA1. $\vdash \Box P \equiv P \wedge \Box[P \Rightarrow P']_P$

TLA2. $\dfrac{P \wedge [A]_f \Rightarrow Q \wedge [B]_g}{\Box P \wedge \Box[A]_f \Rightarrow \Box Q \wedge \Box[B]_g}$

Weitere Schlußregeln

INV1. $\dfrac{I \wedge [N]_f \Rightarrow I'}{I \wedge \Box[N]_f \Rightarrow \Box I}$

INV2. $\vdash \Box I \Rightarrow (\Box[N]_f \equiv \Box[N \wedge I \wedge I']_f)$

WF1.
$$\frac{\begin{array}{l} P \wedge [N]_f \Rightarrow (P' \vee Q') \\ P \wedge \langle N \wedge A\rangle_f \Rightarrow Q' \\ P \Rightarrow \mathit{Enabled}\ \langle A\rangle_f \end{array}}{\Box[N]_f \wedge WF_f(A) \Rightarrow (P \rightsquigarrow Q)}$$

WF2.
$$\frac{\begin{array}{l} \langle N \wedge B\rangle_f \Rightarrow \langle(:M:)\rangle_{(:g:)} \\ P \wedge P' \wedge \langle N \wedge A\rangle_f \Rightarrow B \\ P \wedge (:\mathit{Enabled}\ \langle M\rangle_g:) \Rightarrow \mathit{Enabled}\ \langle A\rangle_f \\ \Box[N \wedge \neg B]_f \wedge WF_f(A) \wedge \Box F \Rightarrow \Diamond\Box P \end{array}}{\Box[N]_f \wedge WF_f(A) \wedge \Box F \Rightarrow (:WF_g(M):)}$$

SF1.
$$\frac{\begin{array}{l} P \wedge [N]_f \Rightarrow (P' \vee Q') \\ P \wedge \langle N \wedge A\rangle_f \Rightarrow Q' \\ \Box P \wedge \Box[N]_f \wedge \Box F \Rightarrow \Diamond \mathit{Enabled}\ \langle A\rangle_f \end{array}}{\Box[N]_f \wedge SF_f(A) \wedge \Box F \Rightarrow (P \rightsquigarrow Q)}$$

SF2.
$$\frac{\begin{array}{l} \langle N \wedge B\rangle_f \Rightarrow \langle(:M:)\rangle_{(:g:)} \\ P \wedge P' \wedge \langle N \wedge A\rangle_f \Rightarrow B \\ P \wedge (:\mathit{Enabled}\ \langle M\rangle_g:) \Rightarrow \mathit{Enabled}\ \langle A\rangle_f \\ \Box[N \wedge \neg B]_f \wedge SF_f(A) \wedge \Box F \Rightarrow \Diamond\Box P \end{array}}{\Box[N]_f \wedge SF_f(A) \wedge \Box F \Rightarrow (:SF_g(M):)}$$

wobei

F, G, H_c	TLA-Formeln sind,
A, B, N, M	Aktionen sind,
P, Q, I	Prädikate sind und
f, g	Zustandsfunktionen sind.

Abbildung 3-2: Axiome und Schlußregeln der einfachen TLA

Syntax

$$
\begin{aligned}
&\langle \textit{allgemeine Formel} \rangle \triangleq \\
&\qquad \langle \textit{Formel} \rangle \mid \exists \langle \textit{Variable} \rangle : \langle \textit{allgemeine Formel} \rangle \\
&\qquad \mid \exists \langle \textit{starre Variable} \rangle : \langle \textit{allgemeine Formel} \rangle \\
&\qquad \mid \langle \textit{allgemeine Formel} \rangle \wedge \langle \textit{allgemeine Formel} \rangle \\
&\qquad \mid \neg \langle \textit{allgemeine Formel} \rangle \\
&\langle \textit{Formel} \rangle \triangleq \text{ eine Formel der einfachen TLA (siehe Abbildung 3-1)}
\end{aligned}
$$

Semantik

$$
\begin{aligned}
&\ll s_0, s_1, \ldots \gg =_x \ll t_0, t_1, \ldots \gg \triangleq \forall n \in \mathbf{Nat}: \forall \text{"v"} \neq \text{"x"}: s_n[\![v]\!] = t_n[\![v]\!] \\
&\natural \ll s_0, s_1, s_2, \ldots \gg \triangleq \mathbf{if}\ \forall n \in \mathbf{Nat}: s_n = s_0 \\
&\qquad \mathbf{then} \ll s_0, s_0, s_0, \ldots \gg \\
&\qquad \mathbf{else\ if}\ s_1 = s_0\ \mathbf{then} \natural \ll s_1, s_2, s_3, \ldots \gg \\
&\qquad\qquad \mathbf{else} \ll s_0 \gg \circ \natural \ll s_1, s_2, \ldots \gg \\
&\sigma[\![\exists x: F]\!] \triangleq \exists \rho, \tau \in \mathbf{St}^\infty: (\natural\sigma = \natural\rho) \wedge (\rho =_x \tau) \wedge \tau[\![F]\!] \\
&\sigma[\![\exists c: F]\!] \triangleq \exists c \in \mathbf{Val}: \sigma[\![F]\!]
\end{aligned}
$$

Schlußregeln

E1. $\vdash F(f/x) \Rightarrow \exists x: F$

E2.
$$\frac{F \Rightarrow G \quad \text{x kommt in G nicht frei vor}}{(\exists x: F) \Rightarrow G}$$

F1. $\vdash F(e/c) \Rightarrow \exists c: F$

F2.
$$\frac{F \Rightarrow G \quad \text{c kommt in G nicht frei vor}}{(\exists c: F) \Rightarrow G}$$

wobei

x	eine <*Variable*> ist,
f	eine <*Zustandsfunktion*> ist,
c	eine <*starre Variable*> ist,
e	ein Konstanten-Ausdruck ist,
Val	die Menge aller Werte ist,
F,G	jeweils <*allgemeine Formel*>n sind,
$s, s_0, t_0, s_1, t_1, \ldots$	Zustände sind,
$\mathbf{St}^\infty$	die Menge aller (unendlichen) Zustandsfolgen ist und
σ	eine Zustandsfolge ist.

Abbildung 3-3: Quantifikation in der TLA

gegeben werden muß. Ersteres wird dadurch gerechtfertigt, daß nur so eine einfache Logik möglich ist, die den Aufwand für Beweise klein hält, und letzteres ermöglicht eine differenzierte Ausdrucksfähigkeit von Parallelität bzw. Ordnung. (Abstrakte Algorithmen haben üblicherweise ohnehin wenige verschiedene Aktionen.)

Die TLA unterstützt die Entwicklung von einem sehr abstrakten Algorithmus bis zu einem detaillierten Programm (sofern man eine formale Semantik der Programmiersprache hat). Dies ist neu, und es wird möglich durch die Invarianz gegen Stotterschritte.

Die TLA ist geeignet, um Safety- und Liveness-Eigenschaften von diskreten Systemen zu spezifizieren und zu verifizieren. Die TLA kann *keine* Wahrscheinlichkeits-Eigenschaften ausdrücken. Aber Lamport hält diese nicht für nützlich, da es nicht hilft, wenn man nur weiß, daß das richtige Ergebnis lediglich möglich ist. Diese Art von Eigenschaften ist seiner Meinung nach nur ein Ersatz für Liveness, welche in der TLA ausdrückbar ist.

Gegenüber Pnuelis temporaler Logik ist neu, daß Aktionen als elementare Formeln verwendet werden. Dies ergibt wichtige Ausdrucksmöglichkeiten bei nur wenig zusätzlicher Komplexität.

Eine ausführlichere Erläuterung dessen, was in diesem Kapitel nur kurz angerissen werden kann, findet sich in dem TLA-Report [Lam91].

Weiterentwicklung zur TLA$^+$

Zur Zeit der Fertigstellung dieser Arbeit beschäftigt sich Lamport mit der Erweiterung der TLA zu einer „TLA$^+$“. [Lam91b] gibt den bis dahin erreichten Stand der Entwicklung wieder. Es fehlt noch vieles, zum Beispiel eine formale Definition der (Erweiterung zur) TLA$^+$, und nach Lamports Aussage ([Lam91a]) sind auch noch etliche Fehler enthalten. Soweit die TLA$^+$ für unsere Arbeit Nützliches oder Notwendiges enthält, werden wir in Kapitel 5.4 darauf zurückkommen.

Mechanisierung des Beweisens mit der TLA

Die Entwicklung der TLA und die Erforschung ihrer Anwendungsmöglichkeiten ist also zur Zeit unserer Arbeit noch in vollem Gange. Auf eine weitere aktuelle Aktivität sei besonders hingewiesen. In Kapitel 10.1 von [Lam91] schreibt Lamport (hier leicht gekürzt wiedergegeben):

Da die TLA eine einfache Logik ist, ist sie ideal zur Mechanisierung geeignet. Urban Engberg und Peter Grønning haben an der mechanischen Verifikation von TLA[13] gearbeitet. Dabei haben sie LP verwendet, ein Beweisersystem „von der Stange“, das auf Ersetzung (englisch: „rewriting“) basiert ([GaGu89][14]). Obgleich anfängliche Experimente zeigten, daß LP direkt verwendet werden kann, wurde entschieden, einen Präprozessor zu schreiben. Denn der Präprozessor erlaubt nicht nur lesbarere Spezifikationen, sondern er erlaubt darüberhinaus auch getrennte LP-Beweise für Aktionen-Formeln und temporale Formeln, wobei er einfachere Kodierungen für die Formeln benutzt, als sie bei einem einzigen Beweis möglich wären. Da das meiste Schlußfolgern in einem TLA-Beweis über Aktionen stattfindet, ist eine einfache Kodierung von Aktionen-Formeln wichtig.

Ein Beweis aus [Lam91], daß eine dortige Formel, die ein Beispielprogramm beschreibt, die Formel für ein zweites Beispielprogramm impliziert, wurde mit LP überprüft.

[13] Gemeint ist: Verifikation von TLA-Formeln

[14] Anmerkung: Diese Quelle beschreibt die Fähigkeiten des Larch Provers (LP). Die Sprache Larch, die er verwendet, ist zum Beispiel in [GuHo83] beschrieben.

Die Arbeit, TLA-Formeln mit LP mechanisch zu verifizieren, ist noch nicht abgeschlossen. Bisher wurden erst sehr einfache Beispiele in Angriff genommen. Der Präprozessor ist ein Prototyp, für den etwa zwei Mann-Monate Arbeit investiert wurden. Das Ziel dieses Projektes ist es, so schließt Lamport sein Kapitel, die Durchführbarkeit der Implementation eines Verifikationssystems einzuschätzen, das für reale Aufgabenstellungen brauchbar ist.

3.4 Prädikatentransformatoren

In diesem Kapitel wollen wir eine kurze Zusammenfassung der Semantikdefinitionstechnik der Prädikatentransformatoren nach Dikjstra geben, soweit wir sie für unsere Arbeit in späteren Kapiteln benötigen.

Wir verwenden Prädikatentransformatoren im wesentlichen, wie sie im kürzlich erschienenen Buch [DiSc90] von Dijkstra und Scholten definiert werden. Auf diesem Monographen basiert auch die Zusammenfassung dieses Kapitels. Im nächsten Kapitel werden wir dann noch einige neue, für unsere Zwecke nützliche Prädikatentransformatoren einführen.

Bevor wir aber die Prädikatentransformatoren als Semantikdefinitionstechnik einführen können, müssen wir das zugrunde liegende Konzept von Variablen vorstellen und ein wenig Notation vereinbaren, soweit sie in [DiSc90] den allgemein bekannten Rahmen überschreitet. Die Autoren gehen an einigen zentralen Stellen eigene Wege, und die weitere Argumentation wird nur verständlich, wenn man berücksichtigt, was anders definiert ist als üblich.

Variablen und der Zustandsraum

Die Autoren gehen von einem Zustandsraum voller anonymer Punkte aus. Jede boolsche Variable teilt diesen Raum in zwei Teile: Die Punkte, für die die Variable wahr ist, und die Punkte, für die sie nicht wahr ist. Zwei boolsche Variablen zusammen teilen den Zustandsraum in vier Teile. Daher kann man jede Variable als Dimension oder ≫Koordinatenachse≪[15] des Zustandsraumes ansehen. Wenn wir zum Beispiel annehmen, daß der Zustandsraum aus genau vier Punkten besteht und daß die beiden boolschen Variablen `X` und `Y` orthogonal sind, dann können wir die vier Punkte durch die vier Prädikate `¬X∧¬Y`, `¬X∧Y`, `X∧¬Y` und `X∧Y` beschreiben.

Im allgemeinen ist der Zustandsraum beliebig groß, so daß die obigen vier Prädikate den Zustandsraum vollständig in vier disjunkte Klassen aufteilen, die jeweils viele Punkte enthalten können. Wenn wir nicht nur boolsche Variablen, sondern zum Beispiel ganzzahlige Variablen verwenden, dann erhalten wir Koordinatenachsen, die den Raum in mehr als nur zwei Teile unterteilen. (Ein Prädikat dagegen bleibt weiterhin eine Zweiteilung des Raumes in die Punkte, für die es wahr ist, und in die Punkte, für die es nicht wahr ist.)

Ein Prädikatentransformator ist eine Funktion von Prädikaten nach Prädikaten, also von Punktklassen nach Punktklassen. Damit ist ein Prädikatentransformator eine Koordinatentransformation im Zustandsraum.

Beispiel: Der Prädikatentransformator `(n:=E)`, der den Text ≫`n`≪ durch den Text ≫`E`≪ ersetzt. Die Anwendung eines Prädikatentransformators `(X:=Y).(X∧Y)` ergibt das Prädikat `(Y∧Y)`, womit wir im obigen Beispiel die Klasse mit dem einen Punkt `X∧Y` in die Klasse mit den beiden Punkten `¬X∧Y` und `X∧Y` transformiert haben.

Etwas Notation

Es ist allgemein bekannt, wie man Verknüpfungen, zum Beispiel ≫`+`≪ oder ≫`-`≪, auf Strukturen von einfachen Komponenten verallgemeinern kann: Durch komponentenweise Anwendung.
Beispiel: `(a,b,c)+(d,e,f)` $\triangleq$ `(a+d,b+e,c+f)`

[15] In diesem und dem nächsten Kapitel werden wir ausnahmsweise die Zeichen ≫ ≪ als Anführungszeichen im laufenden Text verwenden, da die Zeichen „ “ in angeführten Texten zur Darstellung der Zuweisungsoperation ≫`"n:=E"`≪ benötigt werden.

Unüblich ist es allerdings, die »=«-Verknüpfung auf die gleiche Weise zu behandeln. Man definiert sie gewöhnlich so, daß sie auch für Strukturen nur einen einzigen boolschen Wert liefert:
`(1,2,3)=(1,2,4)` $\triangleq$ `False`

[DiSc90] definiert auch alle Verknüpfungen mit boolschen Werten für Strukturen komponentenweise, zum Beispiel »=«, »<«, »∧«, »⇒« usw. Im obigen Beispiel erhält man dann:
`(1,2,3)=(1,2,4)` $\triangleq$ `(True,True,False)`

Dies ist eine Struktur mit boolschen Werten, ein Konzept, welches für viele Leser sicherlich ungewohnt ist. Dafür soll es den Vorteil größerer Einheitlichkeit bieten.

Der aufmerksame Leser wird sicherlich schon bemerkt haben, daß unser Beispiel »`(1,2,3)=(1,2,4)`« ein Beispiel für ein Prädikat ist. Und in der Tat ist eine boolwertige Struktur nichts anderes als ein Prädikat.

Die Indexmenge einer (beliebigen) Struktur wird als zusätzliche Dimension zum bisherigen Zustandsraum betrachtet. »`(True,True,False)`« beschreibt also die folgende Punktklasse: In der ersten (Hyper-)Ebene enthält sie alle Punkte des bisherigen Zustandsraumes, ebenso in der zweiten Ebene, und in der dritten Ebene enthält sie keine Punkte.

Nachdem wir die »=«-Verknüpfung umdefiniert haben, fehlt uns allerdings eines: Wir möchten wissen, ob das obige Prädikat `(True,True,False)` in jeder seiner Komponenten wahr ist, also den gesamten neuen Zustandsraum umfaßt. Wir benötigen einen Operator, der uns immer einen der beiden boolschen Skalare `True` oder `False` liefert.

[DiSc90] definiert hierfür den »Überall«-Operator als Funktion von boolschen Strukturen nach boolschen Skalaren, dargestellt durch ein Paar eckige Klammern »`[ ]`« um den Operanden. Er liefert genau dann den Wert `True`, wenn alle Komponenten `True` sind.

Für Strukturen mit drei Komponenten:
`[(X,Y,Z)]` $\triangleq$ `[X]` $\wedge$ `[Y]` $\wedge$ `[Z]`

Beispiel: `[(True,True,False)] = False`

Da [DiSc90] die Indizierung der Komponenten eines Tupels als eine weitere Dimension des Zustandsraumes interpretiert, heißt »`[ ]`« nicht nur »überall in den Komponenten einer Struktur«, sondern ganz allgemein »überall in jedem Punkt des Zustandsraumes«. Und wenn unser Ausdruck eine Variable »`n`« vom Typ $\mathbb{N}$ enthält, dann ist eine der Koordinatenachsen des Zustandsraumes die Menge der natürlichen Zahlen. Damit der »Überall«-Operator den Wert `True` liefert, muß dann daher jede der von »`n`« abgeteilten Ebenen die gesamte zugehörige Ebene des Raumes umfassen, das heißt, das Prädikat muß für diese Ebene `True` sein. Beispiel: »$[n \leq n^2]$« ist `True`, da »$0 \leq 0$«, »$1 \leq 1$«, »$2 \leq 4$«, »$3 \leq 9$«, ... alle `True` sind. Also beinhaltet der »Überall«-Operator auch eine Allquantifizierung über die im Ausdruck enthaltenen Variablen, die damit zu »gebundenen« Variablen werden.

Anmerkung: In Kapitel 5 werden wir im Rahmen der TLA/PT den freien Variablen Werte zuordnen (die von der »Zeit« abhängig sein werden). Dies kann man auch verstehen als Auswahl eines bestimmten Punktes des Zustandsraumes. Variablen innerhalb des »Überall«-Operators werden wir dabei keinen Wert zuordnen, da sie nicht frei sind.

Eine weitere Notation wird in [DiSc90] im obigen Zusammenhang eingeführt: Die Verknüpfung »≡« für die Gleichheit von boolwertigen Operanden, auch »Äquivalenz« genannt. Damit wird das Problem der »Bindungsstärke« der »=«-Verknüpfung gelöst: »=« erhält eine höhere Bindungsstärke als die logischen Ver-

knüpfungen, »≡« die niedrigste. So läßt sich der Ausdruck »a<b ≡ c=d« klammerfrei schreiben.

Der wichtigste Grund für die Einführung von »≡« aber war, daß »≡« assoziativ ist, »=« dagegen nicht: [((X≡Y)≡Z) ≡ (X≡(Y≡Z))]

Nachdem wir die Notation von Prädikaten geklärt haben, ist noch die Schreibweise von Beweisen zu erläutern. [DiSc90] haben ein festes Schema entworfen, das Beweise recht übersichtlich macht.

Nehmen wir an, daß wir den boolschen Wert von [A] bestimmen wollen, welcher True sei. Dann können wir zum Beispiel in folgenden Schritten vorgehen:

Zuerst weisen wir [A] ≡ [B] nach,
dann weisen wir [B] ≡ [C] nach und
dann weisen wir [C] ≡ True nach.

Um die Ausdrücke [B] und [C] nicht doppelt zu schreiben, notieren wir den Beweis so:

```
  [A]
= { Hinweis, warum [A] ≡ [B] }
  [B]
= { Hinweis, warum [B] ≡ [C] }
  [C]
= { Hinweis, warum [C] ≡ True }
  True
```

Dabei wird stets mindestens eine ganze Zeile für die Begründung eines Schrittes reserviert.

Um Sätze der Form [A = D] zu beweisen, wandeln wir diesen Ausdruck nicht wie eben schrittweise in True um, sondern wir vereinfachen das Beweisformat noch weiter:

```
  A
= { Hinweis, warum [A = B] }
  B
= { Hinweis, warum [B = C] }
  C
= { Hinweis, warum [C = D] }
  D
```

Weiterhin lassen wir dies nicht nur für Ausdrücke der Form [A = D] zu, sondern auch für Ausdrücke der Form [A ⇒ D] und [A ⇐ D]. Entsprechend dürfen in diesen Fällen in der linken Spalte außer »=« auch »⇒« beziehungsweise »⇐« stehen. Letzteres Zeichen dürfte etwas ungewohnt sein, es ist aber gelegentlich äußerst praktisch, um einen Beweisschritt besser zu motivieren. Man macht nicht zuerst viele überraschende Umformungen, die erst am Ende ihren Sinn bekommen, sondern man kann den aufklärenden Schritt als erstes tun. Und formal macht dieses »auf den Kopf stellen« eines Beweises keine Probleme.

Schließlich müssen wir noch ein Wort zur Darstellung von Funktionen verlieren. [DiSc90] verwendet eine bestimmte Punkt-Notation, die ohne die sonst übliche Klammer-Schreibweise für die Argumente einer Funktion auskommt. Anstelle von »f(a)« schreiben die Autoren »f.a«. Für Funktionen mit mehreren Argumenten machen sie sich das Ergebnis von Curry zunutze, daß sich diese Funktionen stets in einstellige Funktionen umwandeln lassen, indem man höhere Funktionen einführt.

Beispiel: Statt »f(a,b)« schreiben wir »f.a.b«. Dabei ist »f.a« eine normale einstellige Funktion, die »b« als Argument bekommt, und »f« ist eine höhere Funktion,

die uns als Wert die vorige einfache Funktion liefert, wenn wir »f« auf das Argument »a« anwenden.

Prädikatentransformatoren

Nun können wir endlich zu den Prädikatentransformatoren kommen.

Prädikatentransformatoren dienen dazu, die Bedeutung von Zeichenketten zu definieren, welche Programme in einer bestimmten Sprache enthalten. Dabei ist der Ansatz nicht operational, sondern axiomatisch (siehe dazu auch Kapitel 3.1). Es wird ermöglicht, Eigenschaften der Operation abzuleiten, die einer Zeichenkette zugeordnet wird.

Wenn wir eine Operation (ein »Programm« oder ein Teil davon) entwerfen, beabsichtigen wir, mit dieser Operation etwas Bestimmtes zu erreichen. Dieses Ziel können wir durch ein Prädikat darstellen. Damit ein solches Ziel durch die Operation erreicht wird, müssen bestimmte Bedingungen vorher erfüllt sein, die wir ebenfalls durch Prädikate darstellen können. Von Details der Operation, die für unsere jeweiligen Zwecke unwichtig sind, möchten wir bei diesen Betrachtungen gern abstrahieren. Diese Beziehung zwischen dem »Vorher« und »Nachher« können wir durch Prädikatentransformatoren beschreiben.

Wir betrachten also Berechnungen (englisch: »computations«), die alle je einen *Anfangszustand* haben und die je einen *Endzustand* haben (können). Wir betrachten *keine Zwischenzustände*, da sie mit der Bedeutung der Operation nichts zu tun haben. (Andere, operationale Techniken zur Definition der Semantik von Operationen benötigen Zwischenzustände als Hilfsmittel der Darstellung.)

Um ein System durch einen Prädikatentransformator beschreiben zu können, muß dieses allerdings im wesentlichen ein transformationelles System sein (siehe dazu das Ende von Kapitel 3.1). Systeme, die niemals zu einem Ende kommen sollen, und (reaktive) Systeme, die auch noch »zwischendurch« Eingaben bekommen, lassen sich wegen der Abstraktion von den Zwischenzuständen auf diese Weise nicht gut beschreiben.

Bei den Prädikatentransformatoren abstrahieren wir nicht nur von den Zwischenzuständen, auch der (operationale) Begriff des »Zustandes« selbst tritt stark in den Hintergrund. Das Kalkül der Prädikatentransformatoren kommt völlig ohne die Erwähnung des Begriffs des Zustandes aus, er kommt lediglich bei der Anwendung der Prädikatentransformatoren auf die Definition der Bedeutung von Operationen ins Spiel, und zwar als Anfangs- und Endzustand.

Zu einer Operation gehört eine Menge von möglichen Anfangszuständen und eine Menge von möglichen Endzuständen, und die (terminierenden) Berechnungen der Operation sind bestimmt durch eine Relation zwischen diesen beiden Mengen. Da aber eine Beschreibung durch eine Relation laut [DiSc90] bisher wenig erfolgreich war, wird dieser Begriff vermieden. ([DiSc90] haben ebenso eine Abneigung gegen eine Mengen-Darstellung, weshalb die Autoren lieber von »Klassen« von Zuständen sprechen als von Mengen von Zuständen.)

Statt einer Relation zwischen zwei Mengen von Zuständen betrachten wir daher eine Funktion über Zustandsklassen. Funktionen lassen sich mathematisch wesentlich besser handhaben.

Wie oben bereits angedeutet, bezeichnen wir die Klassen von Zuständen auch als »Prädikate«. Ein Prädikatentransformator ist eine Funktion von Prädikaten nach Prädikaten.

Ein Prädikat X teilt den gesamten Zustandsraum in zwei Teile: Jeder Zustand gehört genau entweder zu der Klasse X oder zu der Klasse ¬X. Damit steht es uns frei, die

Berechnungen einer Operation S in drei Klassen zu teilen:

- »ewig«: Alle Berechnungen, die keinen Endzustand besitzen.
- »schließlich X«: Alle Berechnungen, deren Endzustand in der Klasse X liegt.
- »schließlich ¬X«: Alle Berechnungen, deren Endzustand in der Klasse ¬X liegt.

Weiterhin führen wir zu der Operation S die Funktionen »wp.S« und »wlp.S« von Prädikaten nach Prädikaten ein. Für alle Prädikate X soll gelten:

wp.S.X ist die Klasse aller Anfangszustände, in denen keine anderen als Berechnungen der Klasse »schließlich X« starten.

wlp.S.X ist die Klasse aller Anfangszustände, in denen keine Berechnungen der Klasse »schließlich ¬X« starten.

»wp« steht für »weakest precondition«, das heißt für die schwächste Vorbedingung. Damit ein Endzustand aus X sicher erreicht wird, muß der Anfangszustand der Berechnung mindestens das Prädikat wp.S.X erfüllen.

»wlp« steht für »weakest liberal precondition«, das heißt für die schwächste »freie« Vorbedingung. Alle Berechnungen, die in einem Zustand aus wlp.S.X starten, enden entweder in X, oder sie enden niemals.

»wp« und »wlp« selbst sind Funktionen von Zeichenketten nach Prädikatentransformatoren.

[DiSc90] weisen nach, daß die Relation zwischen den Anfangs- und Endzuständen und die Menge der nicht endenden Berechnungen durch die beiden Funktionen wp.S und wlp.S zusammen vollständig beschrieben sind, weshalb wp.S und wlp.S die Operation S vollständig beschreiben.

[DiSc90] kommt zwar völlig ohne graphische Darstellungen aus, aber wir hoffen, daß die Abbildung 3-4 die eben geschilderten Zusammenhänge deutlicher macht.

Wie diese Abbildung bereits nahelegt, beschränken wir uns auf Operationen, bei denen in jedem Anfangszustand mindestens eine Berechnung startet.

Wenn in einem Anfangszustand mehrere Berechnungen starten, dann ist die Operation nichtdeterministisch.

Anstelle daß wir die Berechnungen in die drei oben beschriebenen Klassen einteilen, können wir sie auch in die zwei folgenden einteilen:

- »anfänglich Y«: Alle Berechnungen, deren Anfangszustand in der Klasse Y liegt.
- »anfänglich ¬Y«: Alle Berechnungen, deren Anfangszustand in der Klasse ¬Y liegt.

Hiermit können wir zu der Operation S die Funktion »sp.S« von Prädikaten nach Prädikaten einführen. Für alle Prädikate Y soll gelten:

sp.S.Y ist die Klasse aller Endzustände, in denen mindestens eine Berechnung der Klasse »anfänglich Y« endet.

»sp« steht für »strongest postcondition«, das heißt für die stärkste Nachbedingung. Das Prädikat sp.S.Y gibt die Klasse aller Endzustände an, die von einem Anfangszustand in Y aus erreicht werden könnten.

Über Berechnungen, die keinen Endzustand haben, läßt sich mit sp keine Aussage machen, weshalb wir mit sp allein die Operation nicht vollständig beschreiben können. Und ein Gegenstück zu sp, wie es wp zu wlp ist, existiert hier nicht.

Um die stärkste Nachbedingung anschaulicher zu machen, haben wir Abbildung 3-5 entworfen.

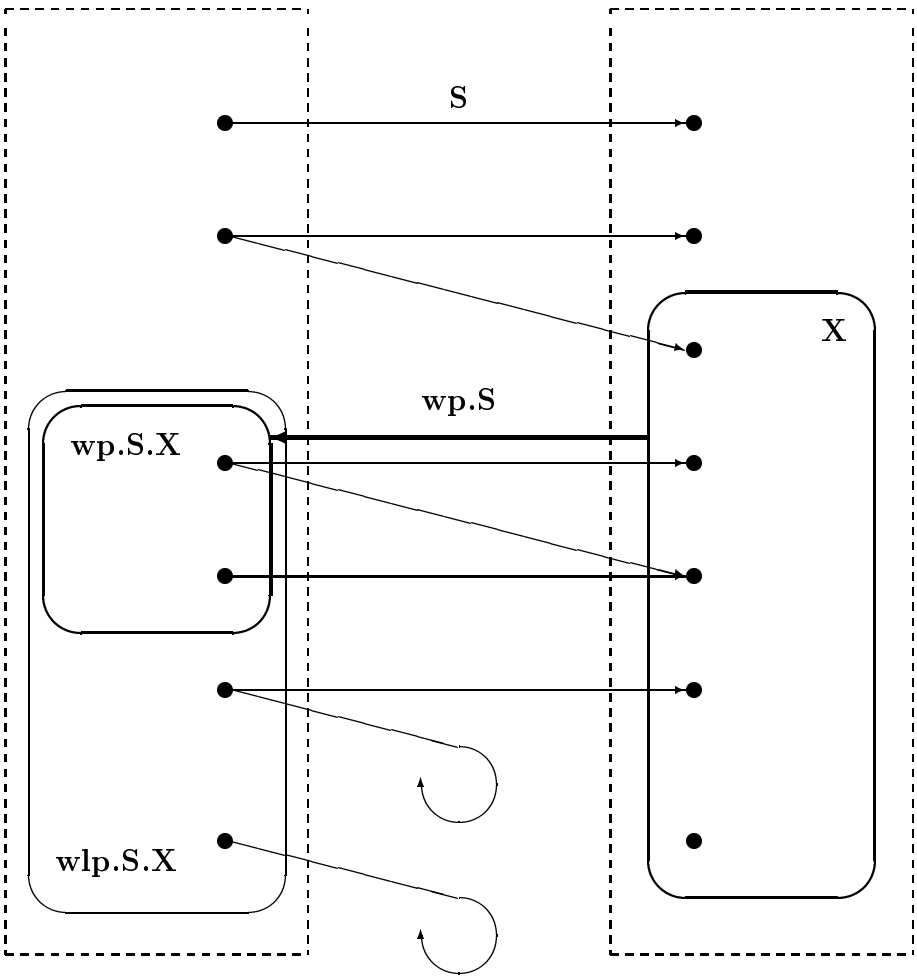

Abbildung 3-4: Veranschaulichung von `wp` und `wlp`

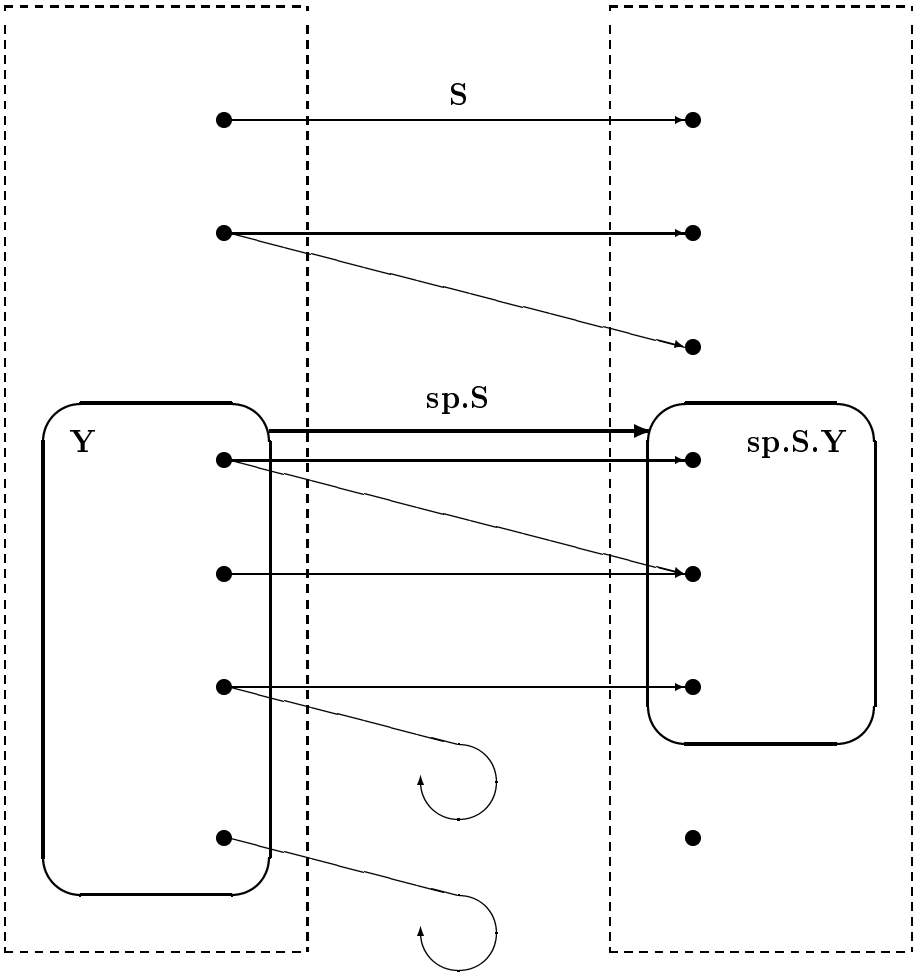

Abbildung 3-5: Veranschaulichung von `sp`

Damit zwei Prädikatentransformatoren `wp.S` und `wlp.S` sinnvoll als Definition der Bedeutung einer Operation angesehen werden können, müssen sie genau die zwei folgenden Bedingungen erfüllen:

- `wlp.S` muß vollständig konjunktiv (englisch: »universally conjunctive«) sein, das heißt, `wlp.S` muß »die Allquantifikation beliebig durchdringen«, und
- für `wp.S` muß `[wp.S.False ≡ False]` wahr sein.

Da keine Berechnung existiert, deren Endzustand das Prädikat »`False`« erfüllt, beschreibt »`wp.S.False`« die Klasse der Anfangszustände, in denen überhaupt keine Berechnung startet. Nach unserer obigen Annahme über Operationen muß diese Klasse leer sein, wie die zweite Bedingung ausdrückt.

Dijkstra nannte diese Eigenschaft einst (etwas irreführend) »das Gesetz vom ausgeschlossenen Wunder«.

Zwei der wichtigsten Eigenschaften der eben eingeführten Prädikatentransformatoren sollten wir noch erwähnen:

Jede Operation erfüllt hinterher das Prädikat »`True`«, oder sie endet nicht, so daß folgende Formel für alle `wlp.S` wahr ist:

`[wlp.S.True]`

»`wp.S.True`« beschreibt genau die Anfangszustände, deren sämtliche Berechnungen terminieren:

`[wp.S.X ≡ wp.S.True ∧ wlp.S.X]`

Schließlich wollen wir noch die Prädikatentransformatoren zu einigen wichtigen Grundoperationen angeben, wie sie in [DiSc90] stehen:

Havoc

Eine Operation, die immer terminiert, aber alle Programmvariablen nichtdeterministisch auf beliebige Werte setzt.

`[wlp.Havoc.X ≡ [X]]` für alle `X`
`[wp.Havoc.X ≡ [X]]` für alle `X`

`[sp.Havoc.Y ≡ ¬[¬Y]]` für alle `Y`

Abort

Eine Operation, die auf keinen Fall terminiert. (Bei [BrNe89] »`Loop`« genannt.)

`[wlp.Abort.X ≡ True]` für alle `X`
`[wp.Abort.X ≡ False]` für alle `X`

`[sp.Abort.Y ≡ False]` für alle `Y`

Skip

Eine Operation, die nichts verändert und immer terminiert.

`[wlp.Skip.X ≡ X]` für alle `X`
`[wp.Skip.X ≡ X]` für alle `X`

`[sp.Skip.Y ≡ Y]` für alle `Y`

"n:=E"

Eine Operation, die immer terminiert und alle Programmvariablen unverändert läßt außer »`n`«, welches den Wert »`E`« bekommt.

`[wlp."n:=E".X ≡ (n:=E).X]` für alle `X`
`[wp."n:=E".X ≡ (n:=E).X]` für alle `X`

Hierbei ist der Prädikatentransformator `(n:=E)` die Textersetzung, die den Text »`n`« durch den Text »`E`« ersetzt. Der Text »`E`« darf dabei den Text »`n`« wiederum enthalten.

Falls »`n`« nicht in »`E`« enthalten ist:

`[sp."n:=E".Y ≡ n = E ∧ (∃n:: Y)]` für alle `Y`

Falls »`n`« in »`En`« enthalten ist:
`[sp."n:=En".Y ≡`
`(∃y: n = (n:=y).En: (n:=y).Y)]` für alle `Y`
wobei `y` eine frische Variable ist.

"S0;S1"

Die bekannte sequentielle Komposition der Operationen `S0` und `S1`.

`[wlp."S0;S1".X ≡ wlp.S0.(wlp.S1.X)]` für alle `X`
`[wp."S0;S1".X ≡ wp.S0.(wp.S1.X)]` für alle `X`

`[sp."S0;S1".Y ≡ sp.S1.(sp.S0.Y)]` für alle `Y`

IF

Die Alternative:

If `B.0 -> S.0` ▯ `B.1 -> S.1` ▯ `...` ▯ `B.n -> S.n` **Fi**

Dazu sei:

`[BB ≡ (∃i:: B.i)]`

Diese Operation trifft eine nichtdeterministische Auswahl unter den erfüllten Bedingungen `B.i` und tut das, was die zugehörige Operation `S.i` tut. Gibt es keine erfüllte Bedingung, dann terminiert diese Operation nicht.

`[wlp.IF.X ≡ (∀i: B.i: wlp.(S.i).X)]` für alle `X`
`[wp.IF.X ≡ BB ∧ (∀i: B.i: wp.(S.i).X)]` für alle `X`

`[sp.IF.Y ≡ (∃i:: sp.(S.i).(B.i ∧ Y))]` für alle `Y`

(Man beachte, daß der »`(∀:: )`«-Operator als Wert selbstverständlich nicht einen boolschen Skalar, sondern eine boolsche Struktur liefert, hier also eine Klasse von Zuständen beschreibt, die der Durchschnitt einer Familie von Klassen von Zuständen ist. Im übrigen gilt
`[(∀x: A: B) ≡ (∀x:: A ⇒ B)]` und
`[(∃x: A: B) ≡ (∃x:: A ∧ B)]`)

DIF

Die deterministische Alternative, ein Spezialfall der Operation `IF`. Alle Bedingungen `B.i` müssen disjunkt sein.

`[wlp.DIF.X ≡ ¬BB ∨ (∃i: B.i: wlp.(S.i).X)]` für alle `X`
`[wp.DIF.X ≡ (∃i: B.i: wp.(S.i).X)]` für alle `X`

DO

Die allgemeine Iteration:

Do `B -> S` **Od**

Operational betrachtet, also mit Hilfe von Zwischenzuständen, tut diese Operation folgendes: Wenn die Bedingung `B` nicht erfüllt ist, verändert sie nichts und terminiert. Sonst beginnt sie die Operation `S`. Falls diese terminiert, beginnt das Spiel

von vorn.

Für alle X ist wlp.DO.X definiert als die *schwächste* Lösung von

$Y: [(B \vee X) \wedge (\neg B \vee wlp.S.Y) \equiv Y]$

Für alle X ist wp.DO.X definiert als die *stärkste* Lösung von

$Y: [(B \vee X) \wedge (\neg B \vee wp.S.Y) \equiv Y]$

Es läßt sich beweisen:

$[wlp.DO.X \equiv (\forall i: 0 \leq i: (wlp.IF)^i.(B \vee X))]$ für alle X

Für sogenannte »oder-kontinuierliche« wp.S läßt sich beweisen, daß für alle X gilt:

$[wp.DO.X \equiv$
$(\exists i: 0 \leq i: k^i.False)]$
mit k definiert durch:
$[k.Y \equiv (B \vee X) \wedge (\neg B \vee wp.S.Y)]$ (k ist abhängig von X!)

Für oder-kontinuierliche wlp.IF läßt sich beweisen, daß für alle X gilt:

$[wp.DO.X \equiv$
$(\exists i: 0 \leq i: (wlp.IF)^i.False) \wedge (\forall i: 0 \leq i: (wlp.IF)^i.(k.True))]$
mit demselben k wie eben.

Und schließlich gilt:

$[sp.DO.Y \equiv \neg B \wedge (\exists i: 0 \leq i: (sp.IF)^i.Y)]$ für alle Y

3.5 Einige neue, nützliche Prädikatentransformatoren

Wie zu Anfang des letzten Kapitels bereits angekündigt, werden wir nun einige neue Prädikatentransformatoren defininieren, die uns für unsere Arbeit nützlich sein werden.

Die meisten der bisher vorgestellten Prädikatentransformatoren waren wie der für die Zuweisungsoperation ≫`"n:=E"`≪ aufgebaut:

`[wlp."n:=E".X ≡ (n:=E).X]` für alle `X`

In dem Prädikat `X` wird ein bestimmter Text, hier ≫`n`≪, durch einen anderen Text, hier ≫`E`≪, ersetzt, und der Rest des Prädikates bleibt im wesentlichen unverändert.

Dies hat zur Folge, daß die Aussage über die Variable ≫`n`≪ geeignet verändert wird und daß Aussagen über andere Variablen unverändert bleiben. Damit wird spezifiziert, daß sich nur die Variable ≫`n`≪ durch die Operation verändert, während die anderen Variablen unverändert bleiben.

Nur ein Prädikatentransformator fiel bisher aus dem Rahmen: ≫`Havoc`≪

`[wlp.Havoc.X ≡ [X]]` für alle `X`

Mit ≫`Havoc`≪ beschreiben wir eine Operation, die alle Variablen auf beliebige Werte setzt.

Für unsere weitere Arbeit werden wir auch Prädikatentransformatoren benötigen, die zwischen diesen beiden Extremen liegen. Mit ihnen wollen wir spezifizieren, daß entweder eine bestimmte Menge von Variablen unverändert bleibt oder aber, daß sich die Variablen dieser Menge beliebig verändern dürfen. Zu diesem Zweck werden wir zwei neue Prädikatentransformatoren vorstellen.

Wir definieren:

```
[Change.{v1, v2, ..., vn}.X ≡
(∀v1:: (∀v2:: ...(∀vn:: X)...))]    für alle X
```

`Change` ist eine einstellige Funktion, die, wenn man sie auf eine Menge von Variablen `{v1, v2, ..., vn}` anwendet, einen Prädikatentransformator liefert, also eine Funktion von Prädikaten nach Prädikaten. (Falls einem die Schreibweise von Curry mit ausschließlich einstelligen Funktionen zu ungewohnt ist, kann man Change auch als zweistellige Funktion `Change({v1, v2, ..., vn}, X)` betrachten.)

`Change` beschreibt eine Operation, die den aufgeführten Variablen beliebige Werte zuweist, wie es `Havoc` tut, aber die die anderen Variablen unverändert läßt.

Indem wir über Variablen `v1, v2, ..., vn` allquantifizieren, fordern wir, daß das Prädikat keine Aussage zu diesen Variablen enthält. Beispiel: Habe eine Operation `"op"` einen solchen Prädikatentransformator als schwächste Vorbedingung:

`[wlp."op".X ≡ Change.{v1, v2}.X]` für alle `X`

Sobald in `X`, dem Prädikat über den Nachzustand, eine Aussage über die Variable `v1` gemacht wird, ergibt sich als schwächste Vorbedingung ≫`False`≪, weil die Operation nicht sicherstellt, daß die Aussage `X` hinterher erfüllt ist:

```
  wlp."op".(x=5 ∧ v1=7)
= { Einsetzen für die Definitionen von wlp."op" und Change }
  (∀v1:: (∀v2:: (x=5 ∧ v1=7)))
= { logische Umformung }
  False
```

Andere Variablen als `v1` und `v2` werden dagegen nicht beeinflußt:

```
  wlp."op".(x=5)
= { Einsetzen für die Definitionen von wlp."op" und Change }
  (∀v1:: (∀v2:: (x=5)))
= { logische Umformung }
  x=5
```

Als zweiten neuen Prädikatentransformator definieren wir:

```
[Same.{v1, v2, ..., vn}.X ≡
(∀z1:: (∀z2:: ...(∀zj:: X)...))]    für alle X
```

Dabei ist die Menge {z1, z2, ..., zj} genau die Menge derjenigen Variablen, die frei im Prädikat X vorkommen, aber nicht Element der Menge {v1, v2, ..., vn} sind. (Man kann die Definition auch variieren und zulassen, daß auch über weitere Variablen quantifiziert wird, solange nur nicht über v1, v2, ..., vn quantifiziert wird.)

Same beschreibt eine Operation, die die aufgeführten Variablen unverändert läßt, aber allen anderen beliebige Werte zuweist, wie es Havoc tut.

Im Vorgriff auf Kapitel 5, in dem wir Prädikatentransformatoren in die TLA integrieren werden, sei vielleicht schon angemerkt, daß mit dem Prädikatentransformator Same.{v1, v2, ..., vn} dieselbe Operation beschrieben wird wie durch die TLA-Aktion *Unchanged* (v1, v2, ..., vn), die definiert ist als (v1′, v2′, ..., vn′) = (v1, v2, ..., vn). Falls einem unsere Definition von Same zu unhandlich sein sollte, insbesondere die Bestimmung der Variablen, über die quantifiziert werden soll, und auch die unter Umständen zahlreichen Quantifikationen, dann kann man im Anschluß an Kapitel 5 den Prädikatentransformator Same auch auf der Basis der Aktion *Unchanged* definieren.

Und noch eine Anmerkung zu den freien Variablen eines Ausdrucks: Wir definieren sie (wie allgemein üblich) als genau die Variablen, die im Ausdruck textuell vorkommen, abzüglich derjenigen, die durch eine Quantifikation gebunden sind.

Change und Same verhalten sich in einem gewissen Sinne komplementär zueinander. Es gilt:

Falls das Prädikat X keine anderen Variablen als v1, ..., vn, z1, ..., zn enthält, dann gilt (und *nur* dann):

```
[Change.{v1, v2, ..., vn}.X ≡ Same.{z1, z2, ...,zn}.X]
für dieses X
```

Der Beweis folgt unmittelbar aus den Definitionen. Diese Eigenschaft werden wir nicht weiter ausnutzen, aber wir haben sie angeführt, weil wir hoffen, daß sie das Verständis der beiden Prädikatentransformatoren verbessert.

4. Wahl einer Semantikdefinitionstechnik für Estelle

Bei Beginn dieser Diplomarbeit war vorgesehen, Estelle-Transitionen durch prädikatenlogische Zusicherungen darzustellen. Da dies, wie wir in Kapitel 4.5 noch sehen werden, nicht durchführbar ist, wurde es notwendig, die Möglichkeiten für die Darstellung der Semantik von Estelle-Transitionen umfassend zu untersuchen.

In Kapitel 4.1 wird die Entscheidung für die grundsätzliche Methode zur Definition der Semantik gefällt, in Kapitel 4.2 werden die Bedingungen an eine geeignete Semantikdefinitionstechnik untersucht und in Kapitel 4.3 werden die verfügbaren Techniken zusammengetragen. In Kapitel 4.4 wird ein einfacher Algorithmus als Beispiel vorgestellt, anhand dessen sie in Kapitel 4.5 auf ihre Eignung geprüft werden, und in Kapitel 4.6 schließlich werden die Ergebnisse zusammengefaßt.

4.1 Auswahl einer grundsätzlichen Semantikdefinitionsmethode für Estelle

Im Grundlagen-Kapitel 3.1 wurden die verschiedenen grundsätzlichen Methoden zur Semantikdefinition vorgestellt: Übersetzersemantik, operationale Semantik, denotationale Semantik, axiomatische Semantik. (Weiterhin wurde zwischen transformationellen und reaktiven Systemen differenziert.) Nun treffen wir die Entscheidung, welches Beschreibungsprinzip wir für unsere Arbeit zugrunde legen wollen.

Im ISO-Standard [ISO89a] wird Estelle beschrieben durch globale Zustände und Zustandsübergänge zwischen diesen, also eindeutig operational.

Bei der Definition der Semantik von Programmen, die eine Eingabe bekommen, dann eine Verarbeitung ausführen und schließlich eine Ausgabe machen, also von transformationellen Programmen, hat der operationale Ansatz die unangenehme Eigenschaft, Zwischenzustände in der Beschreibung zu betonen, obwohl nur der erste und der letzte Zustand für die Gesamtbedeutung von Interesse sind. Denotationale und axiomatische Methoden sind dort zur Definition der Gesamtbedeutung von Programmen besser geeignet.

Bei einer Sprache wie Estelle ist dies anders. Mit Estelle spezifiziert man vorzugsweise ein System von interagierenden Individuen, welches im allgemeinen nicht terminiert. Da es kein Endergebnis gibt, sind das Wesentliche die einzelnen Schritte des Vorgangs, also die Kommunikationsereignisse und Zustandsänderungen.

Anmerkung: Obwohl ein gesamtes Estelle-System im allgemeinen nicht terminiert, paßt eine Klassifizierung als reaktives System nicht so ganz, da es keine Interaktion mit der Umgebung gibt. Nur die einzelnen Teil-Systeme kommunizieren miteinander und können gut als reaktiv bezeichnet werden.

Wie auch immer, es müssen die einzelnen Schritte des Gesamtsystems ausgedrückt werden können, und somit ist grundsätzlich eine operationale Beschreibung angemessen. Ein solches System ließe sich als Gesamtheit denotational oder axiomatisch gar nicht beschreiben.

Trotz dieses grundsätzlich operationalen Ansatzes bleibt es uns aber unbenommen, die seit dessen Entwicklung entstandenen neueren Ansätze, insbesondere des axiomatischen, im Auge zu behalten und ihre Vorteile soweit wie möglich zu nutzen. Wie in der Einleitung bereits begründet, soll unser formales Modell insbesondere die Ve-

rifikation unterstützen, wozu wie in Kapitel 3.1 gesehen die axiomatische Semantik besonders geeignet ist. Laut desselben Kapitels ist eine durchgängig operationale Beschreibungsform, wie zum Beispiel die im ISO-Standard, für die Verifikation kaum geeignet.

Nach der Entscheidung für einen Grundansatz zur Semantikdefinition bleiben noch zwei weitere wichtige Grundfragen zu klären, eine zur Nebenläufigkeit und eine zur Behandlung der Zeit.

Bei der Beschreibung der Semantik von Estelle müssen wir die Konzepte des Nichtdeterminismus′ und der Nebenläufigkeit bedenken. In Estelle ist zum Beispiel die Auswahlentscheidung für eine von mehreren schaltbereiten Estelle-Transitionen nichtdeterministisch, sie ist nicht von der in Estelle geschriebenen Spezifikation festgelegt. Bei der Beschreibung von Nichtdeterminismus hat der operationale Ansatz keine Schwierigkeiten, anstelle einer Zustandsübergangsfunktion muß nur eine Zustandsübergangsrelation gesetzt werden. Auch die axiomatische Methode kann Nichtdeterminismus mit Hilfe des logischen Oders „$\vee$“ sehr einfach ausdrücken.

Das Konzept der Nebenläufigkeit ist allgemeiner. Bei ihr geht es um die kausale Unabhängigkeit von Ereignissen. Wenn man nicht nur daran interessiert ist, auf welche Weise Ereignisse zeitlich angeordnet sein können, sondern wenn man sich insbesondere dafür interessiert, welche Ereignisse durch eine Ursache-Wirkung-Beziehung voneinander abhängen und welche dies nicht tun, dann muß man zwischen Nebenläufigkeit und lediglich nichtdeterministischer zeitlicher Anordnung unterscheiden.

Um dies zu tun, kann man Teilordnungen von Ereignissen betrachten, die die kausalen Abhängigkeiten ausdrücken (zum Beispiel [Gai88], [CaFrMo82], [BoCa86], [BoCa88], siehe auch [Win88]). Durch Einfügen zusätzlicher, nichtkausaler Abhängigkeiten ist es anschließend möglich, wieder zu einer (zum Beispiel) totalen Ordnung von Ereignissen zurückzukehren, die dann die zeitliche Anordnung wiederspiegelt. Unter anderem [BoCa86] (und [BoCa88]) analysiert die Semantik von Nebenläufigkeit. Transitionssysteme werden dort so erweitert, daß sie nicht mehr (total geordnete) Ausführungsfolgen beschreiben, sondern Teilordnungen von Aktionen. Damit wird es möglich, zwischen Nichtdeterminismus und Nebenläufigkeit zu unterscheiden. Milners Expansionstheorem „`(a | b)=(ab + ba)`“, das die Nebenläufigkeit von zwei Aktionen als die nichtdeterministische Entscheidung zwischen den zwei möglichen sequentiellen Abfolgen erklärt, gilt dann nicht mehr.

Mit diesem Konzept folgt aus Kausalität die zeitliche Ordnung, aber aus zeitlicher Ordnung folgt noch nicht Kausalität.

Es stellt sich die Frage, ob wir bei der Beschreibung der Semantik von Estelle zwischen Nebenläufigkeit und Nichtdeterminismus unterscheiden sollen.

Das Hauptziel unseres Semantikdefinitionsansatzes ist es (siehe Kapitel 1), Aussagen über die Eigenschaften von Systemen formal ableiten zu können, die in Estelle spezifiziert sind. Hierbei handelt es sich insbesondere um die Eigenschaften des Verhaltens des Systems, also dessen, was beobachtbar ist. Das Konzept der Kausalität geht darüber hinaus und fragt nach dem Warum des Beobachtbaren. Hierüber macht eine Estelle-Spezifikation keine Aussagen mehr, nur über das Was. Daher steht es uns frei, nur die zeitliche Ordnung von Aktionen zu betrachten. Dies vereinfacht im übrigen auch die Beschreibung, da dann gewöhnliche Folgen (Ausführungspfade) zur Beschreibung des Systems ausreichen. Anderenfalls wären Teilordnungen notwendig.[16], [17]

[16] In [DaDe90] wird allerdings auch ein Ansatz vorgestellt, um mit markierten Folgen (genauer: Bäumen) von Ereignissen Kausalität ausdrücken zu können.

[17]

Als nächstes folgt die Entscheidung, wie wir mit „gleichzeitig“ schaltenden Estelle-Transitionen umgehen. Wenn zwei Estelle-Transitionen verschiedenen Systemprozessen angehören, dann sollen sie völlig unabhängig voneinander schalten können. Hier wünschen wir uns intuitiv, daß sie auch parallel schalten können, also im selben Zustandsübergang des abstrakten Automaten (bei operationaler Semantik, siehe dazu Kapitel 3.2). Es zeigt sich allerdings, daß dies für unsere Zwecke nicht sinnvoll ist.

Solange eine Estelle-Transition nur den lokalen Zustand ihrer eigenen Modulinstanz verändert, sind mehrere solcher Veränderungen leicht zu einem gemeinsamen, globalen Zustandsübergang zusammenzufassen. Aber eine Estelle-Transition kann auch nichtlokale Effekte haben, indem sie Nachrichten überträgt. Dann wird es recht aufwendig zu formulieren, wie beispielsweise eine Estelle-Transition aus einer Nachrichten-Warteschlange das erste Element entnimmt, wobei im selben Schritt die Möglichkeit besteht, daß eine andere Estelle-Transition hinten eine neue Nachricht anfügt. Noch aufwendiger wird das parallele, atomare Schalten von Estelle-Transitionen durch das Konzept der gemeinsamen Warteschlange (**Common Queue**) für mehrere Interaktionspunkte einer Modulinstanz. Denn hier können mehrere Estelle-Transitionen aus verschiedenen anderen Modulinstanzen im selben Zustandsübergang Nachrichten übertragen, die alle in dieselbe gemeinsame Warteschlange eingereiht werden müssen. Da die Nachrichten nach dieser Aktion aber eine Reihenfolge besitzen müssen, muß ein explizites Verfahren vorgesehen werden, um diese Nachrichten zu sequentialisieren.

Dies alles würde sehr viel Aufwand bei der Darstellung erfordern. Ein Hauptziel unserer Formalisierung soll aber das Ermöglichen der Verifikation sein, und für die praktische Durchführbarkeit der Verifikation muß der verwendete Formalismus so einfach wie nur möglich sein.

So erscheint es dringend angezeigt, die Modellierung derart zu konstruieren, daß in einem Zustandsübergang des Gesamtsystems immer höchstens eine Estelle-Transition schaltet.

Schließlich bedeutet dies keine praktische Einschränkung der Ausdruckskraft, denn man kann nicht-synchronisierten physikalischen Vorgängen (einer Implementation) bei einer genügend hohen Zeitauflösung immer verschiedene Zeitpunkte zuordnen. Bei den oben zitierten Verfahren, die auf (kausalen) Teilordnungen aufbauen, entspricht dies dem dort immer möglichen Einfügen weiterer Präzedenzen zwischen den Ereignissen, um zu einer totalen (zeitlichen) Ordnung der Ereignisse zu gelangen.

Mit der Festlegung, daß pro Zustandsübergang höchstens eine Estelle-Transition schaltet, haben wir eine *„Interleaving“-Semantik* gewählt. (Wir verwenden hier den englischen Begriff, weil uns die deutschen Begriffe wie „Verzahnungs“-Semantik oder „Verschränkungs“-Semantik aufgrund von vielen Nebenbedeutungen zu undeutlich sind.)

Der Vergleich mit dem ISO-Standard, Kapitel 5.3.4, zeigt, daß auch dort eine Interleaving-Semantik gewählt wird. Allerdings geschieht dies dort ohne Begründung, abgesehen von einer Rechtfertigung, daß zwei zum Schalten endgültig ausgewählte Estelle-Transitionen sich gegenseitig nicht mehr am Schalten hindern können. (Zum Beispiel verhindert die Vater-Sohn-Vorrangregelung zwischen Modulinstanzen, daß

[KaPe88] schlägt vor, bei der Verifikation nicht Mengen von Folgen zu verwenden, sondern Teilordnungen, um die kombinatorische Explosion der Anordnungsmöglichkeiten von Ereignissen besser einzudämmen. Da das in unserer Arbeit schließlich gewählte Verfahren ebenfalls erlaubt, von „gerade uninteressanten“ Ereignissen während der Verifikation zu abstrahieren, und da unklar ist, ob und falls ja mit welchem Aufwand der in [KaPe88] skizzierte Ansatz mit dem in dieser Arbeit vorgestellten Ansatz zu vereinigen ist, haben wir diese Richtung nicht weiter verfolgt.

die Modulinstanz, zu der die zweite Transition gehört, von der ersten Transition vernichtet wird.)

Schließlich ist noch eine weitere Grundfrage zu klären, die Entscheidung über die Darstellung der quantitativen Zeitaspekte von Estelle. Die **Delay**-Klauseln von Estelle-Transitionen erlauben die Angabe von Verzögerungszeitintervallen. Hierfür müssen wir über konkrete Zeitwerte reden können. Der ISO-Standard (Kapitel 5.3.5) nimmt an, daß es einen von der Spezifikation unabhängigen „Zeit-Prozeß“ gibt, der die verbleibenden „Verzögerungszeiten“ für die Estelle-Transitionen beeinflußt. Über diesen Zeitprozeß macht der ISO-Standard genau zwei Annahmen:

- Die Zeit schreitet voran, wenn die Berechnung dies tut, und
- dieser Fortschritt ist für die Verzögerungszeiten von allen beteiligten Estelle-Transitionen gleichmäßig.

Wir sehen keinen Grund, dies anders zu modellieren, und werden folglich ebenfalls eine abstrakte „globale Uhr“ spezifizieren. Auf diese soll die Estelle-Spezifikation keinen Zugriff haben, nur Zeitintervalle sollen mit der Uhr für alle Systemkomponenten gleichmäßig gemessen werden können.

Auf den ersten Blick scheinen sich das Konzept einer globalen Uhr und das Konzept eines verteilten Systems, wie es durch eine Estelle-Spezifikation typischerweise beschrieben wird, zu widersprechen. Denn ein verteiltes System ist ganz wesentlich dadurch charakterisiert, daß den Komponenten der globale Gesamtzustand nicht bekannt ist.

Aber da die Komponenten auf unsere angenommene globale Uhr keinen Zugriff haben, nur auf Zeitdifferenzen, ist eine verteilte Implementation leicht möglich. Sie kann viele verteilte lokale Uhren vorsehen, die verschiedene Zeiten anzeigen können, aber genügend genau gleichschnell gehen. Damit reduziert sich die globale Uhr zu einer Abbildung der universellen Zeit in der Newtonschen Physik[18]. Es ist garantiert, daß die Zeit überall gleichschnell abläuft, aber ein „Nullpunkt der Zeit“ ist nicht definiert, nach dem man alle Uhren stellen könnte.

[18] Einsteins Relativitätstheorie gibt die Annahme einer universellen Zeit auf. Relativ zueinander beschleunigte Uhren laufen verschieden schnell.

4.2 Welche Beweismethoden müssen unterstützt werden

Im vorigen Kapitel haben wir uns entschieden, eine Estelle-Spezifikation im Grundsatz operational darzustellen, also durch Angabe eines abstrakten Automaten beziehungsweise Transitionssystems. (Siehe dazu auch Kapitel 3.2.) Damit haben wir aber noch nicht festgelegt, wie wir dabei die Zustandsübergangsrelation darstellen wollen. In diesem Kapitel werden wir die Forderungen aufstellen, die eine Beschreibungstechnik für die Zustandsübergangsrelation erfüllen muß. Im nächsten Kapitel werden wir dann die vorhandenen Techniken vorstellen und eine davon auswählen. Da die Zustandsübergangsrelation zu einem wesentlichen Teil bestimmt wird durch die Estelle-Transitionen, wird ihnen unsere Aufmerksamkeit hauptsächlich gelten.

Wie bereits in der Einleitung begründet, soll die Formalisierung insbesondere dazu geeignet sein, die Verifikation von Estelle-Spezifikationen zu unterstützen. Um Korrektheitsbeweise zu führen, stehen uns für Transitionssysteme im wesentlichen dieselben Grundverfahren zur Verfügung wie für die allgemeine **do..od**-Schleife der sequentiellen Programmierung (Abbildung 4-1).

```
do
  Bedingung1 -> Anweisung1
[]
  Bedingung2 -> Anweisung2
[]
  ...
[]
  Bedingungn -> Anweisungn
od
```

Abbildung 4-1: Die allgemeine **do..od**-Schleife der sequentiellen Programmierung

Denn diese kann man auch als Transitionssystem auffassen: Es ist immer höchstens einer der alternativen Zweige in Bearbeitung, so daß man die Bearbeitung eines Zweiges als atomar ansehen kann (sofern er mit Sicherheit terminiert). Die Auswertung der Bedingungen und die Auswahl eines der Zweige kann als Auswahl einer Transition betrachtet werden. Der Systemzustand zu den Zeitpunkten, zu denen die Auswahl stattfindet, kann auf den Zustand eines Transitionssystems zwischen dem atomaren Schalten von Transitionen abgebildet werden.

Um die vollständige Korrektheit einer **do..od**-Schleife zu zeigen, muß man einerseits ihre partielle Korrektheit und andererseits ihre Termination zeigen.

Der Begriff der Termination der **do..od**-Schleife und die zugehörigen Verifikationsmethoden helfen allerdings bei der Untersuchung von Transitionssystemen nicht sehr weit, da bei diesen häufig weder gewünscht noch vorgesehen ist, daß sie terminieren. Und ohne Termination hilft uns auch die partielle Korrektheit nicht, da ein nicht terminierendes System per Definition immer partiell korrekt ist. Daher müssen diese beiden Begriffe sinnvoll verallgemeinert werden.

In der sequentiellen Programmierung bedeutet partielle Korrektheit, daß „nichts Unerwünschtes herauskommt“. Die zugehörige Verallgemeinerung ist daher, daß „nichts Unerwünschtes geschieht“. Solche Eigenschaften werden auch als *Safety-Eigenschaften* bezeichnet.

Termination heißt, daß „das Programmende mit Sicherheit eintritt“. Verallgemeinert heißt dies, daß „etwas Gewünschtes mit Sicherheit eintritt“. Diese Eigenschaften heißen *Liveness-Eigenschaften*.

Wie wir gleich sehen werden, lassen sich nicht nur die beiden Begriffe, sondern ebenso auch die zugehörigen Untersuchungsmethoden verallgemeinern.

Lamport hat in [Lam91] mit seiner „temporalen Logik der Aktionen“ (TLA) die Untersuchungsmethoden für die Korrektheit formalisiert, und da wir auf diesen Formalismus noch ausführlich zurückkommen werden, ist eine Begriffsklärung notwendig. Lamport verwendet nicht den Begriff der Safety, sondern den Begriff der *Invarianz*-Eigenschaft. Dabei meint er nicht allgemein invariante Eigenschaften, da diese auch Liveness-Eigenschaften umfassen könnten, sondern invariant geltende Safety-Eigenschaften. Safety-Eigenschaften, die nur zu Beginn gelten, sind in seinem Formalismus ebenfalls ausdrückbar, aber sie fallen nicht unter den Begriff der Invarianz-Eigenschaften.

Ebenso verwendet er statt des Begriffs der Liveness (formal beschrieben durch Formeln der Art „$\Diamond$F“) den Begriff der *Unvermeidlichkeits*-Eigenschaft (englisch: „eventuality-property“), mit dem sich die seiner Meinung nach wichtigen Liveness-Eigenschaften direkt ausdrücken lassen. Der dafür verwendete „leads-to“-Operator $\leadsto$ ist formal definiert durch „$F \leadsto G \triangleq \Box(F \Rightarrow \Diamond G)$“, wodurch sich Liveness-Eigenschaften, die nur zu Beginn gelten, nur umständlich ausdrücken lassen. Letzlich lassen sich aber alle Liveness-Eigenschaften auch auf diese Weise ausdrücken.

Betrachten wir nun, welche Verifikationsverfahren es gibt und wie sie unterstützt werden müssen.

Für Beweise von Safety-Eigenschaften steht uns das folgende grundlegende Verfahren zur Verfügung:

- Es wird eine geeignete globale Invariante des Transitionssystems bestimmt,
- es wird gezeigt, daß diese Zusicherung zum Systemstart gilt,
- es wird gezeigt, daß jede einzelne Transition die Gültigkeit dieser Zusicherung stets erhält, und
- schließlich wird nachgewiesen, daß aus der Zusicherung die gewünschten Eigenschaften folgen.

Wenn die gewünschten Eigenschaften sehr umfangreich und auch kompliziert sind, wird es sehr schwer, die globale Invariante zu finden. Da es ohnehin empfehlenswert ist, die gewünschten Eigenschaften zu strukturieren, kann man sich in diesem Falle damit behelfen, das obige Verfahren jeweils für Teileigenschaften durchzuführen, aus deren Konjunktion sich dann die gesamten gewünschten Eigenschaften ergeben. Auf diese Weise sind dann viele, aber wieder einfachere globale Invarianten des Transitionssystems zu finden und zu beweisen.

Selbstverständlich kann man dieses Grundverfahren noch verfeinern, indem man zum Beispiel in geeigneter Weise von vornherein ausschließt, daß durch bestimmte Gruppen von Transitionen die Gültigkeit der Zusicherung verändert wird, weil diese „in den betreffenden Fällen nicht aktiviert sind“. So kann die zu leistende Arbeit erheblich reduziert werden.[19]

[19] Auch in [Bre90] wurden die hier beschriebenen Verfahren im Prinzip angewandt. Die vielen, getrennt nachgewiesenen Lemmata waren nichts anderes als Teile einer großen globalen Invarianten. Die Argumentation in dieser Arbeit benutzte sehr stark Begriffe wie Zustandsmengen und Ausführungspfade. Dies waren im wesentlichen Überlegungen, mit denen sehr schnell für eine große Anzahl von Estelle-Transitionen ausgeschlossen wurde, daß sie die Erfülltheit der jeweils betrachteten Zusicherung verändern. Die Beweisführung war allerdings häufig auf natürliche Sprache angewiesen, weil die formalen Mittel fehlten, um zum Beispiel über die Inhalte von Warteschlangen zu sprechen.

Nun zu den Liveness-Eigenschaften: Die wichtigsten sind wie gesagt „leads-to“-Eigenschaften. Ein Spezialfall ist die Zusicherung der Termination. Im allgemeinen Fall kann die Bedingung, deren Erfüllung nach einer endlichen Anzahl von Schritten zugesichert wird, aber auch eine beliebige andere Aussage über den Zustand sein. Außerdem wird meist nicht nur die einmalige Erfüllung einer Bedingung zugesichert, sondern für jede erneute Erfüllung einer Start-Bedingung die Erfüllung einer Folge-Bedingung zugesichert. Gelegentlich ist die Start-Bedingung `True`, womit spezifiziert wird, daß die Folge-Bedingung unendlich oft erfüllt sein soll.

In manchen Spezifikationstechniken ist nicht nur ausdrückbar, daß eine Bedingung notwendigerweise (mindestens einmal) erfüllt werden muß, sondern auch, daß es lediglich potentiell ist, daß sie erfüllt wird. Die Beweisverfahren sind für diesen Fall ganz ähnlich, nur daß nicht gezeigt werden muß, daß alle Ausführungsfolgen zur Erfüllung der Folge-Bedingung führen, sondern daß es im Ausführungsbaum mindestens einen Pfad gibt, der zur Erfüllung der Folge-Bedingung führt. (Für Aussagen über die potentielle Erfüllung einer Bedingung fassen wir die Menge aller möglichen Ausführungsfolgen zu einem einzigen Ausführungsbaum zusammen, indem wir die gemeinsamen Anfangsstücke von je zwei Folgen „aufeinanderkleben“, so daß sie sich erst trennen, wenn sich die korrespondierenden Zustände zum ersten Mal unterscheiden.)

Der „leads-to“-Operator, der die unvermeidliche Erfüllung einer Bedingung beschreibt, muß für Eigenschaften dieser Art durch einen Operator ersetzt werden, der die potentielle Erfüllung einer Bedingung beschreibt. Damit kann zum Beispiel `LATTICE`, die grundlegende Schlußregel für unvermeidliche Eigenschaften aus [Lam91] (siehe etwas weiter unten), durch einen einfachen Austausch der Operatoren in eine Schlußregel für potentielle Eigenschaften verwandelt werden.

Lamport hält nichts davon, potentielles Verhalten zu beschreiben, weil dies seiner Meinung nach nur ein schwacher Ersatz für Liveness-Eigenschaften ist, welche in seiner TLA direkt ausdrückbar sind. Unserer Erfahrung nach wären derartige Ausdrucksmöglichkeiten in Lamports TLA nur schwer sinnvoll zu integrieren, weil die TLA insbesondere dazu geeignet ist, ein System auf verschiedenen Stufen der Abstraktion zu beschreiben. Dagegen ergab sich im Rahmen der Arbeiten für [Bre90] die Erkenntnis, daß Aussagen über potentielles Verhalten nicht invariant gegen einen Wechsel der Abstraktionsebene sind.[20]

Zum Beweis von Unvermeidlichkeitseigenschaften können wir entweder auf bereits vorhandene Zusicherungen über Liveness aufbauen, beispielsweise auf eine Zusicherung, daß die nichtdeterministische Auswahl von Transitionen eines Systems in einem bestimmten Sinne „fair“ ist. (Für Estelle steht uns eine solche Aussage nicht zur Verfügung.) Oder wir müssen ein grundlegendes Beweisverfahren anwenden, bei dem mit Hilfe einer Wohlordnung die Unvermeidlichkeit der Erfüllung einer Bedingung gezeigt wird. (Das formale Schlußregelsystem in [Lam91] enthält für beide Möglichkeiten spezielle Schlußregeln, wobei sich alle diese Regeln letztlich von der einen Regel über die Wohlordnung ableiten lassen.)

Das genannte grundsätzliche Beweisverfahren für Unvermeidlichkeitseigenschaften

[20] Der Begriff des potentiellen Verhaltens ist nur im Kontext von nichtdeterministischen Entscheidungen sinnvoll. Aber Nichtdeterminismus ist eng mit dem Grad der Abstraktion verbunden: Auf der untersten Abstraktionsebene sind alle Details des Systems bekannt. Daher gibt es dort kein nichtdeterministisches Verhalten mehr. (Ausnahmen entstehen nur durch erstens die Heisenbergsche Unschärfenrelation der Quantenphysik und zweitens den nicht formalisierbaren menschlichen Willen, sofern im System Menschen vorkommen.) Für alle anderen Abstraktionsstufen gilt, daß die Beschreibung um so weniger Nichtdeterminismus enthält, je mehr Details betrachtet werden.

funktioniert zum Beispiel folgendermaßen:

Es wird die Existenz einer Funktion mit folgenden Eigenschaften gezeigt:

- Die Funktion ordnet jedem Zustand ein Element einer Wohlordnung, zum Beispiel der natürlichen Zahlen, zu.
- In jedem Zustand, dem das kleinste Element der Wohlordnung zugeordnet ist, ist die gewünschte Bedingung erfüllt.
- Allen Nachfolgerzuständen eines Zustandes sind kleinere Elemente zugeordnet, außer wenn dem Zustand bereits das kleinste Element zugeordnet war.

Außerdem muß es zu jedem Zustand, dem nicht das kleinste Element zugeordnet ist, mindestens einen Nachfolgerzustand geben.

Und das Transitionssytem muß wohldefiniert sein, was für eine Estelle-Spezifikation bedeutet, daß jede Estelle-Transition immer terminiert, wenn sie schaltet.

(Falls eine Liveness-Eigenschaft nur die Möglichkeit der Erfüllung einer Bedingung fordert, ist in der letzten der Forderungen an die Funktion die Allquantifizierung durch eine Existenzquantifizierung zu ersetzen.)

Das eben beschriebene Verfahren wird oft variiert. Zum Beispiel muß man nicht unbedingt jedem Zustand genau ein Element der Wohlordnung zuordnen. Es können auch mehrere oder gar keines sein. Man kann dann den Beweis so konstruieren, daß die gewünschte Eigenschaft erfüllt ist, wenn kein kleineres Element der Wohlordnung mehr existiert. Dabei wird nach wie vor ausgenutzt, daß jede streng monoton sinkende Folge von Elementen einer Wohlordnung endlich sein muß. Die formale Schlußregel `LATTICE` in [Lam91] ist für einen derartigen Beweis konstruiert.

Anmerkung: Diese Schlußregel `LATTICE` benutzt in ihrer Prämisse den „leads-to“-Operator, mit dem nicht nur unmittelbar aufeinanderfolgende Zustände betrachtet werden. In der Anwendung bedeutet dies aber nur, daß eine Menge von Verzweigungsmöglichkeiten Einzelschritt für Einzelschritt untersucht werden muß, damit gezeigt wird, daß die gewünschte Bedingung nach endlich vielen Schritten erfüllt wird.

Wie wir gesehen haben, besitzen die Beweisverfahren für Safety- und Liveness-Eigenschaften viele Gemeinsamkeiten: In einem ersten, kreativen Schritt muß eine geeignete globale Invariante beziehungsweise eine geeignete Funktion über dem Zustandsraum gefunden werden. Im zweiten Schritt müssen dann bestimmte Eigenschaften von Transitionen beziehungsweise Zuständen nachgewiesen werden. Und erst zum Schluß kommt jeweils eine einzige Schlußregel ins Spiel, die die Ergebnisse zusammenfaßt und auf eine temporale Aussage führt.

Das Entscheidende dabei ist: Außer im letzten Schritt wird nicht mit temporalen Eigenschaften argumentiert, sondern nur über prädikatenlogische Zusicherungen über Zustände oder allenfalls über Zusicherungen über das Verhältnis unmittelbar aufeinanderfolgender Zustände. Letzteres wurde in [Lam91] mit dem Begriff der „Aktion“ gut formalisiert. Somit müssen wir während der wesentlichen Beweisschritte nicht mit Ausführungsfolgen, temporalen Operatoren und ähnlichem hantieren, die einfacheren Mittel der gewöhnlichen Logik reichen dabei aus.

Aus diesem Grunde ist es auch angemessen, im Rahmen unserer Diplomarbeit Estelle-Transitionen mit prädikatenlogischen Mitteln darzustellen.

Voraussetzungen für formales Schließen über Estelle-Spezifikationen in dieser Darstellung sind allerdings eine geeignete formale Beschreibung der nachzuweisenden Eigenschaften, eine geeignete Formalisierung der Relation von aufeinanderfolgenden Zuständen sowie ein geeignetes formales Schlußsystem.

Lamports Begriff der „Aktion“ ist hier hervorragend geeignet, aufeinanderfolgende

Zustände zu beschreiben, und auch ein formales Schlußsystem wird von der TLA geboten. Deshalb werden wir uns im folgenden bei den Beispielen von temporalen Eigenschaften an die TLA anlehnen.

Da eine für alles geeignete Beschreibungstechnik noch nicht gefunden ist und vermutlich auch nicht existieren kann, wird es sicherlich Eigenschaften geben, die sich in der TLA nicht (gut) spezifizieren lassen, so daß in diesen Fällen die TLA für einen Beweis nicht verwendet werden kann. Abzusichern ist nur, daß in der TLA die wichtigen Beweistechniken formalisiert sind. Denn für diese Diplomarbeit geht es nur darum, für Estelle eine formale Darstellung zu finden, die für alle wichtigen Beweistechniken gut geeignet ist.

Die eben geforderte Absicherung ist in der Tat notwendig: Lamport legt nur eine sehr einfache temporale Logik zugrunde. Aber es gibt auch ausdrucksstärkere Varianten von temporaler Logik, die es zum Beispiel auch erlauben, sich auf die Vergangenheit zu beziehen; die Erweiterungen können bis hin zu einer Intervallogik gehen (vergleiche etwa [Got92] und [ScMeVo83]). Werden hierfür neue Beweisverfahren notwendig?

Bei Betrachtung dieser Logiken stellt sich zum Glück heraus: Das einzig wesentlich Neue ist, daß sich die spezifizierten Eigenschaften nicht mehr immer auf die gesamte Zeit beziehen, sondern daß sie sich auch nur auf ein Intervall beziehen können. Innerhalb dieses Intervalls greifen weiterhin die alten Beweisverfahren, sowohl für Invarianten wie für Unvermeidlichkeitseigenschaften. Es kann lediglich notwendig werden, die Existenz von Intervallgrenzen nachzuweisen, aber dies läuft letztlich wieder auf den Nachweis von Unvermeidlichkeitseigenschaften hinaus.

Weiterhin sind noch die oben erwähnten Logiken zu bedenken, in denen die Möglichkeit der Erfüllung einer Bedingung ausdrückbar ist. Wie aber oben bereits beschrieben, sind die dafür nötigen Beweisverfahren sehr nahe mit denen für Unvermeidlichkeitseigenschaften verwandt. (Man mußte im Grundverfahren lediglich eine Allquantifizierung durch eine Existenzquantifizierung ersetzen.)

Daher reicht es aus, nur die Beweisverfahren zu berücksichtigen, die wir in diesem Kapitel ausführlich betrachtet haben und die auch Lamport in der TLA formalisiert hat.

4.3 Verschiedene prädikatenlogische Darstellungsformen für Estelle-Transitionen

Eines der Ergebnisse im vorigen Kapitel war, daß wir prädikatenlogische Mittel zur Darstellung von Estelle-Transitionen verwenden sollten. Die Schaltbedingungen von Estelle-Transitionen sind ohnehin bereits so gut wie in einer prädikatenlogischen Form, so daß ihre Darstellung trivial ist. Es bleibt für die Untersuchung also die Darstellung der Rümpfe der Estelle-Transitionen übrig. Die Rümpfe enthalten einen Pascal-artigen Code, und für Stücke von Pascal-artigem Code finden sich in der Literatur (Kapitel 10) die folgenden Möglichkeiten einer prädikatenlogischen Darstellung der Semantik:

Prädikatenlogische Darstellungsformen für Estelle-Transitionsrümpfe

(a) Ein Prädikat über die sich ändernden Teile des Systemzustandes

(b) Ein Prädikat über den Zusammenhang zwischen dem gesamten alten und dem gesamten neuen Systemzustand

(c) Eine Zusicherung nach Hoare aus Vorbedingung, Transition und Nachbedingung

(d) Ein Prädikatentransformator „stärkste Nachbedingung“ nach Dijkstra

(e) Ein Prädikatentransformator „schwächste Vorbedingung“ nach Dijkstra

(f) Eine Mischform aus einigen der vorigen Darstellungformen

Unabhängig von der Beschreibung der Schaltwirkung muß auf jeden Fall die Schaltbedingung einer Transition angegeben werden, dies geschieht für jede der obigen Möglichkeiten sinnvollerweise durch ein einfaches Prädikat.

In den meisten Darstellungsformen könnte man die Darstellung der Schaltbedingung leicht in die Darstellung der Schaltwirkung integrieren, aber damit würden wir wertvolle Information verschenken, die uns die Estelle-Form bereits fertig liefert. Denn für Beweise braucht man sehr oft die Information, wann eine Estelle-Transition schaltbereit ist.

4.4 Das untersuchte Beispiel: Der Euklidische Algorithmus

Die im vorigen Kapitel zusammengetragenen prädikatenlogischen Darstellungsformen für die Rümpfe von Estelle-Transitionen sollen anhand eines einfachen Beispiels auf ihre Verwendbarkeit für unsere Zwecke untersucht werden. Dieses Beispiel stellen wir jetzt vor:

Der Euklidische Algorithmus

Er berechnet iterativ den größten gemeinsamen Teiler (`ggT`) von zwei positiven ganzen Zahlen.

Es handelt sich um einen gewöhnlichen sequentiellen, terminierenden Algorithmus. Er ist zum Beispiel in ähnlicher Form in [Dij76] beschrieben, nur wird die dortige allgemeine **do..od**-Schleife hier mit Estelle-Transitionen ausgedrückt.

Weil die besonderen Möglichkeiten von Estelle nicht genutzt werden, können wir in diesem einfachen Beispiel das Ausführungsmodell von Estelle einfach als ein ganz primitives Transitionssystem ansehen und müssen für dieses Moment nicht zwischen Estelle-Transitionen und Transitionen des abstrakten Automaten unterscheiden. So ersparen wir uns vorläufig die explizite Angabe des Estelle-Ausführungsmodells.

Folgende Eigenschaften wünschen wir uns vom Algorithmus (wobei `a`, `b` „starre" Variablen wie in der TLA, oft auch Konstanten genannt, sind). Zur Darstellung benutzen wir eine Mischung der Notation der temporalen Logik der Aktionen, siehe Kapitel 3.2, mit der Schreibweise für Beweise aus [DiSc90] [21], siehe Kapitel 3.4:

```
(1)   □( State = finished  ⇒  g = ggT(a,b) )     (Safety)
(2)   a > 0 ∧ b > 0  ⇒  ◇(State = finished)      (Liveness)
```

Dabei ist die Definition von `ggT(a,b)`:

```
[(a > 0 ∧ b > 0)
 ⇒ (ggT(a,b) | a
    ∧ ggT(a,b) | b
    ∧ ¬(∃ c: c | a ∧ c | b: c > ggT(a,b)))]
```

Eine Anmerkung: In unserer Definition können `a` und `b` auch kleiner oder gleich null sein.[22] (In der Estelle-Spezifikation weiter unten müssen sie lediglich vom Typ **Integer** sein.) Allerdings ist die Implikation für $a \leq 0 \vee b \leq 0$ trivialerweise erfüllt, so daß keinerlei Aussage mehr über `ggT(a,b)` gemacht wird. Oder anders ausgedrückt: Egal welchen Wert man für `ggT(a,b)` in diesem Falle wählt, es widerspricht nicht unserer Definition. Nur wenn `a` und `b` beide größer als null sind, legt die Definition `ggT(a,b)` eindeutig fest. Dies entspricht Lamports Vorgehen in der TLA zum Beispiel bei der Definition der Division durch null. Die Division ergibt garantiert immer einen Wert, aber Lamport gibt keinerlei Zusicherung über die Beschaffenheit des Ergebnisses. Daher kann dann keine Zusicherung mehr bewiesen werden, die irgendeine Eigenschaft des Ergebnisses voraussetzt, und sei es nur, daß das Ergebnis eine reelle Zahl ist.

---

[21] Hier sind noch Notationsprobleme zu lösen: Eckige Klammern ohne tiefgestellten Index bezeichnen den Überall-Operator für boolsche Strukturen, während sie mit tiefgestellter Zustandsfunktion anzeigen, welche Variablen von der eingeklammerten Aktion während eines „Stotter"-Schritts unverändert gelassen werden.

[22] Dahinter verbirgt sich das altbekannte Problem der partiellen Funktionen (wie es zum Beipiel in [Par92] von Parnas diskutiert wird). Im Rahmen von Software-Spezifikationen hat man häufig mit Funktionen zu tun, die nicht für beliebige Argumente definiert sind. Andererseits verlangen konventionelle formale Interpretationen von logischen Ausdrücken, daß alle vorkommenden Funktionen total sind, das heißt, für alle denkbaren Argumente definiert

Noch eine Anmerkung: In den beiden Eigenschaften (1) und (2) wird die Variable „State“ erwähnt, die mit dem eigentlichen Problem nichts zu tun hat und nur benötigt wird, um das Ende der Berechnung ausdrücken zu können. Im Rahmen der TLA könnte man diese unschöne Formulierung leicht vermeiden, indem eine „Aktion“ spezifiziert wird, die das gewünschte Ergebnis der Variablen „g“ zuweist. Trotzdem wurde die obige Formulierung mit Bedacht gewählt, denn sie enthält nur einfache „Prädikate“ im Sinne von [Lam91]. Da wir verschiedene Formalismen untersuchen wollen, dürfen wir nicht die Spezialität eines davon von vorneherein zur Grundlage machen. Dann wäre das Ergebnis bereits festgelegt.

Drei abgeleitete Eigenschaften von ggT sind:

```
[a > 0  ⇒  ggT(a,a) = a]

[(a > 0 ∧ b > 0)  ⇒  ggT(a,b) > 0]

[(a > 0 ∧ b > 0)  ⇒  ggT(a,a+b) = ggT(a,b)]
```

Beweis der letzten Behauptung:

```
    a > 0 ∧ b > 0
⇒ { Arithmetik }
    a > 0 ∧ a + b > 0
⇒ { Definition von ggT }
    ggT(a,a+b) | a
    ∧ ggT(a,a+b) | a+b
    ∧ ¬(∃ c: c | a ∧ c | a+b: c > ggT(a,a+b)))
= { Definition von „|“, „Trading“ }
    (∃d: d ∈ ℕ⁺: (∃e: e ∈ ℕ⁺:
      ggT(a,a+b) * d = a
      ∧ ggT(a,a+b) * e = a+b
    ∧ ¬(∃ c:: (∃f: f ∈ ℕ⁺: (∃g: g ∈ ℕ⁺:
        c * f = a ∧ c * g = a+b ∧ c > ggT(a,a+b)))))))
= { zwei arith. Umformungen, wg. a,b,d,e,f,g,ggT(a,a+b) > 0) }
    (∃d: d ∈ ℕ⁺: (∃e: e ∈ ℕ⁺:
      ggT(a,a+b) * d = a
      ∧ ggT(a,a+b) * e = a+b
      ∧ d/e = a/(a+b)
    ∧ ¬(∃ c:: (∃f: f ∈ ℕ⁺: (∃g: g ∈ ℕ⁺:
        c * f = a ∧ c * g = a+b ∧ f/g = a/(a+b)
        ∧ c > ggT(a,a+b)))))))
= { zwei arith. Umform.: jeweils Subtr. von 1 auf beiden Seiten }
    (∃d: d ∈ ℕ⁺: (∃e: e ∈ ℕ⁺:
      ggT(a,a+b) * d = a
      ∧ ggT(a,a+b) * e = a+b
      ∧ (e-d)/e = b/(a+b)
    ∧ ¬(∃ c:: (∃f: f ∈ ℕ⁺: (∃g: g ∈ ℕ⁺:
        c * f = a ∧ c * g = a+b ∧ (g-f)/g = b/(a+b)
        ∧ c > ggT(a,a+b)))))))
⇒ { zweimal Einsetzen, Weglassen von Teiltermen }
```

sind. Manche Autoren bemühen daher dreiwertige Logiken, Parnas definiert die grundlegenden Prädikate =,<,≥,... so um, daß sie bei undefinierten Argumenten False ergeben, und Lamport löst das Problem in der TLA auf die oben im folgenden beschriebene Weise. Wir halten uns an Lamports Lösung, da sie elegant ist und außerdem über die TLA auf natürliche Weise bereits in die TLA/PT einfließt.

```
    (∃d: d ∈ ℕ⁺: (∃e: e ∈ ℕ⁺:
      ggT(a,a+b) * d = a
      ∧ ggT(a,a+b) * (e-d) = b
    ∧ ¬(∃ c:: (∃f: f ∈ ℕ⁺: (∃g: g ∈ ℕ⁺:
        c * f = a ∧ c * (g-f) = b ∧ c > ggT(a,a+b)))))))
= { Definition von „|“, „Trading“ }
    ggT(a,a+b) | a
    ∧ ggT(a,a+b) | b
    ∧ ¬(∃ c: c | a ∧ c | b: c > ggT(a,a+b)))
= { Def. ggT }
    ggT(a,a+b) = ggT(a,b)
```

Wir behaupten nun, daß folgende Spezifikation in Estelle die Eigenschaften (1) und (2) erfüllt:

```
Specification euklid Systemprocess;

Const
  a = Any Integer;
  b = Any Integer;

Var
  x : Integer;
  y : Integer;
  g : Integer;

State
  working,
  finished;

Initialize
  To working
    Begin
      x := a;
      y := b
    End;

Trans
  From working
    Provided x < y
Name t1:
      Begin
        y := y - x
      End;

Trans
  From working
    Provided y < x
Name t2:
      Begin
        x := x - y
      End;
```

```
Trans
  From working
    Provided x = y
      To finished
Name te:
        Begin
          g := x
        End;

End.
```

Für dieses einfache Beispiel ist eine globale Invariante zum Beweis der Safety-Eigenschaft (1) leicht zu finden, wenn man die Funktionsweise des Algorithmus′ verstanden hat. Sie lautet:

(3) $(a > 0 \wedge b > 0 \;\Rightarrow\; x > 0 \wedge y > 0 \wedge ggT(x,y) = ggT(a,b))$
$\wedge\; (State = finished \;\Rightarrow\; g = ggT(a,b))$

Für die Liveness-Eigenschaft (2) ist eine passende Funktion des Zustandsraumes nach $\mathbb{N}_0$:

(4) $f(x,y,g,State) = max(0,\ (x + y) * ind1(State \neq finished))$

(Dabei ist `ind1(p)` die Indikatorfunktion, die den Wert eins annimmt, wenn `p` wahr ist, und den Wert null, falls `p` falsch ist.)

Durch die Verwendung der Maximumfunktion ist offensichtlich sichergestellt, daß die Funktion `f` niemals Werte kleiner als null liefern kann, egal für welchen Zustand. Hierdurch sichern wir ab, daß der Wertebereich der Funktion `f` tatsächlich nur die Menge $\mathbb{N}_0$ umfaßt, also nach unten beschränkt ist, so daß wir $(\mathbb{N}_0, <)$ als Wohlordnung für unseren Liveness-Beweis verwenden können. Ansonsten ist aber völlig uninteressant, welche Werte `f` liefert, falls `a` oder `b` kleiner als null ist, da dann die Liveness-Eigenschaft (2) trivialerweise erfüllt ist. Und falls weder `a` noch `b` kleiner als null ist, können wir aus (3) sofort den Satz $(a>0 \wedge b>0) \Rightarrow (x>0 \wedge y>0)$ ableiten und schließen, daß auch weder `x` noch `y` kleiner als null ist, so daß dann das rechte Argument der Maximumfunktion ohnehin nicht kleiner als null ist.

4.5 Untersuchung der prädikatenlogischen Darstellungsformen für Estelle-Transitionen

Um die in Kapitel 4.3 zusammengetragenen Semantikdefinitionstechniken auf ihre Eignung für unsere Zwecke zu prüfen, versuchen wir, sie auf das Beispiel aus dem vorigen Kapitel anzuwenden.

Als erstes versuchen wir überhaupt einmal, unser Estelle-Beispiel in die jeweilige Darstellungsform umzusetzen. Wenn dies gelingt, versuchen wir anschließend zu verifizieren, daß erstens jede Transition die Gültigkeit der Zusicherung (3) vom Ende des vorigen Kapitels erhält und daß zweitens die dortige Funktion f aus der Definition (4) die gewünschten Eigenschaften besitzt. Wenn sowohl die Darstellung als auch beiden Verifikationen möglich sind, dann unterstützt die Darstellungstechnik die wesentlichen Beweistechniken, wie wir sie in Kapitel 4.2 beschrieben haben.

Die in diesem Kapitel geführten Beweise sind halb formal, da zwar der Estelle-Text jeweils mathematisch dargestellt wird, wir als Schlußregeln aber nur die allgemein bekannten Regeln der Prädikatenlogik und der Arithmetik verwenden. Aber da wir uns zunächst einmal nur für die Ausdruckskraft der formalen Darstellung interessieren, schadet es nicht, wenn die Schlußregeln noch nicht formalisiert sind.

(a) Beschreibung einer Estelle-Transition durch ein Prädikat über die sich ändernden Teile des Systemzustandes

Ein Prädikat gibt den Zusammenhang zwischen dem gesamten alten und dem gesamten neuen Systemzustand an. Dabei bedeutet die Nichterwähnung einer Variablen implizit, daß sie unverändert bleibt.

Wir setzen die Estelle-Transitionen des Beispiels aus Kapitel 4.4 in diese Darstellungsform um.

Für die Unterscheidung des alten und des neuen Systemzustandes bei den Schalteffekten verwenden wir die meist übliche Notation, wie sie unter anderem auch in der TLA verwendet wird: Ein ungestrichener Variablenname beschreibt den Wert vor dem Schalten und ein gestrichener den Wert nachher.[23]

Die Notation für die Aufteilung in Schaltbedingung und Schaltwirkung ist locker an CRS ([MaNe87]) angelehnt.

```
Initialize
  Effects:   x′ = a
           ∧ y′ = b
           ∧ State′ = working
```

(Es wird an dieser Stelle trotz der Nichterwähnung der Variablen g keine Aussage über g′ gemacht, da vor der Initialisierung kein Zustand existiert, und es folglich auch keine Aussage über g vor der Initialisierung gibt. Auf diese Weise wird beschrieben, daß die Variable g nicht initialisiert wird.)

```
Trans t1:
  Cond:      State = working
           ∧ x < y
  Effects:   y′ = y - x
```

(Die Variablen x, g und State bleiben unverändert, da x′, g′ und State′ nicht in den Schaltwirkungen erwähnt werden.)

[23] Einige Autoren machen es auch genau umgekehrt, zum Beispiel [MaNe87] in CRS.

```
Trans t2:
  Cond:      State = working
           ∧ y < x
  Effects:   x′ = x - y

Trans te:
  Cond:      State = working
           ∧ x = y
  Effects:   g′ = x
           ∧ State′ = finished
```

Hier bei (a) werden im Gegensatz zu (b) nur die Variablen aufgeführt, die verändert werden. Das erscheint auf den ersten Blick eleganter und weniger aufwendig. Allerdings handeln wir uns damit das sogenannte „Frame-Problem“[24] ein: Wir können nicht immer sofort sagen, welche Variablen unverändert bleiben, da diese Menge nicht explizit angegeben ist, sondern ihre Größe von der Definition der Umgebung (daher „Frame“) abhängt.

Im Gegensatz zu (b) ist das Prädikat „`(z′ = z′)`“ nicht immer wahr: Wenn wir „`∧ (z′ = z′)`“ an eine Formel anhängen, verändern wir unter Umständen ihren Wahrheitswert. Beispiel:

(1) `x′ = x+1 ∧ y′-4 = x`

(2) `x′ = x+1 ∧ y′-4 = x ∧ z′ = z′`

(1) enthält implizit den Konjunktor „`(z′ = z)`“. In (2) aber ist er nicht enthalten, `z′` darf jeden beliebigen Wert annehmen. Damit sind (1) und (2) nicht äquivalent, und „`(z′ = z′)`“ ist folglich nicht Einselement der Konjunktion. Diese Eigenschaft besitzt aber „`True`“, weshalb „`(z′ = z′)`“ nicht äquivalent zu „`True`“ sein kann.

Aus diesem Grunde geht Lamport mit seiner TLA ([Lam91]) den in (b) beschriebenen Weg.

Ansonsten ist (a) aber gleichwertig zu (b). Wir betrachten daher gleich als nächstes (b), um die für beide Versionen gemeinsamen Argumente nicht doppelt anzuführen:

(b) Beschreibung einer Estelle-Transition durch ein Prädikat über den gesamten Systemzustand

Ein Prädikat wie bei (a), aber ohne die implizite Annahme über die ungenannten Variablen: Die ungenannten Variablen dürfen im Nachzustand jeden beliebigen Wert annehmen.

Auch hier setzen wir die Estelle-Transitionen des Beispiels aus Kapitel 4.4 in diese Darstellungsform um:

```
Initialize
  Effects:   x′ = a
           ∧ y′ = b
           ∧ State′ = working
```

(Da die Variable `g` nicht initialisiert werden soll, wird wiederum keine Aussage über `g′` gemacht. `g′` darf jeden beliebigen Wert einnehmen. Und da es hier keinen Vorzustand gibt, wäre es auch sinnlos, ein ungestrichenes `g` zu verwenden, um zum Beispiel zu spezifizieren, daß die Variable `g` unverändert bleibt.)

[24] Beschreibungen des allgemeinen Frame-Problems, eine Bibliographie und verschiedene Ansätze zur Lösung des allgemeinen Problems finden sich in [Bro87].

```
Trans t1:
  Cond:      State = working
           ∧ x < y
  Effects:   y′ = y - x
           ∧ x′ = x
           ∧ g′ = g
           ∧ State′ = State

Trans t2:
  Cond:      State = working
           ∧ y < x
  Effects:   x′ = x - y
           ∧ y′ = y
           ∧ g′ = g
           ∧ State′ = State

Trans te:
  Cond:      State = working
           ∧ x = y
  Effects:   g′ = x
           ∧ x′ = x
           ∧ y′ = y
           ∧ State′ = finished
```

Wie man leicht sieht, läßt sich unser Algorithmus damit gut verifizieren. (Bei aufwendigen Prädikaten über die Estelle-Transitionen wird die Umformungsarbeit natürlich größer.)

Bisher unbeantwortet geblieben ist aber die Frage, wie die Prädikate aus dem Text der Transitionen abgeleitet werden.

Um ein Prädikat zu erhalten, das alle Wirkungen einer gesamten Estelle-Transition beschreibt, ist es sinnvoll, zuerst Prädikate für die primitiven Operationen zu definieren und dann Regeln anzugeben, wie die Prädikate für zusammengesetzte Operationen zu bestimmen sind. Unter anderem Hoare hat diesen Ansatz verfolgt, er sagte in [Hoa85] sogar: „Programme sind Prädikate“.

Für Operationen wie die einfache Zuweisung oder die bedingte Verzweigung lassen sich leicht die zugehörigen Prädikate angeben. Wir bekommen aber nur dann ein Prädikat für den gesamten Rumpf der Estelle-Transition, wenn wir auch ein gutes Verfahren zur Darstellung der sequentiellen Komposition von Operationen haben. Und hier besteht laut [ZwRo89] eine Schwäche der dort „Sat-Systeme“ (*PP*) genannten Systeme. So mußte auch Hoare in [Hoa85] auf S. 152 zugeben, daß die sequentielle Komposition recht kompliziert ist, und er gab in seinem dort vorgestellten, einfachen Kalkül keine allgemeine Definition für sie an. Die Schwierigkeiten entstehen insbesondere, wenn sequentielle Komposition und Iteration aufeinander treffen.

Die Schwierigkeiten mit der sequentiellen Komposition verdienen eine ausführlichere Diskussion:

In [Hoa85] wird für den Operator der bedingten Verzweigung des Kalküls eine Auflösung in die gewöhnliche prädikatenlogische Form gegeben:

$$P \not< b \not> Q \equiv (b \wedge P \vee \neg b \wedge Q) \equiv ((b \Rightarrow P) \wedge (\neg b \Rightarrow Q))$$

Ebenso für den Zuweisungsoperator:

```
(e >- x -> P(x)) ≡ P(e)
```

Auch für die Rekursion wird eine mathematische Definition gegeben, die allerdings auf die Fixpunkttheorie zurückgreifen muß.

Im Gegensatz zu den übrigen Operatoren gibt Hoare für die sequentielle Komposition nur algebraische Eigenschaften in Bezug auf die anderen Operatoren (außer der Rekursion) an. Da die Bedeutung der sequentiellen Komposition nur umgangssprachlich erklärt wird, aber nicht auf der Grundlage der gewöhnlichen Prädikatenlogik definiert wird, ist ihr Operator „;“ folglich selbst ein grundlegendes Element des definierten Kalküls. Da „;“ aber sicherlich nicht zu den gewöhnlichen Operatoren der Prädikatenlogik gehört, wird damit der selbstgestellte Anspruch „Programme sind Prädikate“ nicht vollständig eingelöst.

Es bleibt das Problem, wie man aus zwei Prädikaten, die sequentiell verbundene Programmstücke beschreiben, ein einziges Prädikat gewinnen kann, das die Bedeutung des gesamten Objektes beschreibt. Der einfachste Weg, der letzendlich auch dem von Hoare entspricht, ist, nach jeder Elementaroperation einen Zwischenzustand zu betrachten und dies mit Hilfsvariablen auszudrücken. Beispiel:

Pascal	Prädikat
x := y	$x_1 = y_0 \wedge y_1 = y_0$
x := x+1	$x_2 = x_1+1 \wedge y_2 = y_1$
x := y; x := x+1	$x_1 = y_0 \wedge y_1 = y_0 \wedge x_2 = x_1+1 \wedge y_2 = y_1$

Um daraus ein Prädikat ausschließlich über das gesamte Objekt zu erhalten, müssen die Hilfsvariablen eliminiert werden, oder falls das nicht möglich ist, mit Hilfe von Existenzquantifikationen verborgen werden:

$$(\exists x_1, y_1 :: x_1 = y_0 \wedge y_1 = y_0 \wedge x_2 = x_1+1 \wedge y_2 = y_1)$$

$$x_2 = y_0+1 \wedge y_2 = y_0$$

Wenn die Bedeutung eines seqentiellen Programmstücks stückweise zusammengefügt wird, nehmen die Prädikate der Teilstücke im Laufe der Arbeit komplizierte Formen an, so daß die individuelle Inspektion der Struktur des jeweiligen Prädikates für die Vereinfachung erforderlich wird. Unterläßt man die Vereinfachung, so wird das Prädikat sehr schnell hoffnungslos unübersichtlich.

An dieser Stelle wird auch klar, warum die allgemeine Iteration so viele Schwierigkeiten macht: Bei ihr ist die Zahl der Schleifendurchläufe nicht von vorneherein festgelegt, so daß auch die Anzahl der Zwischenzustände nicht feststeht. Und damit ist das eben angedeutete Verfahren, das eine feste Zahl von Zwischenschritten voraussetzt, nicht mehr durchführbar.

Soweit die Ausführungen zu den Problemen der sequentiellen Komposition, zurück zur Darstellung von Estelle-Transitionen durch Prädikate.

In Estelle-Transitionen wird die Iteration zwar normalerweise keine sehr große Rolle spielen, aber wir können sie auch nicht einfach ausschließen, da wir damit die mögliche Ausdruckskraft einer Estelle-Transition sehr stark einschränken würden.

Das Ergebnis der Betrachtung von (a) und (b) ist also: Da es kein gutes, allgemeines Verfahren gibt, Pascal-ähnlichen Text in ein Prädikat umzusetzen, können wir die Wirkung der Estelle-Transitionen nicht durch Prädikate beschreiben.

Dies steht nicht im Widerspruch zu der Tatsache, daß die temporale Logik der Aktionen ([Lam91]) durchaus erfolgreich mit Prädikaten arbeitet. Denn dort wird

üblicherweise zuerst in TLA in mehreren Verfeinerungsstufen spezifiziert und erst in der letzten Stufe die TLA-Spezifikation in eine Programmiersprache umgesetzt. Wir dagegen verfolgen hier den umgekehrten Ansatz, wir wollen eine vorhandene Estelle-Spezifikation analysieren können.

Auch der unserem Ziel entgegengesetzte Weg, daß wir eine mathematische Beschreibungsform angeben, die formal in Estelle umgesetzt werden kann, wäre ein interessantes Ergebnis, aber es wäre keine Definition der Semantik von Estelle mehr.

Daß die TLA für unsere Zwecke nicht direkt verwendbar ist, bedeutet aber nicht, daß wir uns dieses bereits fertige formale Schlußsystem nicht trotzdem nutzbar machen können. Wir kommen bei der Diskussion des Punktes (f) und bei der Zusammenfassung der Ergebnisse in Kapitel 4.6 darauf zurück.

(c) Beschreibung einer Estelle-Transition durch eine Zusicherung nach Hoare

Vor dem eben beschriebenen Kalkül hatte Hoare bereits ein weiteres und viel bekannteres veröffentlicht ([Hoa69], [HoWi73]), das nach der Klassifikation von [ZwRo89] dem Ansatz „Programme sind Prädikatentransformatoren“ (PT) folgt.

Es wird eine sogenannte Zusicherung aus Vorbedingung, Transition und Nachbedingung angegeben, um die Schaltwirkung zu beschreiben:

```
{ P } tr { Q }
```

Bei dem Hoareschen Axiomen- und Schlußregelsystem tritt das bei (b) geschilderte Problem nicht auf, da es eine elegante Schlußregel für die sequentielle Komposition gibt.

Eine solche Zusicherung unterscheidet sich in einem Punkt wesentlich von einem Prädikat wie bei (b): Im allgemeinen wird die Wirkung der Transition durch die Vor- und Nachbedingungen `P` und `Q` nicht vollständig beschrieben. Wenn das Kalkül vollständig ist, ist es lediglich möglich, für jede gültige Eigenschaft eine Zusicherung herzuleiten. Dies ist für Korrektheitsbeweise sogar vorteilhaft, aber zur kontextunabhängigen Definition der Wirkung einer Estelle-Transition nachteilig:

Will man eine Estelle-Transition in *eine* Zusicherung umsetzen, so muß die Zusicherung stark genug konstruiert werden.

Es ist immer möglich, für eine Transition folgendes nachzuweisen:

```
{ True } tr { True }
```

Denn bei der Anwendung der Schlußregeln hat man immer die Wahl, welche Paare aus Vor- und Nachbedingungen man nachweisen möchte.

Um die vollständige Information über die Estelle-Transition zu bekommen, kann man entweder die Vor- oder die Nachbedingung so formulieren, daß sie genau einen Zustand beschreibt. Für die Nachbedingung (da in ihrem Falle die Zuweisungs-Schlußregel leicht anwendbar ist) sieht das so aus:

```
Trans t1:
  Cond:      State = working
           ∧ x < y
  Effects: { x = x1 ∧ y-x = y1 ∧ g = g1 ∧ State = State1 }
           t1
           { x = x1 ∧ y = y1 ∧ g = g1 ∧ State = State1 }
```

Wenn wir so verfahren, ist allerdings die Nachbedingung (oder im anderen Falle die Vorbedingung) redundant und kann weggelassen werden. Damit sind wir wieder genau bei dem Ansatz (b) angelangt. Das dort vorhandene Problem mit der sequen-

tiellen Komposition ergibt sich hier entsprechend, sobald wir im allgemeinen Fall die Schlußregel für sequentielle Komposition anwenden wollen, *ohne* Information fallenzulassen.

Und selbst wenn es uns gelungen ist, die Vorbedingung ohne Informationsverlust zu finden, so tritt immer noch ein Problem auf, das in [ZwRo89] als Adaptionsproblem bezeichnet wird:

Haben wir

$\{ \mathtt{P} \} \ \mathtt{tr} \ \{ \mathtt{Q} \}$

nachgewiesen, so wollen wir daraus zum Beispiel

$\{ \mathtt{GIV} \wedge \mathtt{Cond} \} \ \mathtt{tr} \ \{ \mathtt{GIV} \}$

nachweisen, ohne auf die Struktur von `tr` zurückzugreifen. Wenn „$\mathtt{GIV} \wedge \mathtt{Cond}$“ nicht `P` impliziert, können wir die Konsequenzregel nicht (direkt) anwenden, und das Problem wird laut [ZwRo89] „nichttrivial“.

Die Eleganz des Ansatzes der Hoareschen Zusicherungen beruht darauf, daß es möglich ist, für den aktuellen Zweck, also für das Beweisen einer bestimmten Zusicherung, unwichtige Information wegzulassen. Damit ist er aber nicht geeignet, unabhängig von einem konkreten Beweisziel eingesetzt zu werden.

Wie in Kapitel 4.2 ausgeführt ist es aber notwendig, die Estelle-Transitionen unabhängig von einem konkreten Beweisziel zu beschreiben, da bei einer Verifikation die globale Invariante im allgemeinen in mehrere Teilaussagen aufgeteilt sein wird, die nacheinander nachgewiesen werden müssen.

(d) Beschreibung einer Estelle-Transition durch einen Prädikatentransformator „stärkste Nachbedingung“

Es gilt nicht nur für die Hoareschen Zusicherungen aus (c), sondern generell für alle PT-Systeme, daß sie nicht dazu gedacht sind, ein Prädikat über das gesamte Systemverhalten zu liefern, sondern jeweils nur Prädikate über die gerade interessierenden Teileigenschaften. Dies ist im Falle der Verifikation sequentieller Programme sogar ein Vorteil.

Mit dem System der Hoareschen Zusicherungen haben wir also einen Prädikatentransformator. Allerdings hat er den Nachteil, daß er nicht explizit als Operator angebbar ist, sondern in der Anwendung der jeweils richtigen Schlußregeln des Kalküls besteht. (Nur falls wir die Probleme der sequentiellen Komposition und das Adaptionsproblem lösen können, können wir uns dies ersparen.)

Es gibt allerdings auch PT-Systeme, in denen die Prädikatentransformatoren explizit angegeben werden, zum Beispiel „`sp`“ und „`wp`“ von Dijkstra. In diese Operatoren kann man ein Prädikat hineinstecken, und durch eine Reihe von Textersetzungen, die auch automatisch ausführbar sind, erhält man das andere Prädikat. Ein Zugriff auf den Programmtext wie bei (c) ist nicht notwendig. Natürlich ist es dann immer noch erforderlich, das erhaltene Prädikat so umzuformen, daß die Äquivalenz mit dem Gewünschten nachgewiesen wird.

Betrachten wir zuerst den Prädikatentransformator `sp` von Dijkstra ([DiSc90]), der die stärkste Nachbedingung berechnet, und beschreiben mit ihm die Estelle-Transitionen des Beispiels aus Kapitel 4.4.

Anmerkung: Wie in Kapitel 3.4 beschrieben, führt der Prädikatentransformator `(x:=z)` die Textersetzung ≫Text „`z`“ an die Stelle von Text „x“≪ auf seinem Prädikat aus, so daß beispielsweise gilt: `(x:=z).(x+y) = (z+y)`

```
Initialize
  Effects: [sp.Initialize.Y ≡
                State = working
              ∧ y = b
              ∧ x = a
              ∧ (∃State:: (∃y:: (∃x:: Y)))]

Trans t1:
  Cond:      State = working
           ∧ x < y
  Effects: [sp.t1.Y ≡ (∃z: y = (z-x): (y:=z).Y)]
  (Wobei die Variable z in Y nicht vorkommt.)

Trans t2:
  Cond:      State = working
           ∧ y < x
  Effects: [sp.t2.Y ≡ (∃z: x = (z-y): (x:=z).Y)]
  (Wobei die Variable z in Y nicht vorkommt.)

Trans te:
  Cond:      State = working
           ∧ x = y
  Effects: [sp.te.Y ≡
                State = finished ∧ g = x
              ∧ (∃State:: (∃g:: Y))]
```

Als Beispiel werden wir jetzt nachweisen, daß die Transition `t1` die globale Invariante tatsächlich erhält:

Sie war definiert durch:

```
[GIV ≡ (a > 0 ∧ b > 0  ⇒  x > 0 ∧ y > 0 ∧ ggT(x,y) = ggT(a,b))
       ∧ (State = finished  ⇒  g = ggT(a,b))]
```

Es ist zu zeigen:

```
[sp.t1.(GIV ∧ Cond1) ⇒ GIV]
```

Beweis:

```
  sp.t1.(GIV ∧ Cond1)
= { Einsetzen für GIV und Cond1 }
  sp.t1.((a > 0 ∧ b > 0  ⇒  x > 0 ∧ y > 0 ∧ ggT(x,y) = ggT(a,b))
         ∧ (State = finished  ⇒  g = ggT(a,b))
         ∧ State = working ∧ x < y)
= { Einsetzen für sp.t1 }
  (∃z: y = (z-x): (y:=z).
        ((a > 0 ∧ b > 0  ⇒  x > 0 ∧ y > 0 ∧ ggT(x,y) = ggT(a,b))
         ∧ (State = finished  ⇒  g = ggT(a,b))
         ∧ State = working ∧ x < y))
⇒ { Textersetzung ausführen, „Trading“ }
  (∃z:: y = (z-x)
       ∧ (a > 0 ∧ b > 0  ⇒  x > 0 ∧ z > 0 ∧ ggT(x,z) = ggT(a,b))
       ∧ (State = finished  ⇒  g = ggT(a,b))
       ∧ State = working ∧ x < z)
= { „z = y + x“ nutzen }
```

```
   (∃z:: y = (z-x)
        ∧ (a > 0 ∧ b > 0  ⇒  x > 0 ∧ y+x > 0
                             ∧ ggT(x,y+x) = ggT(a,b))
        ∧ (State = finished  ⇒  g = ggT(a,b))
        ∧ State = working ∧ x < y+x)
⇒ { Ausklammern, den Teil mit z weglassen }
   (a > 0 ∧ b > 0  ⇒  x > 0 ∧ y+x > 0 ∧ ggT(x,y+x) = ggT(a,b))
   ∧ (State = finished  ⇒  g = ggT(a,b))
   ∧ State = working ∧ x < y+x
⇒ { arith. umformen, den Teil mit „working“ weglassen }
   (a > 0 ∧ b > 0  ⇒  x > 0 ∧ y+x > 0 ∧ ggT(x,y+x) = ggT(a,b))
   ∧ (State = finished  ⇒  g = ggT(a,b))
   ∧ 0 < y
⇒ { Eigenschaft von ggT (s.o.), Weglassung von „y+x > 0“ }
   (a > 0 ∧ b > 0  ⇒  x > 0 ∧ y > 0 ∧ ggT(x,y) = ggT(a,b))
   ∧ (State = finished  ⇒  g = ggT(a,b))
= { Einsetzen für GIV rückwärts }
   GIV
```

q.e.d.

Der Beweis läßt sich also gut führen. Allerdings ist nachteilig, daß bei der stärksten Nachbedingung der Prädikatentransformator für die Zuweisung sehr aufwendig ist, falls die Variable, auf die zugewiesen wird, auch auf der rechten Seite der Zuweisung vorkommt.

Außerdem kann man mit der stärksten Nachbedingung nicht untersuchen, ob eine Estelle-Transition terminiert. Diese Eigenschaft ist aber aufgrund der Annahme des atomaren Schaltens unbedingt notwendig, und man muß sie untersuchen können, um sicherzustellen, daß die Spezifikation wohldefiniert ist.

(e) Beschreibung einer Estelle-Transition durch einen Prädikatentransformator „schwächste Vorbedingung“

Diese beiden Probleme treten nicht auf, wenn man statt der stärksten Nachbedingung die schwächste Vorbedingung als Prädikatentransformator verwendet ([Dij76], [DiSc90], [BrNe89]).

Von dem Prädikatentransformator „schwächste Vorbedingung“ gibt es, wie in Kapitel 3.4 beschrieben, zwei Versionen: Die „schwächste freie Vorbedingung“ `wlp` und die eigentliche „schwächste Vorbedingung“ `wp`. Mit `wlp` können wir die partielle Korrektheit untersuchen und mit `wp` die totale Korrektheit, also die partielle Korrektheit plus Termination. Dabei ist `wlp` für bestimmte Operationen etwas einfacher aufgebaut und damit einfacher handzuhaben als `wp`, eben weil nicht auch Aussagen über die Termination gemacht werden (siehe dazu auch Kapitel 3.4).

Ist eine Estelle-Spezifikation wohldefiniert, dann ist damit die Termination aller Estelle-Transitionsrümpfe garantiert, und zur Untersuchung aller sonstigen Eigenschaften reicht die partielle Korrektheit. Also sollten wir die Semantik eines Estelle-Transitionsrumpfes mit `wlp` beschreiben.

Aber leider kann man einem Pascal-artigen „Estelle-Transitionsrumpf“ nicht immer gleich ansehen, ob er mit Sicherheit terminieren wird. Daher ist es notwendig, daß wir einen „Spezifikationstext“ daraufhin untersuchen können, ob er tatsächlich wohldefiniert ist.

In Kapitel 3.4 haben wir bereits gesehen, daß weder `wlp` noch `wp` für sich allein in der Lage sind, die Situationen zu beschreiben, in denen ein „Estelle-Transitionsrumpf“ nicht terminiert. Um die Wohldefiniertheit einer Estelle-Spezifikation zu

zeigen, benötigen wir daher `wp` zusammen mit `wlp`. Damit können wir dann für jeden Estelle-Transitionsrumpf formal zeigen, daß er, interpretiert als Pascal-artiges Programm, bei jeder „Ausführung“ terminiert. Gegen Ende von Kapitel 5.1 kommen wir auf diesen Punkt noch einmal zurück. Dann haben wir einen geeigneten Formalismus eingeführt und können die obigen Argumente anhand der Definitionen etwas detaillierter ausführen.[25]

Festzuhalten bleibt aber, daß für die eigentlichen Arbeiten mit der Estelle-Spezifikation, nämlich die Untersuchung ihrer Eigenschaften, der etwas einfachere Prädikatentransformator `wlp` allein ausreicht.

Im Vergleich mit der stärksten Nachbedingung scheint die schwächste Vorbedingung einen konzeptuellen Nachteil zu haben: Mit der stärksten Nachbedingung kann man von einem Zustand zum nächsten schließen, also in die gleichen Richtung, in die die Transitionen des Transitionssystems gerichtet sind, und in die die Ausführung beziehungsweise die Zeit voranschreitet. Mit der schwächsten Vorbedingung kann man nur rückwärts schließen. Die Wirkung einer Estelle-Transition wird nicht mehr direkt beschrieben, sondern nur noch durch die Menge derjenigen Prädikate über den Zustand nach dem Schalten, die durch den Prädikatentransformator `wlp` in gültige Prädikate vor dem Schalten transformiert werden.

Diesem Argument entgegnen wir:

• Die Transitionen sind wohldefiniert. Sie sind Relationen über Zustände, und sowohl die Zustände, aus denen Pfeile abgehen, sind angegeben (durch die Schaltbedingung), als auch die Pfeilführung. In Kapitel 3.4 haben wir gesehen, daß eine Operation durch `wlp` (zusammen mit `wp`) vollständig bestimmt ist.

• Die Untersuchungen in Kapitel 4.2 auf der Basis von [Lam91] haben gezeigt, daß es für die Zwecke der Verifikation im wesentlichen ausreicht, nur jeweils zwei aufeinanderfolgende Zustände zu betrachten. Und dies ist mit der schwächsten Vorbedingung sehr schön möglich.

• Auch bei (a) und (b) bei der Beschreibung durch ein einzelnes Prädikat, zum Beispiel in der TLA von Lamport, galt: Dieses Prädikat mußte nicht so aufgebaut sein, daß der Nachzustand für jeden Vorzustand sofort erkennbar ist. Beispiel: „$\sin(x'^2) = x$“

• Wie wir bei (d) gesehen haben, läßt sich mit der stärksten Nachbedingung die Termination nicht untersuchen, mit der schwächsten Vorbedingung ist dies aber möglich.

• Mit dem formalen Ausführungsmodell wollen wir Spezifikationen in Estelle analysieren können, wir sind folglich an den *Eigenschaften* einer Estelle-Transition interessiert. Daher sollte die Beschreibungsform die Eigenschaften in den Vordergrund stellen, nicht die Möglichkeit der Ausführung. Falls man eine Testimplementation konstruieren will, hat man immer noch den ursprünglichen Estelle-Text.

[25] Anmerkung: Bei der Beschreibung einer Estelle-Transition durch ein Prädikat in Punkt (a) und (b) konnten nur terminierende Estelle-Transitionsrümpfe dargestellt werden, so daß man ihre Wohldefiniertheit in Bezug auf die Termination dort nicht untersuchen konnte. Es gibt auch einen Ansatz, der Operationen auf eine ganz ähnliche Art beschreibt, aber zusätzlich auch Aussagen über Nichttermination machen kann. In [Par83] führt D. Parnas die „Limited-Domain-Relationen“ (LD-Relationen) ein. Dort beschreibt eine Relation zwischen Anfangs- und Endzuständen die möglichen terminierenden Berechungen, und als Erweiterung werden mit Hilfe einer sogenannten „Kompetenzmenge“ auch die nicht terminierenden Berechnungen beschrieben. Leider fehlt bisher ein ausgearbeiteter Formalismus zu diesem Ansatz, und natürlich bleiben die oben genannten Probleme von (a) und (b) auch für diese Variante voll gültig.

Ein nur für die spätere Ausarbeitung unseres Semantikentwurfes bedeutsamer Nachteil der Prädikatentransformatoren ist, daß bisher ausschließlich für die Hoareschen Zusicherungen (fast) der gesamte Sprachumfang von Pascal in ein Kalkül gefaßt wurde. Für alle anderen hier vorgestellten Systeme haben die Autoren immer nur für einige Kernkonstrukte Regeln angegeben, um das Prinzip zu demonstrieren. Die Widrigkeiten von Pascal als einer „gewöhnlichen“, für den praktischen Einsatz entworfenen Sprache ist außer Hoare noch kein Autor angegangen. Allerdings wird man sich zunutze machen können, daß Prädikatentransformatoren eine Weiterentwicklung der Hoareschen Zusicherungen sind.

Wie bei den anderen Punkten auch geben wir nun die Umsetzung der Estelle-Transitionen unseres Beispieltextes aus Kapitel 4.4 in den Formalismus an:

```
Initialize
  Effects: [wlp.Initialize.Y ≡
              (x:=a).((y:=b).((State:=working).Y))]

Trans t1:
  Cond:      State = working
           ∧ x < y
  Effects: [wlp.t1.Y ≡ (y:=y-x).Y)]

Trans t2:
  Cond:     State = working
          ∧ y < x
  Effects: [wlp.t2.Y ≡ (x:=x-y).Y)]

Trans te:
  Cond:      State = working
           ∧ x = y
  Effects: [wlp.te.Y ≡ (g:=x).((State:=finished).Y))]
```

Als Beispiel werden wir jetzt wie bei (d) nachweisen, daß die Transition `t1` die globale Invariante tatsächlich erhält.

Es ist zu zeigen:

```
[(GIV ∧ Cond1) ⇒ wlp.t1.GIV]
```

Beweis:

```
   wlp.t1.GIV
= { Einsetzen für GIV, Cond1 und wlp.t1 }
   (y:=y-x).((a > 0 ∧ b > 0  ⇒  x > 0 ∧ y > 0
             ∧ ggT(x,y) = ggT(a,b))
            ∧ (State = finished  ⇒  g = ggT(a,b)))
= { Textersetzung ausführen }
   (a > 0 ∧ b > 0  ⇒  x > 0 ∧ y-x > 0 ∧ ggT(x,y-x) = ggT(a,b))
   ∧ (State = finished  ⇒  g = ggT(a,b))
= { Eigenschaft von ggT (s.o.), dabei „x>0“ und „y-x>0“ benutzt }
   (a > 0 ∧ b > 0  ⇒  x > 0 ∧ y-x > 0 ∧ ggT(x,y-x+x) = ggT(a,b))
   ∧ (State = finished  ⇒  g = ggT(a,b))
= { arith. Umformung, „x>0“ benutzt }
   (a > 0 ∧ b > 0  ⇒  x > 0 ∧ y > x ∧ y > 0 ∧ ggT(x,y) = ggT(a,b))
   ∧ (State = finished  ⇒  g = ggT(a,b))
⇐ { Umstellen, Logik }
```

```
   (a > 0 ∧ b > 0  ⇒  x > 0 ∧ y > 0 ∧ ggT(x,y) = ggT(a,b))
   ∧ (State = finished  ⇒  g = ggT(a,b))
   ∧ x < y ∧ State = working
= { Einsetzen für GIV und Cond1 rückwärts }
   GIV ∧ Cond1
```

q.e.d.

Wie man sieht, ist der Beweis einfacher geworden, da der Transformator für die Zuweisung viel einfacher ist.

Als nächstes werden wir einen der Beweisschritte, die wir in Kapitel 4.2 beschrieben haben, für die Liveness-Eigenschaft (2) aus Kapitel 4.4 durchführen: Wir werden zeigen, daß die Transition `t1` den Wert der dort definierten Funktion `f` immer verkleinert, wenn sie schaltet.

Die Funktion `f` des Zustandes hatten wir definiert durch:

```
[f(x,y,g,State) = max(0, (x + y) * ind1(State ≠ finished))]
```

Es ist also unter der Randbedingung „`(a>0 ∧ b>0)`“ (und unter Zuhilfenahme der bereits nachgewiesenen Invarianten `GIV`) zu zeigen:

```
[(Cond1 ∧ wlp.t1.(f(x,y,g,State) = n)) ⇒ f(x,y,g,State) > n]
```

(Wobei wir mit `n` eine frische Variable verwendet haben.)

Umgangssprachlich bedeutet diese Formel ungefähr: Wenn die Funktion `f` im Zustand nach dem Schalten der Estelle-Transition `t1` den Wert `n` hat, und wenn im Vorzustand die Schaltbedingung erfüllt war, dann muß das mindestens implizieren, daß die Funktion `f` im Vorzustand einen Wert größer als `n` gehabt hat.

Beweis:

```
   Cond1 ∧ wlp.t1.(f(x,y,g,State) = n)
= { Einsetzen für Cond1, für wlp.t1 und für f }
   State = working ∧ x < y
   ∧ (y:=y-x).(max(0, (x + y) * ind1(State ≠ finished)) = n)
= { Textersetzung durchführen, ind1 auswerten }
   State = working ∧ x < y
   ∧ max(0, y)) = n
= { Gültigkeit von GIV und (a>0 ∧ b>0) ergibt x>0 ∧ y>0 }
   State = working ∧ x < y
   ∧ y = n ∧ x > 0 ∧ y > 0
⇒ { Hinzufügen der Def. von f }
   State = working ∧ x < y
   ∧ y = n ∧ x > 0 ∧ y > 0
   ∧ f(x,y,g,State) = max(0, (x + y) * ind1(State ≠ finished))
= { arith. Umformung, ind1 auswerten }
   State = working ∧ x < y
   ∧ y = n ∧ x > 0 ∧ y > 0
   ∧ f(x,y,g,State) = x + y
= { arith. Umformung, Weglassen }
   f(x,y,g,State) > n
```

q.e.d.

Anmerkung: Wie bereits früher kurz angedeutet, ist es im Prinzip auch möglich, die

Schaltbedingung einer Transition mit in den Prädikatentransformator aufzunehmen. Dazu interpretieren wir die Schaltbedingung als „Wächter“ (englisch „guard“) für den nachfolgenden Transitionsblock (notiert als „`b -> S`“, siehe auch [BrNe89]). Die Menge der Berechnungsfolgen eines derartigen „guarded command“ ist die Menge derjenigen Berechnungsfolgen des Blocks, bei denen im Anfangszustand das „Guard“ erfüllt ist. Ist das „Guard“ nicht erfüllt, kann keine Berechnung starten. (Damit die Relation von Berechnungsfolgen nicht mehr immer total, das „Gesetz vom ausgeschlossenen Wunder ([$wp.S.False \equiv False$])“, siehe Kapitel 3.4, gilt nicht mehr.)

Beispiel:

$$\textbf{Trans}\ t1:\quad [wlp^{+}.t1.Y \equiv \neg(State = working \wedge x < y) \vee (y:=y-x).Y)]$$

In den Safety-Eigenschaften ist dann anstelle der Formel weiter oben für die Transition `t1` nur noch zu beweisen:

$$[GIV \Rightarrow wlp^{+}.t1.GIV]$$

Denn diese Formel ergibt sich sofort aus der anderen für `wlp` durch:

$$[wlp^{+}.t.Y \equiv \neg Cond \vee wlp.t.Y]$$

Leider läßt sich aber allein mit wlp^{+} der Beweis der Liveness-Eigenschaften nicht mehr führen, da sich die Bedingung nicht ausdrücken läßt, daß der Wert der Funktion `f` nach jedem Schalten kleiner wird, wenn vor dem Schalten die Schaltbedingung erfüllt war. (Man vergleiche die Formeln.)

Damit ist diese Alternative leider nicht brauchbar, und wir führen die Schaltbedingung weiterhin getrennt mit an.

Im Zusammenhang mit der Darstellung von Estelle-Transitionen durch ein Prädikat unter Punkt (a) und (b) ergaben sich zwei Probleme, von denen zu untersuchen bleibt, ob sie auch im Zusammenhang mit dem Prädikatentransformator „schwächste Vorbedingung“ auftreten.

Das erste Problem trat nur bei (a) auf: Dort wurde festgelegt, daß die Werte aller im Prädikat nicht erwähnten Variablen von der Transition nicht verändert werden. Dies schuf das Problem, daß eine Aussage über eine Menge von Variablen gemacht wurde, die überhaupt erst einmal scharf definiert werden muß. Als Folge wurde die Logik komplizierter, da das Prädikat „$(z' = z')$“ nicht mehr immer wahr ist (siehe oben).

Für Prädikatentransformatoren gilt: Das entsprechende Prädikat „$(z = z)$“ ist in der zugehörigen Logik immer wahr, und es wird auf die gleiche Weise transformiert wie das Prädikat „wahr“. Wie wir in Kapitel 3.4 gesehen haben, gilt trotzdem für die Transformatoren, die zur Beschreibung der Semantik von Pascal verwendet werden, daß die im Transformator nicht erwähnten Variablen unverändert bleiben, da ein entsprechendes Prädikat „$(z = z_1)$“ dort in sich selbst transformiert wird. Nur bei Transformatoren, die Nichttermination oder noch größere Verheerungen (englisch „havoc“) beschreiben, erhält man andere Ergebnisse. Zum Beispiel erzeugt Dijkstras Transformator `Havoc` einen völlig unbestimmten Gesamtsystemzustand.

Was bei der Darstellung durch Prädikate also nur wahlweise zu haben war, ist bei Prädikatentransformatoren vereint: Die Nichterwähnung unveränderter Variablen und die Tautologie „$(z = z)$“.

An diesem Punkt wies uns Leslie Lamport darauf hin ([Lam91a]), daß ein weiteres, verwandtes Problem aber immer noch bestehen bleibt:

Wir wollen auf Grundlage der Prädikatentransformatoren eine Logik konstruieren, mit der wir die Semantik von Estelle-Spezifikationen ausdrücken können und mit der wir Schlußfolgerungen über diese ziehen können. Eines der Gesetze einer Logik sollte sein, daß eine Formel `F`, wenn eine Variable `x` in ihr nicht frei vorkommt, äquivalent sein sollte zu `(∀x:: F)`. Dabei definiert man üblicherweise das freie Vorkommen einer Variablen als ihr textuelles Enthaltensein (falls sie nicht durch eine Quantifikation gebunden ist). Wenn wir nun den üblichen Prädikatentransformator für die Pascal-Anweisung „`y := y+1`“ betrachten, nämlich die Textersetzung `(y:=y+1)`, dann ergibt sich damit, daß `x` darin nicht frei vorkommt. Trotzdem macht dieser Transformator die Aussage, daß die Variable `x` nicht verändert wird. Wenn wir ihn in eine Logik integrieren und dadurch eine Formel der Art `(∀x:: (y:=y+1))` erhalten, dann ist diese offensichtlich nicht äquivalent zu einer Formel `(y:=y+1)`.

Aber auch dieses Problem läßt sich lösen: In Kapitel 5 werden wir Prädikatentransformatoren auf eine solche Weise in eine Logik integrieren, daß die Variablen, die sie beeinflussen, immer explizit aufgezählt werden. Über alle anderen Variablen machen sie dann keinerlei Aussagen.

Noch eine weitere Lösung wäre denkbar: Man könnte sich auf solche Prädikatentransformatoren beschränken, für die sich auf eine sinnvolle Art und Weise die Menge der freien Variablen derart definieren läßt, daß sie leicht erkennbar ist. In Kapitel 3.5 haben wir unsere neuen Prädikatentransformatoren `Change` und `Same` vorgestellt. Die Menge ihrer freien Variablen, beziehungsweise das Mengenkomplement dazu, wäre direkt aus ihren Parametern ablesbar. Weitere Prädikatentransformatoren ließen sich durch Zusammensetzung aus den herkömmlichen Prädikatentransformatoren und `Same` konstruieren. Auf dieser Grundlage ließe sich dann ebenfalls eine Logik definieren, die Lamports Forderung erfüllt. Aber in Kapitel 5.1 werden wir sehen, daß die dortige Lösung doch etwas günstiger ist.

Kommen wir zum zweiten Problem, das sich unter Punkt (a) und (b) ergeben hatte:

Dort ließ sich die sequentielle Komposition schlecht darstellen, insbesondere die Kombination mit der allgemeinen Iteration schuf Probleme.

Der Prädikatentransformator `wlp` ist für die Beschreibung der Semantik sequentieller Sprachen geschaffen worden, und so wird die sequentielle Komposition durch ihn sehr elegant behandelt. Es wird einfach der Prädikatentransformator des zweiten Programmstücks in den Prädikatentransformator des ersten Programmstücks eingesetzt:

$$\texttt{wlp."x:=y; x:=x+1".Y} \equiv \texttt{wlp."x:=y".(wlp."x:=x+1".Y)}$$

Damit ist der Prädikatentransformator `wlp` für diesen sehr wichtigen Anwendungsfall sehr gut geeignet. (Und für `wp` gilt in diesem Zusammenhang immer das Gleiche wie für `wlp`.) Jetzt bleibt die allgemeine Iteration zu untersuchen.

Die allgemeine **`do`**`..`**`od`**-Schleife wird von `wlp` durch eine unendliche Konjunktion in Form einer Allquantifizierung dargestellt. Sie ist definiert durch (siehe Kapitel 3.4):

$$[\texttt{wlp.DO.X} \equiv (\forall \texttt{i}:\ 0 \leq \texttt{i}:\ (\texttt{wlp.IF})^{\texttt{i}}.(\texttt{B} \vee \texttt{X}))] \qquad \text{für alle } \texttt{X}$$

(Man beachte, daß in Dijstras Notation „∀“ nicht einen boolschen Wert, sondern eine boolsche „Struktur“ liefert, hier eben die Konjunktion.)

Für `wp.DO` gilt entsprechend (sofern `wp.S` „oder-kontinuierlich“ ist):

$$[\texttt{wp.DO.X} \equiv (\exists \texttt{i}:\ 0 \leq \texttt{i}:\ \texttt{k}^{\texttt{i}}.\texttt{false})] \qquad \text{für alle } \texttt{X}$$

mit

$$[\texttt{k.Y} \equiv (\texttt{B} \vee \texttt{X}) \wedge (\neg\texttt{B} \vee \texttt{wp.S.Y})]$$

Diese unendliche Konjunktion beziehungsweise Disjunktion muß bei der späteren Verwendung in Korrektheitsbeweisen von Hand analysiert und aufgelöst werden. Sofern die Termination garantiert ist, ist es (für oder-kontinuierliche `wlp.IF`) immer möglich, eine obere Grenze für die Zahl der Iterationen anzugeben, die ihrerseits die Zahl der Konjunktions- beziehungsweise Disjunktionsglieder bestimmt ([DiSc90], S. 186). Da die Aufgabe der Analyse das bekannte Halteproblem ([Eng88]) enthält, welches nicht Turing-berechenbar ist, war die Existenz einer automatischen Analysemethode auch nicht zu erwarten.

Folglich ist im Zusammenhang mit der Verifikation einer allgemeinen **do..od**-Schleife in jedem beliebigen Formalismus immer an mindestens einer Stelle „Handarbeit“ notwendig. Interessant ist nur, an welcher Stelle.

In der praktischen Anwendung wird in einer Estelle-Transition eine allgemeine **do..od**-Schleife nur sehr selten vorkommen. Wenn überhaupt Schleifen vorkommen, läßt sich die maximale Zahl der Durchläufe meist leicht bestimmen, weil dann typischerweise eine feststehende Menge von Objekten manipuliert wird. Entscheidend ist, daß die Semantik überhaupt automatisch in eine geschlossene formale Form gebracht werden kann und der volle Ausdrucksumfang von Estelle grundsätzlich abgedeckt ist.

Die „Handarbeit“ wird erst in Beweisführungen über allgemeine **do..od**-Schleifen notwendig, also an der spätestmöglichen Stelle, dann, wenn sie aus den erwähnten prinzipiellen Gründen absolut unumgänglich ist.

Damit ist die Darstellung von Estelle-Transitionen durch Prädikatentransformatoren auch hier der Darstellung durch Prädikate überlegen. Wie bereits ausgeführt, ist bei letzterer ja bereits jede sequentielle Komposition umständlich (wobei von dieser Ausdrucksmöglichkeit in fast jeder Estelle-Transition intensiv Gebrauch gemacht wird).

(f) Eine Mischform aus einigen der anderen Darstellungsformen

Bei Mischformen muß es im wesentlichen darum gehen, die Darstellung durch Prädikate *PP* (a) und (b) mit der Darstellung durch Prädikatentransformatoren *PT* (c) bis (e) zu kombinieren. (Klassifikation wie oben nach [ZwRo89].)

Dazu sehen wir zwei Möglichkeiten. Die erste ist:

Hat man wie bei (e) den Prädikatentransformator `wlp` für eine Estelle-Transition aus der Textdarstellung erhalten, so kann man auf die Idee kommen, ein Prädikat hineinzustecken, daß den Gesamtzustand vollständig beschreibt. Damit bekommt man ein Prädikat aus dem Transformator heraus, das die Wirkung der Estelle-Transition vollständig beschreibt. (Solch ein Prädikat wird im Kontext der TLA allerdings „Aktion“ genannt.) Beispiel:

```
  wlp."x:=x+1".(x'=x ∧ y'=y ∧ z'=z)
= { Einsetzen }
  x'=x+1 ∧ y'=y ∧ z'=z
```

Dies ist auch in der Tat durchführbar. Allerdings verlieren wir damit einen Vorteil des Prädikatentransformators. Die Schlußregel zum Nachweis einer invarianten Eigenschaft (Safety-Eigenschaft) kann in einer auf Prädikatentransformatoren aufbauenden Logik etwa so lauten (für nur eine Transition und ohne Berücksichtigung einer Schaltbedingung):

$$\frac{\texttt{I} \Rightarrow \texttt{wlp.S.I}}{\texttt{I} \wedge \texttt{"S ist einzige Estelle–Transition"} \Rightarrow \Box\texttt{I}}$$

Wie man sieht, ist die Prämisse der Schlußregel äußerst einfach. In der TLA lautet die entsprechende Schlußregel:

$$\texttt{INV1:}\quad \frac{\texttt{I} \wedge [N]_{\texttt{f}} \Rightarrow \texttt{I}'}{\texttt{I} \wedge \Box[N]_{\texttt{f}} \Rightarrow \Box\texttt{I}}$$

Setzt man für die Aktion N das aus dem Prädikatentransformator gewonnene Prädikat ein, erhält man:

$$\frac{\texttt{I} \wedge [\texttt{wlp.S.(f'=f)}]_{\texttt{f}} \Rightarrow \texttt{I}'}{\texttt{I} \wedge \texttt{"S ist einzige Estelle–Transition"} \Rightarrow \Box\texttt{I}}$$

Wie man sieht, ist die Prämisse der Schlußregel hier etwas aufwendiger. Da sämtliche Komponenten des Zustands in `f` enthalten sind, müssen sie alle durch `wlp.S` transformiert werden, was ebenfalls aufwendiger ist. Auch die restliche boolsche Formel ist etwas komplizierter und erfordert folglich mehr Umformungsaufwand zu ihrem Nachweis.

Anmerkung: Hat man mehrere Estelle-Transitionen, so muß die Schlußregel um gleichartige Prämissen für jede Transition erweitert werden. Erst wenn die oben angeführte Prämisse für jede der Transitionen nachgewiesen ist, kann eine Schlußfolgerung entsprechend der obigen gezogen werden. Und die Schaltbedingungen von Estelle-Transitionen lassen sich ebenfalls leicht ergänzen:

`wlp.S.X` wird durch `(¬Cond ∨ wlp.S.X)` ersetzt.

Aus `(I ⇒ wlp.S.I)` wird damit `((I ∧ Cond) ⇒ wlp.S.I)` .

Kommen wir nun zur zweiten Möglichkeit, Prädikate und Prädikatentransformatoren zu kombinieren:

Es ist denkbar, ein Kalkül zu konstruieren, in dem Prädikatentransformatoren und Prädikate gleichberechtigt nebeneinander stehen, indem das eine wechselweise auch als das andere interpretiert werden kann.

Beispielsweise verfolgt [ZwRo89] diesen Ansatz und präsentiert die Sprache MCL (mathematical core language), auf der wiederum die industrielle Spezifikations- und Entwurfssprache COLD basiert, die in den Philips Forschungslabors verwendet wird.

In MCL sind Prädikate, die schwächste Vorbedingung wp und Zusicherungen nach Hoare vereinigt. Es gibt eine ganze Reihe von Schlußregeln, darunter auch einige, mit denen die verschiedenen Operatoren ineinander transformiert werden können. Zur Anwendung dieser Regeln ist allerdings menschliche Mithilfe notwendig, um die Struktur der Prämissen in geeigneter Weise auf die Struktur von bereits bewiesenen Sätzen abzubilden.

Folglich ist es zwar auf der Basis von MCL möglich, wie unter (e) beschrieben eine Estelle-Transition als einen Prädikatentransformator wlp zu beschreiben, aber bei einer Darstellung oder Transformation in eine andere Form erhalten wir unvermeidlicherweise auch deren beschriebene Nachteile. Darüberhinaus stehen für eine eventuelle Transformation zwar Schlußregeln bereit, aber die durch die Transforma-

tion entstehenden Probleme sind nur formal eingefangen und müssen weiterhin von Hand bei der Anwendung der Schlußregel gelöst werden.

Für die *Beschreibung* von Estelle-Transitionen bringt diese Art von Kombination der verschiedenen Ansätze also nichts Neues. Ebenfalls sehen wir keinen Grund, für unsere Aufgabe die explizite Transformation zwischen den Beschreibungsformen einzuführen. Sie würde uns wie gesehen nur zusätzliche Schwierigkeiten eintragen, ohne Nutzen zu bringen.

Und doch ist eine bestimmte Art von kombiniertem Ansatz sinnvoll: Mit der TLA haben wir ein fertiges, gutes Schlußsystem auf PP-Basis, mit dem wir über die temporalen Aspekte schließen können, das aber die Estelle-Transitionsrümpfe nicht gut beschreiben kann. Mit der schwächsten Vorbedingung haben wir auf PT-Basis eine gute Beschreibungsmöglichkeit für Estelle-Transitionsrümpfe, die aber ganze Transitionssysteme und ihre temporalen Eigenschaften nicht ausdrücken kann.

Daher wäre es schön, diese beiden Formalismen derart zu kombinieren, daß wir sowohl die Estelle-Transitionsrümpfe beschreiben können, als auch temporale Schlußregeln nach Art der TLA nutzen können. Eine Transformation zwischen PP-Beschreibung und PT-Beschreibung muß dabei nicht explizit als Schlußregel anwendbar sein, sie kann auch nur implizit in der Definition des Formalismus′ vorhanden sein.

Einen derartigen Ansatz werden wir in Kapitel 5 vorstellen, wenn wir unsere „*temporale Logik der Aktionen, mit Prädikatentransformatoren*“ (TLA/PT) skizzieren.

4.6 Zusammenfassung der Diskussion

Eine Estelle-Spezifikation wollen wir im Grundsatz operational darstellen, also durch Angabe eines abstrakten Automaten beziehungsweise Transitionssystems. Die Formalisierung soll insbesondere dazu geeignet sein, die Verifikation von Estelle-Spezifikationen zu unterstützen.

Voraussetzungen für formales Schließen über Estelle-Spezifikationen sind

- eine geeignete formale Beschreibung der nachzuweisenden Eigenschaften,
- eine geeignete Formalisierung der Relation von aufeinanderfolgenden Zuständen
- sowie ein geeignetes formales Schlußsystem.

Wir untersuchten, welche Beweisverfahren das formale Schließen unterstützen muß, und fanden heraus, daß diejenigen ausreichen, die auch Lamport in seiner temporalen Logik der Aktionen TLA ([Lam91]) formalisiert hatte. In dieser Beschreibungstechnik lassen sich auch wie gefordert Eigenschaften ausdrücken, die verifiziert werden sollen.

Daraufhin blieb noch zu untersuchen, wie die Zustandsübergangsrelation des abstrakten Automaten darzustellen ist. Sie wirft die meisten Probleme bei der Formalisierung von Estelle auf, da sie zu einem erheblichen Teil durch die Pascal-artigen Rümpfe der Estelle-Transitionen beschrieben wird. Wir fanden heraus, daß es angemessen ist, im Rahmen unserer Aufgabenstellung Estelle-Transitionen mit prädikatenlogischen Mitteln darzustellen. Anschließend stellten wir die Bedingungen auf, die die Darstellung erfüllen muß, und untersuchten dann die vorhandenen Formalismen dieser Art ausführlich auf diese Bedingungen hin.

Dabei ergab sich im einzelnen:

Zur Darstellung von Estelle-Transitionen als jeweils einzelnes Prädikat existiert bereits ein fertiger, guter Formalismus, die TLA. Mit ihr ist formales Schließen über temporale Eigenschaften eines Transitionssystems sehr schön möglich. Leider ist dieser Ansatz nur zur Darstellung abstrakter Algorithmen und zu ihrer Verfeinerung geeignet, nicht aber zur Analyse sequentiell ausgedrückter Rümpfe von Estelle-Transitionen.

Eine einzelne Zusicherung nach Hoare beschreibt einen Estelle-Transitionsrumpf im allgemeinen nicht in ihrer gesamten Wirkung. Sie ist damit zur Definition der Semantik der Estelle-Transitionsrümpfe nicht zu gebrauchen. Erzwingt man aber eine vollständige Wirkungsbeschreibung, fällt dieser Ansatz mit dem vorgenannten zusammen.

Explizite Prädikatentransformatoren nach Dijkstra sind gut geeignet, die Wirkung von Estelle-Transitionsrümpfen zu definieren. Dabei erwies sich die „schwächste Vorbedingung“ ,`wlp` und `wp`, als handlicher und ausdrucksstärker als die „stärkste Nachbedingung“, `sp`. Allerdings ist dieser Ansatz *nur* für transformationelle Systeme[26] geeignet, (reaktive) Transitionssysteme und ihre temporalen Eigenschaften lassen sich mit den Prädikatentransformatoren nicht behandeln.

Mischformen in der Darstellung bringen keinen weiteren Nutzen für die Definition der Semantik von Estelle-Transitionsrümpfen.

Aber mit einer Mischform erreichen wir alles, was wir für die Gesamt-Definition benötigen. Denn mit ihr können jeweils die Eigenschaften der einen Beschreibungsform die fehlenden der anderen ergänzen:

Die Vorteile der TLA und der Prädikatentransformatoren lassen sich kombinieren, und zwar einerseits die Eignung für die Verifikation von Transitionssystemen, die

[26] Zu den Begriffen „transformationell“ und „reaktiv“ siehe Kapitel 3.1.

Beschreibbarkeit von Eigenschaften auf einer höheren Abstraktionsebene und das fertige Schlußsystem dazu und andererseits die Darstellbarkeit der Estelle-Transitionsrümpfe und die Fähigkeit, deren Eigenschaften zu verifizieren.

Auf diese Weise werden alle drei der eingangs genannten Bedingungen für formales Schließen über Estelle-Spezifikationen durch einen einzigen Formalismus erfüllt.

Zur Realisierung einer Mischform wäre es möglich, die Prädikatentransformatoren in TLA-Aktionen zu übersetzen. Aber es stellte sich als günstiger heraus, die Prädikatentransformatoren direkt in die TLA zu integrieren, da sich dann Korrektheitsbeweise leichter führen lassen. Daher werden wir diesen Weg in Kapitel 5 skizzieren, um eine

„*temporale Logik der Aktionen,*
mit Prädikatentransformatoren“ (TLA/PT)

zu erhalten.

5. TLA/PT: Erweiterung der TLA um Prädikatentransformatoren

Wie in Kapitel 4.3 begründet, wollen wir Prädikatentransformatoren verwenden, um die Bedeutung von Estelle-Transitionen auszudrücken. Da wir aber die übrige Estelle-Spezifikation in der von Lamport definierten TLA ausdrücken wollen, müssen wir die Prädikatentransformatoren in die TLA integrieren, um die Bedeutung einer Estelle-Spezifikation vollständig in einem einzigen Formalismus ausdrücken zu können.

Dazu müssen wir die Syntax, die Bedeutungsfunktion und einige der „zusätzlichen Notationen“[27] der TLA entsprechend erweitern. Damit man mit dem Formalismus schließlich auch praktisch verifizieren kann, müssen die Schlußregeln entsprechend erweitert werden. Da dies aber den Rahmen dessen, was für unsere Arbeit zur Definition des Ausführungsmodells von Estelle benötigt wird, übersteigt, werden wir nur die zentrale neue Schlußregel einführen. Und in Ermangelung des vollständigen Satzes von Schlußregeln werden wir auch nicht mehr beweisen, daß das erweiterte Kalkül konsistent und vollständig ist.

Das neue Kalkül nennen wir TLA/PT, als Abkürzung für „**T**emporale **L**ogik der **A**ktionen mit **P**rädikaten**t**ransformatoren“.

5.1 Syntax und Semantik der TLA/PT

Die vollständige erweiterte Syntax findet sich in Abbildung 5-1, man vergleiche sie mit der Syntax der TLA in Kapitel 3.3.

Die syntaktischen Konstrukte <*Aktion*> und <*Zustandsfunktion*> werden um Prädikatentransformatoren erweitert. Dadurch wird wiederum eine weitere kleine Anpassung notwendig, um sicherzustellen, daß auf das Prädikat „*Enabled* <*Aktion*>“ kein Prädikatentransformator angewandt werden kann.

Weiterhin wird gegenüber der Syntax aus [Lam91] die Existenzquantifikation „∃“ auch innerhalb von <*Aktion*>en und <*Zustandsfunktion*>en zugelassen, so daß sie nicht nur „ganz außen“ um alle temporalen Operatoren herum stehen kann, sondern auch „ganz innen“, innerhalb aller temporalen Operatoren.

Hiermit erreichen wir allerdings keine zusätzliche Ausdruckskraft, sondern nur Anordnungsvorteile bei der Formulierung des Ausführungsmodells. Denn es gilt zum Beispiel:

```
  ◇((∃i:: P(i)) ⇒ (∃j:: Q(j)))
= (∃i:: (∃j:: ◇(P(i) ⇒ Q(j))))
```

Zu der gewählten Notation für die TLA/PT müssen wir noch ein Wort verlieren, genauer gesagt zur Notation von Funktionsanwendungen. Dijkstra hat für das Kalkül der Prädikatentransformatoren durchgehend die Punktnotation „`f.b`“ für die Anwendung von einstelligen Curry-Funktionen verwendet (siehe Kapitel 3.4). Dies hat den Vorteil, daß man bei der Verkettung vieler Funktionen, wie sie im Zusammenhang mit Prädikatentransformatoren häufig ist, keine tief geschachtelten,

[27] Auch die TLA enthält bereits „zusätzliche Notationen“, für die die Bedeutung nicht mehr gesondert definiert werden muß (siehe Kapitel 3.3), zum Beispiel:

$\Diamond F \triangleq \neg\Box\neg F$

$F \rightsquigarrow G \triangleq \Box(F \Rightarrow \Diamond G)$

Abbildung 5-1

$<\textit{allgemeine Formel}> \triangleq$
$\quad <\textit{Formel}> \mid (\exists <\textit{Variable}>::\ <\textit{allgemeine Formel}>)$
$\quad \mid (\exists <\textit{starre Variable}>::\ <\textit{allgemeine Formel}>)$
$\quad \mid <\textit{allgemeine Formel}> \wedge <\textit{allgemeine Formel}>$
$\quad \mid \neg <\textit{allgemeine Formel}>$

$<\textit{Formel}> \triangleq <\textit{Prädikat}> \mid \Box[<\textit{Aktion}>]_{<\texttt{Zustandsfunktion}>} \mid \neg <\textit{Formel}>$
$\quad \mid <\textit{Formel}> \wedge <\textit{Formel}> \mid \Box <\textit{Formel}>$

$<\textit{Aktion}> \triangleq$ boolwertiger Ausdruck, der enthalten kann:
Konstanten, Variablen, gestrichene Variablen,
„$<\textit{Prädikatentransformator}>.<\textit{einfaches Prädikat}>$“
und „$(\exists <\textit{Variable}>::\ <\textit{Aktion}>)$“

$<\textit{Prädikat}> \triangleq <\textit{einfaches Prädikat}> \mid \textit{Enabled}\ <\textit{Aktion}>$

$<\textit{einfaches Prädikat}> \triangleq$ bool-wertige $<\textit{Zustandsfunktion}>$

$<\textit{Zustandsfunktion}> \triangleq$
Ausdruck, der enthalten kann:
Konstanten, Variablen,
„$<\textit{Prädikatentransformator}>.<\textit{einfaches Prädikat}>$“
und „$(\exists <\textit{Variable}>::\ <\textit{Zustandsfunktion}>)$“

$<\textit{Prädikatentransformator}> \triangleq$
ein Prädikatentransformator wie bei Dijkstra

Abbildung 5-1: Vollständige Syntax der TLA/PT

unübersichtlichen Klammerausdrücke erhält.

Im Rahmen der TLA haben wir andererseits die sonst übliche Notation „`f(a,b)`“ mit Klammern um die Argumente. Diese Notation ist von Vorteil, wenn der Leser durch eine syntaktische Hilfe an die Art und Zuordnung der Parameter erinnert werden soll. Zum Beispiel ließe sich ein dreistelliger Operator, wie der in wenigen Augenblicken eingeführte Existenzquantor[28] „`(∃_:_:_)`“, sicherlich auch nach Curry in einstellige höhere Funktionen auflösen, aber das Ergebnis wäre unübersichtlich.

Indem wir beide Formalismen in der TLA/PT kombinieren, müssen wir eine Lösung für die vereinte Notation finden. Für den TLA-Teil scheint es uns nicht sinnvoll möglich, Dijkstras Curry-Notation zu übernehmen. Andererseits ist diese nach wie vor vorteilhaft bei der Verkettung von Prädikatentransformatoren. Daher belassen wir jeden der beiden Teile in seiner bisherigen Notationsform.

Für die erweiterten syntaktischen Konstrukte definieren wir nun erweiterte Bedeutungen. Die folgenden Semantikdefinitionen in der TLA

$\texttt{s}[\![\texttt{f}]\!] \triangleq \texttt{f}(\forall\texttt{"v":}\ \texttt{s}[\![\texttt{v}]\!]/\texttt{v})$
$\texttt{s}[\![A]\!]\texttt{t} \triangleq A(\forall\texttt{"v":}\ \texttt{s}[\![\texttt{v}]\!]/\texttt{v},\ \texttt{t}[\![\texttt{v}]\!]/\texttt{v}')$

werden in der TLA/PT ersetzt durch:

[28] Genau genommen handelt es sich hier gar nicht um eine Standard-Präfix-Funktionsnotation. Aber dieses Beispiel einer Klammer-Schreibweise zeigt den Effekt besonders deutlich.

$$\mathtt{s}[\![\mathtt{f}]\!] \triangleq (\mathtt{pt_eval}(\mathtt{f}))(\forall\text{"v"}: \mathtt{s}[\![\mathtt{v}]\!]/\mathtt{v})$$
$$\mathtt{s}[\![A]\!]\mathtt{t} \triangleq (\mathtt{pt_eval}(A))(\forall\text{"v"}: \mathtt{s}[\![\mathtt{v}]\!]/\mathtt{v},\ \mathtt{t}[\![\mathtt{v}]\!]/\mathtt{v}')$$

Um nicht einen aufwendigen Algorithmus angeben zu müssen, der „alle inneren Existenzquantoren nach vorne zieht“, führen wir zusätzlich die folgenden Semantikdefinitionen für die erweiterten Existenzquantoren ein:

$$\mathtt{s}[\![(\exists\mathtt{x}::\ \mathtt{f})]\!] \triangleq (\exists\alpha:\ \alpha \in \mathbf{St} \wedge \alpha =_{\mathtt{x}} \mathtt{s} \wedge \alpha[\![\mathtt{f}]\!])$$

$$\mathtt{s}[\![(\exists\mathtt{x}::\ A)]\!]\mathtt{t} \triangleq (\exists\alpha,\beta:\ \alpha,\beta \in \mathbf{St} \wedge \alpha =_{\mathtt{x}} \mathtt{s} \wedge \beta =_{\mathtt{x}} \mathtt{t} \wedge \alpha[\![A]\!]\beta)$$

wobei
$[\![\ ]\!]$ die zu definierende Bedeutungsfunktion ist,
`s`,`t` Zustände sind,
`f` eine <*Zustandsfunktion*> ist,
A eine <*Aktion*> ist,
`x` eine <*Variable*> ist,
St die Menge aller Zustände ist,
$(\forall\text{"v"}: \ldots/\mathtt{v},\ \ldots/\mathtt{v}')$ die Ersetzung für alle Variablen `v` bezeichnet und
`pt_eval` eine Funktion ist, die gleich erläutert werden wird.

Im übrigen erinnern wir uns an die folgende Definition aus Kapitel 3.3:

$$\mathtt{s} =_{\mathtt{x}} \mathtt{t} \triangleq (\forall\text{"v"} \neq \text{"x"}: \mathtt{s}[\![\mathtt{v}]\!] = \mathtt{t}[\![\mathtt{v}]\!])$$

Mit der erweiterten Semantikdefinition von „$\exists$x“ für eine Zustandsfunktion `f` drücken wir in etwa folgendes aus: „Es gibt einen Zustand α, der dem aktuellen Zustand `s` bis auf den Wert der Variablen `x` gleicht, und an dieser Stelle besitzt der Zustand α einen geeigneten Wert für `x`, mit dem die Bedeutung der Zustandsfunktion `f` den Wert `True` hat.“ Wenn es einen solchen Zustand α nicht gibt, ist der Wert des Ausdrucks entsprechend „`False`“.

Für Aktionen ist die Definition ganz analog, nur daß dort die Existenz sowohl eines passenden Vor- wie eines passenden Nachzustandes behauptet wird. Alles in allem handelt es sich also um die übliche Definition von „$\exists$x“. Nur Lamports Definition von „$\exists$x“ für <*allgemeine Formel*>n war aufwendiger, weil sie sogenannte Stotterschritte richtig behandeln mußte.

<u>pt_eval</u>

`pt_eval` ist eine Funktion über einer Menge von bestimmten Zeichenketten, und zwar von <*Zustandsfunktion*>en nach <*Zustandsfunktion*>en und von <*Aktion*>en nach <*Aktion*>en.

`pt_eval` wertet alle in ihrem Argument enthaltenen Anwendungen von Prädikatentransformatoren auf Prädikate aus und ersetzt sie durch den Wert des Prädikatentransformators. `pt_eval` ist eindeutig und total, da auch Prädikatentransformatoren eindeutige und totale Funktionen sind.

Wir verzichten hier darauf, `pt_eval` für jeden Prädikatentransformator zu definieren, und verweisen nur auf [DiSc90], wo alle wichtigen Prädikatentransformatoren definiert werden, und auf unser Kapitel 3.4, wo diese Definitionen zusammengefaßt sind, sowie auf Kapitel 3.5.

Dafür geben wir ein kleines Beispiel für `pt_eval` und eine Anwendung dazu an:

$$[\mathtt{pt_eval}((\mathtt{x:=y}).(\mathtt{x=42})) \equiv (\mathtt{y=42})]$$

$\mathtt{s}[\![\,(\mathtt{x:=y}).(\mathtt{x=42})]\!] \equiv \mathtt{s}[\![\,(\mathtt{y=42})]\!] \equiv (\mathtt{s}[\![\,\mathtt{y}]\!]\mathtt{=42})$

(Dabei ist ≫`(v:=e).X`≪, wie in Kapitel 3.4 erläutert, der Prädikatentransformator für die Zuweisungsoperation ≫`"v:=e"`≪. Dieser Prädikatentransformator ersetzt im Prädikat `X` den Text ≫`v`≪ durch den Text ≫`e`≪.)

Der Grund für die Einführung von `pt_eval` liegt darin, daß die Prädikatentransformatoren *vor* der Bedeutungsfunktion auf die Variablennamen angewandt werden sollen. Denn anderenfalls wäre in einem Zustand `s` mit `x = 7` und `y = 42`:

$\mathtt{s}[\![\,(\mathtt{x:=y}).(\mathtt{x=42})]\!] \;\; ?\!\equiv\!? \;\; (\mathtt{x:=y}).(\mathtt{7=42}) \equiv (\mathtt{7=42}) \equiv \mathtt{False}$

Die Prädikatentransformatoren hätten keine Gelegenheit mehr, Variablennamen zu ersetzen. Die Anwendung der Bedeutungsfunktion können wir aus der Sicht Dijkstras als Auswahl eines Punktes des Zustandsraumes ansehen. Folglich könnten wir danach mit den Prädikatentransformatoren nicht mehr Operationen beschreiben, die den Zustandsraum transformieren, wobei die Koordinatenachsennamen dieses Zustandsraumes Variablennamen sind. Genau dies ist es aber, was wir wollen, und daher haben wir oben `pt_eval` eingeführt.[29]

Falls nun der Eindruck entstanden sein sollte, daß mit der Evaluierungsfunktion `pt_eval` gegenüber der TLA etwas grundsätzlich Neues eingeführt wird, so täuscht dies. Auch die TLA benötigt bereits eine Evaluierungsfunktion `eval`, um im Kontext der Zustandsfunktionen den Wert eines arithmetischen Ausdrucks über Werten zu definieren. In Kapitel 3.3 haben wir dies in einer längeren Fußnote bereits erläutert. Für die TLA/PT sind sowohl `pt_eval` wie auch `eval` notwendig.

Dijkstras „Überall"-Operator „`[ ]`" fassen wir in diesem Zusammenhang ebenfalls als Prädikatentransformator auf, so daß er von `pt_eval` aufgelöst wird. Er erzeugt eine Konjunktion zwischen allen Komponenten von boolwertigen Tupeln, und vor allem erzeugt er eine Allquantifikation über alle in seinem Argument enthaltenen Variablen.

Um die Allquantifizierung verwenden zu können, führen wir nebenbei weitere „zusätzliche Notationen" ein:

$(\forall \mathtt{v}::\ \mathtt{F}) \triangleq \neg(\exists \mathtt{v}::\ \neg\mathtt{F})$
$(\forall \mathtt{c}::\ \mathtt{F}) \triangleq \neg(\exists \mathtt{c}::\ \neg\mathtt{F})$
$(\forall \mathtt{v}:\ \mathtt{F}:\ \mathtt{G}) \triangleq (\forall \mathtt{v}::\ \mathtt{F} \Rightarrow \mathtt{G})$
$(\forall \mathtt{c}:\ \mathtt{F}:\ \mathtt{G}) \triangleq (\forall \mathtt{c}::\ \mathtt{F} \Rightarrow \mathtt{G})$
$(\exists \mathtt{v}:\ \mathtt{F}:\ \mathtt{G}) \triangleq (\exists \mathtt{v}::\ \mathtt{F} \wedge \mathtt{G})$
$(\exists \mathtt{c}:\ \mathtt{F}:\ \mathtt{G}) \triangleq (\exists \mathtt{c}::\ \mathtt{F} \wedge \mathtt{G})$

wobei

`v`	eine <*Variable*> ist,
`c`	eine <*starre Variable*> ist und
`F`,`G`	beide entweder <*allgemeine Formel*>n, <*Aktion*>en oder <*Zustandsfunktion*>en sind.

Eine weitere, wichtige „zusätzliche Notation"

Wir definieren:

$\mathtt{PT}_{\mathtt{f}} \triangleq (\exists \mathtt{hvar}::\ \mathtt{f'=hvar} \wedge \neg\mathtt{PT}.(\mathtt{f}\neq\mathtt{hvar}))$

[29] Anmerkung: Nicht nur die Bedeutungsfunktion führt eine Ersetzung für alle Variablen `"v"` aus (nämlich `(∀"v":.../v)`), auch die „zusätzliche Notation" $\mathtt{f'} \triangleq \mathtt{f}(\forall\mathtt{"v":\ v'/v})$ tut dies. Trotzdem entstehen bei ihr keine Reihenfolgenkonflikte mit der Funktion `pt_eval`, da kein Wert eingesetzt wird, sondern wie bei einem Prädikatentransformator nur Text substituiert wird (und zwar sogar auf eine umkehrbare Weise).

wobei
`PT` ein <*Prädikatentransformator*> ist,
`f` eine <*Zustandsfunktion*> ist und
`hvar` eine <*Variable*> ist.

Damit definieren wir eine weitere Art von Aktionen, ähnlich der Definition der Aktion „$<A>_{\mathtt{f}} \triangleq A \wedge (\mathtt{f}'\neq\mathtt{f})$“.

Mit unserer Definition wollen wir zwei Dinge erreichen. Erstens wollen wir aus einem Prädikatentransformator eine Aktion erzeugen, die den gesamten „Informationsgehalt“ des Prädikatentransformators ausdrückt. Denn wendet man einen Prädikatentransformator auf ein spezielles Prädikat an, so erhält man nur eine Teilinformation über die Operation, die er beschreibt: Falls das spezielle Prädikat nichts über die Variable `z` aussagt, dann wird der Prädikatentransformator auch nicht preisgeben, wie die Operation, die er beschreibt, die Variable `z` verändert. Daher wollen wir mit der neu definierten Aktion beschreiben, wie der Prädikatentransformator auf *alle* Variablen wirkt.

Zweitens wollten wir aber auch eine zentrale Eigenschaft der TLA erhalten, auf deren Wichtigkeit uns Lamport hingewiesen hat ([Lam91a]). Eine Formel einer Logik soll nur Aussagen über Variablen machen, die frei in ihr vorkommen. Beispiel: Wenn die Variable `v` nicht frei in der Formel `t` vorkommt, dann soll `t` äquivalent zu `(∀v:: t)` sein.

Würden wir in unserer obigen Definition nicht die Zustandsfunktion `f` mit anführen, die alle betroffenen Variablen aufzählt, sondern einfach definieren, daß immer alle existierenden Variablen betroffen sind, dann hätten wir ein Problem. Denn es würden damit alle existierenden Variablen frei in der Formel vorkommen, ohne daß sie in der Textdarstellung der Formel erscheinen würden. Damit hätten wir das freie Vorkommen einer Variablen in einer Formel nicht mehr durch ihr textuelles Enthaltensein definieren können, wie dies allgemein üblich und leicht anwendbar ist.

Wir hatten zuerst an eine andere Lösung des Problems gedacht: Wir wollten uns, unter Verwendung der eben erwähnten Alternativdefinition, auf Prädikatentransformatoren beschränken, die nur Aussagen über Variablen machen, die sie explizit erwähnen. (Mit Hilfe des Prädikatentransformators `Same` aus Kapitel 3.5.) Aber wir hätten bei dieser Lösung immer noch den Nachteil gehabt, daß wir in der Definition die Menge aller existierenden Variablen hätten verwenden müssen, die man unter Umständen als unendlich annehmen muß.

Dagegen ist die jetzige Lösung, nicht nur mathematisch vorteilhafter, sondern außerdem auch noch in Hinsicht auf die Eleganz und Kürze der Notation besser.

Zum besseren Verständnis ihrer Arbeitsweise bringen wir ein Beispiel:

$$
\begin{aligned}
& (\mathtt{x:=5})_{(\mathtt{x,y})} \\
\equiv\ & (\exists \mathtt{hvar}::\ (\mathtt{x,y})'=\mathtt{hvar} \wedge \neg(\mathtt{x:=5}).((\mathtt{x,y})\neq\mathtt{hvar})) \\
\equiv\ & (\exists (\mathtt{h1,h2})::\ (\mathtt{x}',\mathtt{y}')=(\mathtt{h1,h2}) \wedge \neg((5,\mathtt{y})\neq(\mathtt{h1,h2}))) \\
\equiv\ & (\exists \mathtt{h1}::\ (\exists \mathtt{h2}::\ \mathtt{x}'=\mathtt{h1} \wedge \mathtt{y}'=\mathtt{h2} \wedge 5=\mathtt{h1} \wedge \mathtt{y}=\mathtt{h2})) \\
\equiv\ & \mathtt{x}'=5 \wedge \mathtt{y}'=\mathtt{y}
\end{aligned}
$$

Wir haben also eine Aktion erhalten, die folgendes aussagt:

- Egal welchen Wert `x` vorher hat, ist der Wert hinterher gleich `5`.
- `y` verändert sich nicht.
- Alle Variablen außer `x` und `y` dürfen sich beliebig verhalten.

Interessant ist vielleicht noch, was mit Aussagen des Prädikatentransformators über Variablen geschieht, die in der Zustandsfunktion nicht erwähnt werden. Sie werden

in der Aktion nicht berücksichtigt, wie das folgende Beispiel zeigt:

$$
\begin{aligned}
& (\mathtt{z:=5})_{(\mathtt{x,y})} \\
\equiv\ & (\exists \mathtt{hvar}::\ (\mathtt{x,y})'=\mathtt{hvar} \wedge \neg(\mathtt{z:=5}).((\mathtt{x,y})\neq\mathtt{hvar})) \\
\equiv\ & \mathtt{x}'=\mathtt{x} \wedge \mathtt{y}'=\mathtt{y}
\end{aligned}
$$

Indem wir die betroffenen Variablen immer mit angeben, wird der Text unserer Formeln etwas länger. Dieser Effekt trat bereits in der TLA auf, wo in einer Aktion $[A]_{\mathtt{f}}$ zur eigentlichen Aktion A immer eine Zustandsfunktion `f` mit allen Variablen angegeben werden mußte, die unverändert blieben, wenn die Aktion A nicht stattfand. In Kapitel 10.1 von [Lam91] führt Lamport aus, daß dies einen etwa 10% längeren Text bedeutet. Aber nur so bleibt die Logik einfach, so daß solche einfachen, wichtigen Schlußregeln der gewöhnlichen Mathematik wie die oben erwähnte mit „$(\forall \mathtt{x}::\ \mathtt{t})$“ gelten können. Und diese Vereinfachung bei der Beweisarbeit ist das geringe Mehr an Schreibarbeit wert.

Vielleicht noch ein Wort zu unserem grundsätzlichen Ansatz. Der Grund, warum wir überhaupt Ausdrücke der Form $\mathtt{PT}_{\mathtt{f}}$ in das Kalkül eingeführt haben, liegt darin, daß wir auf diese Weise vermeiden, eine Quantifizierung über eine Variable `X` vom Typ Prädikat explizit in das Kalkül aufzunehmen. Denn nur durch eine Quantifizierung kann man nicht nur eine Teilinformation, sondern die gesamte Information aus dem Prädikatentransformator „herausziehen“. Stattdessen geben wir mit unserer Definition ein bestimmtes Prädikat vor und quantifizieren nur über gewöhnliche Variablen. Eine Quantifikation über Variablen vom Typ Prädikat hätte erhebliche Schwierigkeiten verursacht, da wir dann zwei völlig verschiedene Arten von Variablenbegriff nebeneinander gehabt hätten.

Schließlich noch eine Bemerkung zur Verwendung der Definition des Ausdrucks $\mathtt{PT}_{\mathtt{f}}$. Man wird sie nur selten explizit gebrauchen, da man sie im wesentlichen benötigt, um die Konsistenz und Vollständigkeit unseres Kalküls nachzuweisen. Wenn man das Kalkül anwendet und mit den Schlußregeln Schlußfolgerungen zieht, wird man neue abgeleitete Schlußregeln verwenden, die mit den Prädikatentransformatoren direkt und elegant umgehen. (Siehe Kapitel 5.3)

Als nächstes werden wir begründen, warum wir es mit der auf den ersten Blick merkwürdig erscheinenden inneren Struktur unserer Definition von $\mathtt{PT}_{\mathtt{f}}$ erreichen, „den gesamten Informationsgehalt aus `PT` herauszuziehen“.

Dazu betrachten wir Abbildung 3-4 aus Kapitel 3.4. Wir nehmen uns einen der Endzustände (auf der rechten Seite) heraus und wählen ein Prädikat $\mathtt{X}'$, das ihn eindeutig beschreibt.

Entsprechend der Definition von `wlp` beschreibt `wlp.S.X` die Klasse der Anfangszustände, von denen keine Berechnung nach $\neg\mathtt{X}'$ führt. `wlp.S.`$\neg$`X` beschreibt alle Anfangszustände, von denen keine einzige Berechnung nach $\mathtt{X}'$ führt. Und $\neg$`wlp.S.`$\neg$`X` beschreibt alle Anfangszustände, von denen eine Berechnung nach $\mathtt{X}'$ führen kann.[30]

[30] $\neg$`wlp.S.`$\neg$`X` ist nicht äquivalent zu `wp.S.X`: `wp.S.X` beschreibt alle Anfangszustände, in denen keine anderen als diejenigen Berechnungen starten, die nach $\mathtt{X}'$ führen. („Alle ..., die können“ versus „Alle ..., die nichts anderes tun“.) Gegenbeispiel für Abbildung 3-4: Wähle für $\mathtt{X}'$ den zweiten Zustand von oben. $\neg$`wlp.S.`$\neg$`X` beschreibt den Zustand links davon, `wp.S.X` beschreibt keinen Zustand, ist also `False`. Nur für deterministische Operationen wäre die Äquivalenz gegeben. Aber wir werden auch nichtdeterministische Operationen beschreiben müssen, denn bereits Pascal enthält etliche davon. Beispiele: Der Aufruf einer Prozedur mit nichtinitialisierten lokalen Variablen, der Wert des Laufindexes nach dem Ende einer `For`-Schleife, die Reihenfolge der Auswertung von Prozedurparametern und von arithmetischen Ausdrücken, ...

Damit beschreiben wir mit jedem Term $X' \wedge \neg wlp.S.\neg X$ für den zugehörigen Endzustand, von welchen Anfangszuständen aus er erreicht werden kann. Mit einer Disjunktion über alle solche Terme für alle X', die jeweils genau einen Endzustand beschreiben, haben wir somit alle Relationspfeile der Zeichnung vollständig festgelegt, abgesehen von den nicht endenden Pfeilen. Und wenn man jetzt statt einer Disjunktion über alle entsprechenden Prädikate X', wie oben bereits erläutert, eine Quantifikation über gewöhnliche Variablen einführt, dann ist man bei unserer Definition angelangt.

Die auf den ersten Blick merkwürdig erscheinende zweifache Negation in der Definition wird nun auch verständlich. Für deterministische Operationen ist sie überflüssig, aber nichtdeterministische Operationen werden nur so richtig beschrieben. Beispiel: Eine Operation, die die Variable x auf einen beliebigen Wert setzt, beschrieben durch den Prädikatentransformator $Change.\{x\}$ aus Kapitel 3.5.

$$
\begin{array}{ll}
 & Change.\{x\}_{(x,y)} \\
\equiv & \{ \text{ Definition PT_f } \} \\
 & (\exists hvar::\ (x,y)'=hvar \wedge \neg Change.\{x\}.((x,y)\neq hvar)) \\
\equiv & \{ \text{ Definition für } Change.\{x\} \} \\
 & (\exists (h1,h2)::\ (x',y')=(h1,h2) \wedge \neg(\forall x::\ (x,y)\neq(h1,h2))) \\
\equiv & \{ \text{ Auflösen der Paare } \} \\
 & (\exists h1::\ (\exists h2::\ x'=h1 \wedge y'=h2 \wedge \neg(\forall x::\ x\neq h1 \vee y\neq h2))) \\
\equiv & \{ \text{ Eigenschaft von "}\forall x\text{" } \} \\
 & (\exists h1::\ (\exists h2::\ x'=h1 \wedge y'=h2 \wedge \neg(False \vee y\neq h2))) \\
\equiv & \{ \text{ Vereinfachen } \} \\
 & (\exists h1::\ x'=h1 \wedge y'=y) \\
\equiv & \{ \text{ Vereinfachen } \} \\
 & y'=y
\end{array}
$$

Nur durch die zwei Negationen erhalten wir das erwartete Ergebnis.

Die eben erwähnten nicht endenden Berechnungen der Operation haben kein Gegenstück in unserer Definition, wir lassen sie einfach „unter den Tisch fallen“. Mit dem Konzept der Aktion sind sie ohnehin nicht vereinbar, denn eine Aktion hat immer einen Endzustand.

Wenn eine Operation von einem Anfangszustand aus nur nicht endende Berechnungen beginnt, dann kann die zugehörige Aktion überhaupt nicht schalten. Hat die Operation dort endende und nicht endende Berechnungen, dann verhält sich die Aktion so, als gäbe es nur die endenden Berechnungen.

Das Problem der nicht endenden Berechnungen bedarf also noch einiger Überlegungen. In Kapitel 4.5 unter Punkt (e) haben wir das Problem bereits angerissen, und wie dort versprochen geben wir hier nun die Lösung an.

Unsere Definition beschreibt nur die Semantik von Systemen von Operationen, in denen jede Ausführung einer Operation terminiert. Dies ist aber genau das, was wir zur Beschreibung der Semantik von Estelle-Spezifikationen benötigen, denn die Bedeutung einer Estelle-Spezifikation, in der eine Estelle-Transition zu schalten beginnt, aber ihr Rumpf nicht terminiert, ist nicht definiert. (Anderenfalls wäre das Grundkonzept der Atomizität von Estelle-Transitionen zerstört.) Daher ist es gar nicht notwendig, daß wir diesen Fall bei der Semantikdefinition abdecken.

Demgegenüber muß jeder, der formale Eigenschaften einer Estelle-Spezifikation nachweisen will, vorher überhaupt sicherstellen, daß seine Estelle-Spezifikation wohldefiniert ist, es also unmöglich ist, daß eine schaltende Estelle-Transition „nicht terminiert“. Anderenfalls existierten überhaupt keinerlei Eigenschaften der „Estel-

le-Spezifikation“.

Dieser Nachweis kann auch innerhalb der TLA/PT geschehen, indem für jede Estelle-Transition `tr` mit Transitionsrumpf `b`

$$\Box[(\texttt{tr.Selected} \wedge \neg\texttt{tr.Selected}') \Rightarrow \texttt{wp.b.True}]_{\texttt{f}}$$

gezeigt wird. (Wie das Schalten von Estelle-Transitionen genau dargestellt wird, sehen wir später. Der linke Term bedeutet jedenfalls, daß die Estelle-Transition schaltet[31], und der rechte, daß zu Beginn der Aktion das Prädikat `wp.b.True` erfüllt ist, wodurch die Termination sichergestellt wird.)

Wir fordern dabei nicht, daß jeder Transitionsrumpf unter allen Umständen terminieren muß, sondern nur, daß er terminiert, wenn er „zur Ausführung“ kommt.

Es stellt sich natürlich die Frage, wie wir über eine Estelle-Spezifikation schlußfolgern können, die gar nicht wohldefiniert ist, also gar keine Bedeutung hat. Aber wie wir gesehen haben, ist unsere Bedeutungsfunktion auch für Zeichenketten definiert, die deswegen keine Estelle-Spezifikation darstellen, weil die Termination einiger „Transitionsrümpfe“ nicht sichergestellt ist. Unsere Bedeutungsfunktion übersetzt diese Zeichenketten in eine TLA/PT-Spezifikation, die genau unserer intuitiven Vorstellung von der Erweiterung der Bedeutung entspricht, wobei nicht endende Operationen einfach ersatzlos herausfallen.

Und mit Hilfe der obigen Bedingung können wir dann untersuchen, ob es sich um eine „erweiterte“ Spezifikation handelt, oder ob sie die Bedingung erfüllt und eine Estelle-Spezifikation im herkömmlichen Sinne ist.

Unsere Erweiterung der Bedeutung wird dadurch gerechtfertigt, daß bei einer undefinierten Estelle-Transition *alles* geschehen kann, und wir deshalb die freie Auswahl haben, wie wir die erweiterte Bedeutung festlegen.

Die Untersuchung der Wohldefiniertheit einer Estelle-Spezifikation ist mit einem gewissen Aufwand verbunden: Um die sonstigen Eigenschaften abzuleiten, müssen wir den Prädikatentransformator `wlp.b` für jeden Transitionsrumpf `b` ableiten. Nur für diese Untersuchung benötigen wir dagegen den Prädikatentransformator `wp.b`.

Aber wir können auch andersherum argumentieren: Um eine Operation `b` vollständig zu beschreiben, benötigen wir sowohl `wlp.b` als auch `wp.b`. Für alle anderen Untersuchungen als die der Wohldefiniertheit sparen wir schlicht die Arbeit mit dem zweiten Prädikatentransformator ein, weil wir unser System geeignet konstruiert haben.

[31] Hierzu muß die Schaltbedingung erfüllt sein, die sich einerseits aus den explizit in Estelle spezifizierten Bedingungen zusammensetzt und andererseits aus vielen weiteren Bedingungen, die vom Estelle-Ausführungsmodell diktiert werden (siehe Kapitel 7). Daher die obige Aktion und nicht einfach die (explizit spezifizierten) Schaltbedingungen.

5.2 Strukturierte Variablen in der TLA/PT

Bisher haben wir lediglich unstrukturierte Variablen betrachtet, genau wie es die TLA-Beispiele in [Lam91] tun. Für unsere Arbeit werden wir aber auch Verbunde (englisch: „records“) benötigen. In der TLA$^+$ ([Lam91b], siehe auch Kapitel 3.3 und 5.4) werden auch sie eingeführt. Dijkstra ([DiSc90]) dagegen betrachtet Verbunde nicht. Und in der Tat werfen Verbunde im Zusammenhang mit den Prädikatentransformatoren einige Fragen auf.

Diese Fragen treten im wesentlichen bei der Zuweisungsoperation auf. Für die Zuweisungsoperation ≫`"v:=42"`≪ ist die schwächste freie Vorbedingung:

```
wlp."v:=42".X ≜ (v:=42).X
```

Dies ist die übliche, simple Textersetzung, die den Text ≫`v`≪ durch den Text ≫`42`≪ ersetzt.

Nehmen wir nun an, daß die Variable `m` ein Verbund mit den drei Komponenten `m.v1`, `m.v2` und `m.v3` sei. Hier kann man versuchen, den Prädikatentransformator der Zuweisungsoperation ganz analog zu definieren:

```
wlp."m.v1:=42".X ≜ (m.v1:=42).X
```

Solange im Prädikat `X` nur die drei Komponenten `m.v1`, `m.v2` und `m.v3` vorkommen, ergibt dies auch die gewünschte Semantik. Aber sobald die Variable `m` ohne einen der Selektoren vorkommt, erhalten wir etwas nicht Gewünschtes. (Hier dargestellt in der [Lam91b] entnommenen Notation für Werte von Verbunden):

```
  wlp."m.v1:=42".(m = [[v1≜42, v2≜0, v3≜0]])
= (m.v1:=42).(m = [[v1≜42, v2≜0, v3≜0]])
= (m = [[v1≜42, v2≜0, v3≜0]])                ?????
```

Gewünscht hätten wir:

```
= (m = [[v2≜0, v3≜0]])
```

Denn die schwächste freie Vorbedingung sollte sein, daß die Komponente `v1` von `m` einen beliebigen Wert haben kann.

Die Standardlösung (und allgemeine Lösung) für dieses Problem ist, nicht den Text ≫`m.v1`≪ zu ersetzen, sondern den Text ≫`m`≪. Dabei wird allerdings der dafür eingesetzte Text komplizierter:

```
wlp."m.v1:=42".X ≜ (m:=[[v1≜42, v2≜m.v2, v3≜m.v3]]).X
```

Hiermit ergibt sich wie gewünscht:

```
  wlp."m.v1:=42".(m = [[v1≜42, v2≜0, v3≜0]])
= ([[v1≜42, v2≜m.v2, v3≜m.v3]] = [[v1≜42, v2≜0, v3≜0]])
= (m = [[v2≜0, v3≜0]])
```

Gerade bei Verbunden mit vielen Komponenten, wie wir sie verwenden werden, wird dies schnell unhandlich. Daher wollen wir diese Lösung vermeiden.

Bevor wir unsere Lösung angeben, betrachten wir zunächst einmal, wie Lamport in der TLA$^+$ die Semantik von Verbunden definiert.

Verbunde werden dort als eine spezielle Form von Feldern (englisch: „arrays“) definiert. Ein Feld ist in der TLA$^+$ ein grundlegender strukturierter Datentyp, und der zweistellige „Mixfix“-Operator ≫`_[_]`≪ ordnet einem Paar aus einem Feld und einem Indexwert eine Komponente des Feldes zu. Indexwertebereiche können natürliche Zahlen, aber zum Beispiel auch Zeichenketten sein. Beispiel: ≫`nachname["Bob"]`≪

Verbunde sind nun als Felder definiert, deren Indexmenge Zeichenketten sind. Beispiel: ≫`r.nam`≪ ist definiert als ≫`r["nam"]`≪. Hieraus wird ersichtlich, daß nur Zeichenketten-Konstanten als Selektoren von Verbunden syntaktisch zulässig sind, da in der Verbund-Notation die Anführungsstriche weggelassen wurden.

Dies erlaubt übrigens auch eine etwas veränderte Sichtweise: Anstelle des einen zweistelligen Operators ≫`_._`≪ können wir auch etliche einstellige Postfix-Operatoren ≫`_.nam`≪, ≫`_.year`≪, ≫`_.day`≪, ... betrachten.

Ein einzelner Verbund wird, wie oben bereits angedeutet, in doppelten eckigen Klammern notiert, wobei für die Komponente x_i jeweils der Wert e_i angegeben werden kann:

$[[x_1 \triangleq e_1, \ldots, x_n \triangleq e_n]]$

Und es gilt:
$[[\ldots, x_i \triangleq e_i, \ldots]].x_i = e_i$

Kommen wir nun zu unserer Lösung:

In der TLA/PT definieren wir die Verbunde genau wie in der TLA$^+$.

Aber wir schränken die Prädikate syntaktisch ein, die als Argumente von Prädikatentransformatoren verwendet werden können. Wenn sie Verbundvariablen enthalten, dann muß auf diese jeweils eine Selektorfunktion angewendet werden, oder es muß über diese Verbundvariable quantifiziert werden, so daß sie nicht frei ist (und sie daher von einer Textersetzung nicht erfaßt wird).

Diese Einschränkung können wir folgendermaßen erreichen: In der EBNF-Syntaxdefinition der TLA/PT in Kapitel 5.1 sind die Produktionsregeln für das Nichtterminal <*Variable*> (und für <*starre Variable*>) noch offengelassen. Für uneingeschränkte Variablen ist die Definition ohnehin offensichtlich. Für eingeschränkte Variablen ist sie ebenso leicht, aber wir verwenden für diese Variablen ein neues Nichtterminal (und für starre Variablen ein zweites), welches wir statt des alten in denjenigen Produktionsregeln verwenden, die eingeschränkte Variablen enthalten sollen.

Die syntaktische Verwendung von Verbund-Variablen ist in der TLA/PT damit zwar eingeschränkt, aber die Ausdrucksfähigkeit bleibt gleich. Denn anstelle einer Verbundvariablen ohne Selektorfunktion können wir immer einen Verbund verwenden, der als Komponenten die selektierten Komponenten der Verbundvariablen besitzt. Beispiel: Habe die Verbundvariable `m` die Komponenten `v1`, `v2` und `v3`. Dann gilt:

```
m = [[m.v1, m.v2, m.v3]]
```

Und diesen Verbund können wir ohne Einschränkungen verwenden. Für unsere Zwecke werden wir dies allerdings kaum brauchen, und unser Ziel ist ja gerade, diesen komplexen Ausdruck möglichst zu vermeiden.

Die Estelle-Zuweisungsoperation ≫`"v:=e"`≪ in einer Modulinstanz `m` können wir hinfort wieder durch die simple Textersetzung ≫`(m.v:=e).X`≪ definieren, unsere Formeln werden bei Verifikationen nicht durch aufwendige Ersetzungstexte aufgeblasen.

Zur Sicherheit sollten wir noch einen Blick auf die Definition von PT_f aus Kapitel 5.1 werfen, mit der wir „alleinstehende“ Prädikatentransformatoren als Aktionen definiert hatten. Die Definition lautete:

$PT_f \triangleq (\exists hvar :: f' = hvar \wedge \neg PT.(f \neq hvar))$

wobei
PT

ein <*Prädikatentransformator*> ist,
`f` eine <*Zustandsfunktion*> ist und
`hvar` eine <*Variable*> ist.

Wie man sieht, kommt in dem Prädikat, auf das `PT` angewendet wird, die Variable `hvar` vor, und außerdem sind auch alle Variablen, die in der Zustandsfunktion `f` enthalten sind, Teil des Prädikates. Für alle diese Variablen gilt also, daß sie keine Verbundvariablen ohne Selektor enthalten dürfen.

Diese Einschränkung ist recht ärgerlich, denn wenn man zum Beispiel eine Modulinstanz durch einen Verbund `m` beschreiben will, dann soll sich eine Prädikatentransformator-Aktion eben auch genau auf diesen Verbund beziehen, ohne daß man die oben beschriebene Verbund-Ersetzung explizit ausführen muß:

$\mathtt{PT}_{[[\mathtt{m.v1},\ldots,\mathtt{m.vn}]]}$

Daher ergänzen wir die obige Definition von $\mathtt{PT}_{\mathtt{f}}$:

$$\mathtt{PT}_{\mathtt{f}} \triangleq (\exists \mathtt{hvar}::\ \underline{\mathtt{f}}'=\mathtt{hvar} \wedge \neg\mathtt{PT}.(\underline{\mathtt{f}}\neq\mathtt{hvar}))$$

wobei $\underline{\mathtt{f}}$ die Zustandsfunktion ist, die man erhält, wenn man (wie oben beschrieben) in `f` alle freien Verbundvariablen ohne Selektorfunktion durch einen Verbund ersetzt, der als Komponenten alle selektierten Komponenten der Verbundvariablen enthält.

Dazu der Anschaulichkeit halber noch ein Beispiel: Wenn `m` die drei Komponenten `v1`, `v2` und `v3` besitzt, ist $\mathtt{PT}_{\mathtt{m}}$ definiert als:

$$(\exists \mathtt{hvar}::\ [[\mathtt{m.v1},\mathtt{m.v2},\mathtt{m.v3}]]'=\mathtt{hvar} \wedge \neg\mathtt{PT}.([[\mathtt{m.v1},\mathtt{m.v2},\mathtt{m.v3}]]\neq\mathtt{hvar}))$$

Un da wir auf Ausdrücke, die Prädikatentransformatoren enthalten, eigene Schlußregeln anwenden wollen, müssen wir während der Verifikationsarbeit diese Auflösung nicht verwenden, wir können bei der kurzen, handlichen Form bleiben.

5.3 Schlußregeln der TLA/PT

Die Schlußregeln der TLA/PT benötigen wir nicht, um die Semantik von Estelle definieren zu können. Daher werden wir in diesem Kapitel vieles nur andeuten und insbesondere die Untersuchung der Konsistenz und Vollständigkeit nicht ausarbeiten. Der Entwurf einer in jeder Hinsicht vollständigen und „runden“ TLA/PT wäre auf jeden Fall mindestens eine eigene Diplomarbeit wert, weshalb wir diese Arbeit hier nicht leisten können. (Im übrigen wird auch in der Definition der TLA in [Lam91] nur auf die Zukunft verwiesen, als die Frage der Korrektheit zur Sprache kommt.) Wir werden bei der Beschreibung der Schlußregeln nur soweit gehen, daß klar wird, wie mit der TLA/PT über Estelle geschlußfolgert werden kann, das Erreichen von Vollständigkeit (und damit Korrektheit) ist nicht Gegenstand dieser Arbeit.

Für die Schlußregeln der TLA/PT übernehmen wir als erstes sämtliche Schlußregeln der TLA.

Dabei ist allerdings zu beachten, daß wir für die Regeln `E1`, `E2`, `F1` und `F2` den Begriff „`x` kommt in `G` frei vor“ auf solche `G` erweitern müssen, die Prädikatentransformatoren enthalten.

Dank der im letzten Kapitel bereits diskutierten geeigneten Konstruktion unserer Definition macht dies aber keine Schwierigkeiten. Sowohl für einen Ausdruck der Form ≫`PT.X`≪ als auch für einen Ausdruck der Form ≫$\mathtt{PT}_{\mathtt{f}}$≪ definieren wir, daß eine Variable darin frei vorkommt, wenn sie darin textuell enthalten ist. Dies ist auch die Definition für die bisherigen TLA-Ausdrücke, wir erweitern diese Definition damit ohne Änderung auf die neue Art von Ausdrücken.

Die grundlegenden und die abgeleiteten Schlußregeln der TLA sind für Beweise über Prädikatentransformatoren in der Form ≫$\mathtt{PT}_{\mathtt{f}}$≪ nicht sehr handlich, da die Prädikatentransformatoren dafür erst in Aktionen übersetzt werden müßten. Benötigt werden daher weitere spezielle abgeleitete Schlußregeln, die wie das Beispiel in Kapitel 4.5 möglichst sogar einfacher sind als die Schlußregeln für Aktionen.

Wir geben hier die zentrale Schlußregel an, die es erlaubt, Schlußfolgerungen aus Formeln zu ziehen, die die Form von Estelle-Spezifikationen haben. Diese Schlußregel `INV3` ist das Gegenstück zur TLA-Schlußregel zum Beweisen von Invarianten, `INV1` (siehe Kapitel 3.3). Mit `INV3` wollen wir Invarianten für Spezifikationen ableiten, die die spezielle Form von in die TLA/PT umgesetzten Estelle-Spezifikationen haben.

Die alte Schlußregel `INV1` lautete:

$$\frac{(\mathtt{I} \wedge [N]_{\mathtt{f}}) \Rightarrow \mathtt{I}'}{(\mathtt{I} \wedge \Box[N]_{\mathtt{f}}) \Rightarrow \Box\mathtt{I}}$$

wobei
`f` eine Zustandsfunktion ist,
`I` ein Prädikat ist und
N eine Aktion ist.

Unsere neue Schlußregel `INV3` soll eine ähnliche Konsequenz besitzen, aber wir werden der Formel $\Box[N]_{\mathtt{f}}$ gewisse strukturelle Bedingungen auferlegen. Damit können wir die Prämisse derart gestalten, daß in ihr keine „alleinstehenden“ Prädikatentransformatoren der Form ≫$\mathtt{PT}_{\mathtt{f}}$≪ mehr vorkommen. Diese Eigenschaft erleichtert das Beweisen der Prämisse.

Die Schlußregel INV3 findet sich in Abbildung 5-2. Die spezielle Form von φ darin wird in Kapitel 7 klar werden, wenn wir das Ausführungsmodell für Estelle definieren werden. Aber soviel sei jetzt schon gesagt: Die Aktion

$(\texttt{tr1.Selected} \wedge \neg\texttt{tr1.Selected}')$

drückt das Schalten einer Estelle-Transition aus, und der Prädikatentransformator PT1 beschreibt, welche Wirkung dies hat. Die Aktion N faßt alle sonstigen reinen TLA-Aktionen des Systems zusammen, zum Beispiel die Beschreibung der Auswahl von Estelle-Transitionen zum Schalten.

INV3.

$$
\begin{array}{l}
(\texttt{I} \wedge [N]_{\texttt{f}}) \Rightarrow \texttt{I}' \\
(\texttt{I} \wedge \texttt{tr1.Selected}) \Rightarrow \texttt{PT1.((Change.\{tr1.Selected).I)} \\
\ldots \\
(\texttt{I} \wedge \texttt{trn.Selected}) \Rightarrow \texttt{PTn.((Change.\{trn.Selected).I)} \\
\hline
(\texttt{I} \wedge \varphi) \Rightarrow \Box\texttt{I}
\end{array}
$$

wobei

PT1,...,PTn	Prädikatentransformatoren sind,
N	eine Aktion ist,
I	ein Prädikat ist,
tr1,...trn	Verbundvariablen (unter anderem) mit den Komponenten Id und Selected sind,
f	eine Zustandsfunktion ist,

$\texttt{Trans} \triangleq \{\texttt{tr1.Id},\ldots,\texttt{trn.Id}\}$ und

$$
\begin{array}{l}
\varphi \triangleq \\
\quad \Box[N]_{\texttt{f}} \\
\quad \wedge\ \Box[\ (\texttt{tr1.Selected} \wedge \neg\texttt{tr1.Selected}') \\
\qquad \Rightarrow (\texttt{PT1.(Change.\{tr1.Selected\})})_{\texttt{f}}]_{\texttt{tr.Selected}} \\
\quad \wedge\ \ldots \\
\qquad \Box[\ (\texttt{trn.Selected} \wedge \neg\texttt{trn.Selected}') \\
\qquad \Rightarrow (\texttt{PTn.(Change.\{trn.Selected\})})_{\texttt{f}}]_{\texttt{tr.Selected}} \\
\quad \wedge\ \Box[(\exists\texttt{tr: tr.Id} \in \texttt{Trans:} \\
\qquad\quad \texttt{tr.Selected} \neq \texttt{tr.Selected}')]_{\texttt{f}}
\end{array}
$$

Abbildung 5-2: Die abgeleitete Schlußregel INV3.

In Abbildung 5-2 beachte man in der Definition von »φ« die letzten beiden Zeilen. Sie besagen, daß entweder eine der Estelle-Transitionen aus »Trans« ausgewählt wird beziehungsweise schaltet, oder daß (mit den in f erwähnten Variablen) nichts geschieht. Mit anderen Worten: »Trans« muß die *vollständige* Menge der Estelle-Transitionen enthalten. Um die Schlußregel INV3 anwenden zu können, müssen wir *alle* Estelle-Transitionen betrachten. Dies war aber auch zu erwarten, wenn wir Invarianten der gesamten Spezifikation beweisen wollen.

Als Trost für diese scheinbar große Arbeit bleibt uns, daß oft die Prädikatentransformatoren vieler der Estelle-Transitionen die Variablen überhaupt nicht berühren, die in der Invarianten I enthalten sind, so daß der Nachweis vieler der Prämissen trivial ist.

5.4 Weiteres aus der TLA$^+$ für die TLA/PT

Lamport führt in [Lam91] die grundlegende TLA ein. Um mit diesem Formalismus noch besser spezifizieren zu können, war er zur Zeit der Erstellung unserer Arbeit dabei, eine Erweiterung „TLA$^+$“ der TLA zu entwickeln. [Lam91b] gibt den Stand der Entwicklung wieder. Diese Erweiterung verändert nichts an der Semantik der TLA, sie fügt lediglich weitere syntaktische Konstrukte hinzu, die mit einfachen Algorithmen auch wieder entfernt werden könnten. Die große semantische Einfachheit der TLA bleibt trotz aller Erweiterungen erhalten. Weiterhin werden leistungsfähige Datenstrukturen eingeführt, ein Beispiel dafür haben wir mit den Verbunden bereits im Kapitel 5.2 angeführt.

Die syntaktische Unterstützung der TLA hält Lamport für notwendig, um auch praktisch mit größeren Formeln umgehen zu können. Das wichtigste syntaktische Konstrukt, das neu eingeführt wird, ist die *Definition*. Die Notwendigkeit der Definition demonstriert er durch die Überlegung, ob man zum Beispiel Arithmetik praktisch betreiben könne, wenn man die Zahl `27` und den Operator `+` nur mittels der Primitive `0` und der Nachfolgerfunktion ausdrücken soll.

Außer der Definition besitzt die TLA$^+$ weiterhin das Konzept des *Moduls*, um große Spezifikationen strukturieren zu können. Aber sowohl die Definition wie auch das Modul sind wie gesagt rein syntaktische Konzepte, die algorithmisch wieder entfernt werden könnten, um eine große (und unleserliche) TLA-Formel zu erhalten.

Für unsere Arbeit, beziehungsweise für ihre Übersichtlichkeit, werden wir das Konzept der Definition ebenfalls benötigen. Aber leider ist [Lam91b] noch nicht sehr ausgearbeitet, so daß es uns nicht angebracht erscheint, schon jetzt die gesamte TLA$^+$ zu übernehmen. Die meisten TLA$^+$-Konstrukte werden in Lamports Arbeit lediglich eingeführt, indem sie an einem Beispiel ausführlich demonstriert werden.

Sobald die TLA$^+$ fertiggestellt ist, würde es dagegen sehr sinnvoll sein, die TLA/PT zu einer TLA$^+$/PT zu erweitern. Denn auch im Rahmen unserer Arbeit zeigt es sich bereits, daß syntaktische Hilfsmittel sehr willkommen wären, um die Formeln zu strukturieren und übersichtlicher zu machen. Da wir aber die TLA/PT ohnehin nicht vollständig ausarbeiten (zum Beispiel bei den Schlußregeln), können wir auch an diesen Stellen ohne weitere Nachteile einige Punkte für zukünftige Ergänzungen offen lassen.

Das Konzept der Definition werden wir daher in der weiteren Arbeit benutzen, aber wir werden es nicht in einem formalen Rahmen tun.

Die vollständige Syntax und Semantik und die von uns definierten Schlußregeln der TLA/PT sind noch einmal im Anhang zusammengetragen.

6. Beschreibung der Semantik von Estelle-Texten in TLA/PT

6.1 Konzept der Semantikbeschreibung

Der ISO-Standard für Estelle beschreibt bereits per Definition genau die Semantik von Estelle. Wie wir in Kapitel 1 begründet haben, ist der ISO-Standard allerdings an vielen Stellen nicht so formal, daß er als Grundlage für eine Eigenschaftsanalyse oder eine Verifikation mit Hilfe eines vollständig formalen Kalküls dienen könnte.

In dieser Arbeit zeigen wir nun einen Weg auf, wie ein dafür ausreichender Grad an Formalisierung erreicht werden kann, und wir definieren entsprechend formal das sogenannte *Ausführungsmodell* von Estelle. Dies bedeutet natürlich, daß Details von Estelle aus dem ISO-Standard in dieser Arbeit häufig exakter, fast immer aber auch auf eine andere Weise definiert werden. Das hat zur Konsequenz, daß hier notwendigerweise so einiges leicht anders definiert wird.

Einerseits müssen umgangssprachliche Aussagen für eine Fassung in TLA/PT konkretisiert werden, was jeweils Entscheidungen für eine von mehreren möglichen Lösungen bedeutet. Und andererseits hat die ganz andere Art der Darstellung, die für die oben und in Kapitel 1 genannten Ziele besser geeignet sein soll, unvermeidlicherweise die Folge, daß sich ab und zu kleine Differenzen bei der Definition ergeben. (Der theoretische Ausweg wäre eine Verifikation der beiden Definitionen gegeneinander. Aber abgesehen vom großen Arbeitsaufwand wäre der hierfür oft zu geringe Formalisierungsgrad des ISO-Standards ein Hinderungsgrund.)

Wir werden selbstverständlich versuchen, mit der TLA/PT-Semantik so nahe wie möglich am ISO-Standard, beziehungsweise den Absichten hinter ihm, zu bleiben. Trotzdem können unsere Definitionen immer nur *unsere* Auffassung von Estelle festlegen, nicht das ISO-Estelle.

Sobald unser Ansatz zur Definition vollständig umgesetzt ist, könnte man allerdings prüfen, ob es sinnvoll wäre, der neuen Version das Etikett „offizielle Definition“ zu geben. Nur ist es nicht ratsam, mehrere verschiedene Definitionen für dasselbe gleichzeitig zu verwenden, solange ihre Äquivalenz nicht nachgewiesen ist.

Kommen wir nun zu unserem Ansatz, die Semantik einer Estelle-Spezifikation zu beschreiben. Es handelt sich um folgendes Grundproblem: Wir haben einen Text als Zeichenkette vorliegen und wollen ihm eine Bedeutung zuordnen.

Es gibt verschiedene Arten, die Bedeutung zu definieren, in Kapitel 3.1 haben wir bereits einen Überblick gegeben. In Kapitel 4.1 haben wir uns dann im Grundsatz für eine der Arten entschieden, und zwar für die operationale Beschreibungsform und damit für einen abstrakten Automaten.

Ursprünglich hatten wir für unsere Arbeit beabsichtigt, diesen abstrakten Automaten durch Tupel und Mengen, wie in Kapitel 3.2 definiert, explizit anzugeben. Der Automat sollte als Tripel aus der Menge der Anfangszustände, der Menge der Zustände und der Zustandsübergangsrelation aufgeschrieben werden. Wir hatten versucht, einen Zustand dann ebenso als Tupel aus Mengen und weiteren Tupeln hierarchisch strukturiert zu notieren, analog zur Struktur einer Estelle-Spezifikation. Die Zustandsübergangsrelation wäre dann eine Relation über einer Menge dieser Struktur gewesen.

Es zeigte sich aber, daß dies kein guter Weg war. Denn die Tupel wurden sehr umfangreich und dadurch so unübersichtlich, daß ihr Aussagewert verlorenging. Ent-

sprechend war die Zustandsübergangsrelation auf einer derart komplexen Struktur definiert, daß ihre korrekte Definition größte Schwierigkeiten machte. Und es wäre völlig unmöglich gewesen, mit ihrer Hilfe noch irgendwie praktisch Eigenschaften des Systems zu untersuchen.

Daher entschlossen wir uns, den Automaten nicht explizit anzugeben.

Stattdessen spezifizieren wir seine Eigenschaften auf der Basis der Prädikatenlogik. Dabei vermeiden wir die explizite Definition eines Gesamtzustandes dadurch, daß wir dessen Komponenten durch einzelne Variablen beschreiben. Eine Zusammenfassung zu einem Gesamtzustand einer Modulinstanz oder gar einer ganzen Spezifikation findet nicht mehr statt. So wird es möglich, sich nur auf die jeweils relevanten Teile zu beziehen, sowohl bei der Systembeschreibung, wie auch bei der Argumentation über einzelne Eigenschaften des Systems.

Diesen Entschluß, den abstrakten Automaten nur implizit anzugeben, halten wir für entscheidend, ohne ihn wäre die Definition von Estelle so unübersichtlich, daß sie nutzlos wäre.

Es sei vielleicht noch angemerkt, daß der ISO-Standard für Estelle in seinem „formalen“ Teil explizite rekursiv definierte Tupel zur Darstellung des Systemzustandes verwendet. Dies ist schon bei der Definition der Estelle-Konstrukte sehr unübersichtlich, und es ist dann entsprechend für Zwecke der formalen Verifikation gänzlich ungeeignet.

Nachdem wir uns entschlossen hatten, den Automaten indirekt und auf prädikatenlogischem Wege zu beschreiben, stießen wir auf Lamports TLA. Sie erwies sich als sehr gut geeignet für diesen Zweck. Die Komponenten des Zustandes lassen sich auf die gewünschte Art als Variablen darstellen, und auch die Zustandsübergangsrelation läßt sich mit einer Sammlung von „Aktionen“ gut strukturiert spezifizieren. Es trat zwar das Problem auf, daß sich die Pascal-artigen Estelle-Transitionsrümpfe nicht als TLA-Aktionen darstellen ließen, aber dieses Problem haben wir in Kapitel 5 durch die Erweiterung der TLA zur TLA/PT behoben.

Wir beschreiben also die Semantik einer Zeichenkette in TLA/PT. Dabei bleiben zwei Probleme zu lösen. Erstens gibt es nicht zu jeder Zeichenkette eine Bedeutung in Estelle. Und zweitens brauchen wir eine Zuordnung von Zeichenketten und ihrer Bedeutung.

Das erste Problem ist das Problem der Syntax. Nur für Texte, die die Syntax von Estelle erfüllen, ist eine Semantik definiert. Daher ist die Frage, ob man entscheiden kann, für welche Zeichenketten dies der Fall ist. Am einfachsten zu entscheiden ist dies für den kontextfreien Anteil der Syntax. Man gibt hier üblicherweise ein Produktionensystem in erweiterter Backus-Naur-Form (EBNF) an, es gibt gute Algorithmen, die auf dieser Basis eine Entscheidung treffen können. Für Estelle enthält der ISO-Standard bereits eine Spezifikation in EBNF, und wir verweisen an dieser Stelle einfach nur auf den ISO-Standard, da das Problem dort bereits gelöst ist.

Die Syntax von Estelle enthält aber auch einen kontextsensitiven Anteil. Im ISO-Standard wird er ebenfalls beschrieben, allerdings ist diese Beschreibung in natürlicher Sprache abgefaßt und in Form von vielen Unterkapiteln mit dem Titel „Constraints“ eingestreut in die informelle Version der Definition von Estelle. Eine mathematisch-formale Beschreibung der kontextsensitiven Bedingungen existiert noch nicht.

Für eine vollständig formale Definition von Estelle wäre sie notwendig. Daß sie im ISO-Standard nicht vorhanden ist, hat seinen Grund darin, daß es sich um eine durchaus anspruchsvolle Aufgabe handelt. Und da eine vollständige formale Definition von Estelle nicht Gegenstand dieser Arbeit ist, sondern weit über den

gesetzten Rahmen hinausgehen würde, werden auch wir diese Formalisierung nicht durchführen. So bleibt uns nur, an dieser Stelle ebenfalls auf den ISO-Standard zu verweisen.

Die Frage, für welche Texte eine Bedeutung definiert ist, ist sogar noch unangenehmer. Denn es gibt Texte, die die kontextfreie und die kontextsensitive Syntax von Estelle erfüllen, und für die trotzdem keine Bedeutung definiert ist. Dies hängt von den Dingen ab, die, ausführungsorientiert betrachtet, „Laufzeitfehler“ sind. Zum Beispiel ist das Ergebnis einer Division durch Null undefiniert, und mit ihm ist die Bedeutung der ganzen betreffenden Estelle-Spezifikation undefiniert. Da bekanntermaßen nicht immer mit einem automatischen Verfahren entschieden werden kann, ob eine bestimmte Anweisung jemals zur Ausführung kommt, ist nicht immer statisch, das heißt ohne „Ausprobieren“, entscheidbar, ob ein Text eine Bedeutung hat.

Die Entwickler von Estelle haben solche „Laufzeitfehler“ bewußt ohne Bedeutung gelassen, um einerseits ihre Definition nicht mit einer aufwendigen Fehlerbehandlung zu belasten und andererseits, um einem Implementator die Freiheit zu lassen, selbst eine Bedeutungserweiterung nach seinen Wünschen zu entwerfen, sprich, eine geeignete Fehlerbehandlung zu implementieren.

Für unsere Definition der Bedeutung setzen wir voraus, daß derartige „Fehler“ nicht vorhanden sind. Will man also aufgrund unserer Definition Eigenschaften eines spezifizierten Systems ableiten, so muß man zuerst nachweisen, daß es wohldefiniert ist. Wie am Ende von Kapitel 5.1 erläutert, ist dazu sicherzustellen, daß alle Pascal-artigen Transitionsrümpfe bei Ausführung terminieren, und zusätzlich sind solche Dinge wie Divisionen durch Null auszuschließen. Letzteres kann im normalen Beweiskalkül stattfinden, da die TLA/PT alle arithmetischen Operationen als total erklärt und festlegt, daß das Resultat „illegaler“ Operationen völlig beliebig ist, so daß es kein Prädikat des Spezifikators mehr erfüllt, das eine Eigenschaft des Typs des Ergebnisses festlegt.

Jedenfalls bleibt festzuhalten, daß zwar für viele Texte entschieden werden kann, daß sie die Syntax von Estelle nicht erfüllen, daß es aber prinzipiell nicht möglich ist, ein automatisches Entscheidungsverfahren anzugeben, das genau die Texte erkennt, für die eine Bedeutung definiert sein soll.

Damit kommen wir zu dem oben erwähnten zweiten Problem, dem Problem der Zuordnung von Zeichenketten und ihrer Bedeutung. Mathematisch gesehen handelt es sich um eine partielle Funktion von Zeichenketten nach TLA/PT-Formeln. (Eine Funktion ist es, da wir sinnvollerweise nur höchstens eine Bedeutung für je eine Zeichenkette definieren wollen.)

Die Umkehrung der Funktion dagegen wird nicht eindeutig sein, da wir die gleiche Bedeutung erhalten wollen, auch wenn wir zum Beispiel einige weitere Leerzeichen in den Text einfügen oder wenn wir bestimmte Umordnungen vornehmen.

Die tatsächliche Definition der Bedeutungsfunktion ist eine anspruchsvolle Aufgabe. Als Weg schlagen wir ein analoges Vorgehen zu dem Vorgehen bei der Wiener Beschreibungssprache VDL ([LuLaSt70], [LuAlBa68]) vor, bei dem ein konkretes Programm der definierten Sprache in ein sogenanntes abstraktes Programm umgesetzt wird. Allerdings ist dort dieses abstrakte Programm eine Baumstruktur, die von einem (relativ komplizierten) abstrakten Interpreter interpretiert wird.

VDL wurde entwickelt, um die Semantik einer „gewöhnlichen“ Programmiersprache, PL/I, erstmals formal zu definieren. Auch die Semantik von Algol 60 wurde dann damit praktisch vollständig definiert ([Lau68]), viele weitere Arbeiten benutzen später diesen Formalismus (Beispiel: [Lam80]).

VDL geht bei der Umsetzung in ein abstraktes Programm in zwei Stufen vor. Zuerst wird eine kontextfreie Funktion `parse` definiert, die das Zerteilungsproblem (englisch: „parsing problem“) löst und einen EBNF-Ableitungsbaum liefert. Dabei schreiben die Autoren keinen speziellen Zerteilungsalgorithmus vor, sondern definieren nur seine Umkehrfunktion `generate`, die aus dem Ableitungsbaum wieder den Text erzeugen können muß. Falls der Text die kontextfreie Syntax nicht erfüllt, ist der Funktionswert der partiellen Funktion `parse` nicht definiert, dies läßt sich durch den Wahrheitswert eines speziellen, zugehörigen Prädikates erkennen.

Im zweiten Schritt wird dann eine kontextsensitive Funktion `translate` definiert, die einerseits die kontextsensitive Syntax überprüft und andererseits den Baum so umstrukturiert, daß anschließend der Interpreter damit arbeiten kann. Dies beinhaltet zum Beispiel, daß Typdeklarationen direkt den jeweiligen Verwendungen der zugehörigen Variablen zugeordnet werden, so daß der Interpreter nicht immer erst nach der richtigen Deklaration suchen muß.

Die Verkettung der Funktionen `parse` und `translate` ergibt dann die Übersetzungsfunktion von konkreten zu abstrakten Programmen.

Auch andere Autoren gehen bei der Übersetzung von konkreten zu abstrakten Programmen ähnliche Wege. [Pra73] wählt als Repräsentation des abstrakten Programmes nicht mehr Zeichenketten, sondern hierarchische Graphen, und zur Übersetzung verwendet er sogenannte Paar-Grammatiken. Das Übersetzungsverfahren durch Paar-Grammatiken läuft letztendlich wiederum auf eine Verkettung der bekannten Funktionen `parse` und `translate` hinaus, wobei dann allerdings auch `translate` kontextfrei ist, was die Leistungen des Übersetzungsvorganges einschränkt. [KiSøWo88] definiert die Semantik der Spezifikationstechik Z zwar nicht formal, aber es wird ebenfalls eine Umsetzung von konkreter in abstrakte Syntax definiert. Auch [Lau81][32] definiert - für andere Zwecke - eine entsprechende Umsetzung.

Für unsere Zwecke halten wir fest: Die Übersetzungsfunktion von Estelle-Texten nach TLA/PT-Formeln sollte als Verkettung von zwei Funktionen definiert werden. Die erste löst das Zerteilungsproblem, sie kann direkt als Umkehrung eines Generators, beruhend auf der bereits vorhandenen EBNF-Grammatik des ISO-Standards, definiert werden. Die zweite, kontextsensitive Funktion setzt dann den Ableitungsbaum in TLA/PT-Formeln um, wobei sie etliche Prüfungen und Transformationen ausführt.

Wir haben bereits oben erläutert, daß sowohl die formale Definition der Prüfung der kontextsensitiven Syntax als auch erst recht die vollständig formale Definition von Estelle nicht Gegenstand dieser Arbeit sind. Daher werden wir die Übersetzungsfunktion nicht vollständig angeben. Für unsere weitere Argumentation benötigen wir allerdings einige Eigenschaften dieser Funktion, und deshalb werden wir im Rest des Kapitels 6 anhand eines größeren Beispiels erläutern, wie sie arbeiten soll.

Unabhängig vom jeweiligen Text der Estelle-Spezifikation soll die Übersetzungsfunktion eine bestimmte Teil-Formel liefern, die für alle Spezifikationstexte gleich ist. Diese Formel beschreibt den „Kern von Estelle“, das Ausführungsmodell. Die Definition des Ausführungsmodells war das gesteckte Ziel dieser Arbeit (durchgeführt in Kapitel 7), und damit sich das Ausführungsmodell auf die Spezifikation beziehen kann, müssen wir wissen, wie sie in TLA/PT-Form dargestellt werden soll. Wie wir sehen werden, benötigen wir dazu nicht das gesamte Umsetzungsverfahren, sondern es reicht, die Eigenschaften bestimmter Komponenten des Ergebnisses anzugeben.

Weniger Estelle-spezifische Teile, wie etwa Prozedur- und Funktionsvereinbarun-

[32] Der Autor A. Laut ist nicht identisch mit P. E. Lauer.

gen, Parameter für Module und ähnliches werden wir auslassen. Aber wir werden in den Kapiteln 6.3 bis 6.7 Typ- und Variablenvereinbarungen (als Grundlage), die hierarchische Modulstruktur, Interaktionspunkte, die Kommunikationsstruktur und schließlich Estelle-Transitionen beleuchten.

Die äußere Form des Ergebnisses der Übersetzungsfunktion wird im wesentlichen eine große Konjunktion von Teilformeln sein. Der Übersichtlichkeit halber ist es unerläßlich, die Formel in Teilformeln über Teileigenschaften zu strukturieren. Auch sollte sich aus jedem syntaktischen Konstrukt, soweit möglich, eine getrennte Eigenschaft ergeben, um den intuitiven Zusammenhang mit der Estelle-Spezifikation zu wahren. Außerdem können bei Schlußfolgerungen über die Formel einzelne Konjunktoren, die nicht benötigt werden, gleich zu Anfang mit Hilfe des Prädikatenlogik-Axioms „$(A \wedge B) \Rightarrow A$“ fallengelassen werden, was die Verifikationsarbeit erheblich erleichtert.

Auf einen Punkt wollen wir bei der Darstellung der Semantik von Estelle in TLA/PT noch hinweisen. Die TLA/PT beschreibt (wie die TLA) den Zustand eines Systems ausschließlich mit Hilfe des Zustandes von Variablen. Daher hat die Entscheidung in Kapitel 4.6 für die TLA/PT zur Konsequenz, daß im folgenden nicht nur der Zustand von Estelle-Variablen mit Hilfe von TLA/PT-Variablen beschrieben wird, sondern ebenso werden dies damit auch alle anderen Teile des Zustands eines in Estelle spezifizierten Systems.

6.2 Ein Demonstrationsbeispiel

Um zu zeigen, wie die Beschreibung der Semantik eines Estelle-Textes in TLA/PT stattfinden soll, führen wir dies anhand eines Demonstrationsbeispiels vor. Es wird damit nichts sonderlich Sinnvolles spezifiziert, stattdessen haben wir versucht, möglichst viele Konzepte von Estelle in einer möglichst kurzen Spezifikation unterzubringen.

Entsprechend sind die Bezeichner dieser Spezifikation entgegen unserer sonstigen Gewohnheit nicht „sprechend“, da sie hier keine sinnvollen Objekte beschreiben. Der Text der Spezifikation findet sich in Abbildung 6-1.

```
Specification demo;
  Type
    ty1 = Char;
    ty2 = (el1, el2, el3);
  Channel ch1(user, provider);
    By user:
      conreq;
      datreq;
    By provider:
      conconf;
      disind;
  Ip
    ip0: ch1(user) Individual Queue;
  Module m1 Systemprocess;
    Ip
      ip1: ch1(provider) Individual Queue;
    Export
      var1: ty1;
  End; { Module m1 }
  Body b1 For m1;
    Const
      con1 = 0;
    Channel ch2(user, provider);
      By user:
        m_datreq;
      By provider:
        m_datind;
    Ip
      ip11: ch1(provider) Common Queue;
    Module m2 Activity;
      Ip
        ip2: ch2(user) Individual Queue;
    End; { Module m2 }
    Body b2 For m2;
      { (leer) }
    End; { Body b2 For m2 }
    Modvar
      mv2: m2;
    State
      ready,
      busy;
```

```
      Initialize
        To ready
          Begin
            Init mv2 With b2
          End;
      Trans
        From ready
          To busy
  Name tr1:
            Begin
            End;
    End; { Body b1 For m1 }
    Modvar
      mv1: m1;
    Initialize
      Begin
        Init mv1 With b1;
        Connect ip0 To mv1.ip1
      End;
  End. { Specification demo }
```

Abbildung 6-1: Ein Demonstrationsbeispiel in Estelle für die Definition der Semantik

Für die einzelnen Konstrukte des Beispiels werden wir nun in den nächsten Kapiteln darstellen, wie ihre Bedeutung von der Übersetzungsfunktion durch TLA/PT-Formeln ausgedrückt werden soll.

6.3 Beschreibung von Typvereinbarungen

In dem Beispiel aus Abbildung 6-1 finden sich die Typvereinbarungen

```
Type
  ty1 = Char;
  ty2 = (el1, el2, el3);
```

und die Deklaration einer (exportierten) Variablen:

```
Export
  var1: ty1;
```

Damit stellt sich die Frage, wie wir die Bedeutung von Estelle-Variablen darstellen, und wie wir die Eigenschaften beschreiben, die sie von ihrer Typdeklaration zugeordnet bekommen.

Estelle-Variablen können als Funktionen verstanden werden, die jeweils einem Zustand einen Wert zuordnen. Diese Auffassung gilt ebenfalls für TLA/PT-Variablen (siehe Kapitel 5.1), so daß wir grundsätzlich Estelle-Variablen direkt auf TLA/PT-Variablen abbilden können. Dies gilt zumindest solange, wir wir nur global deklarierte Estelle-Variablen betrachten und lokale Gültigkeitsbereiche außer acht lassen. Hierauf werden wir im Kapitel 6.4 über die Modulstruktur zurückkommen.

Im übrigen hilft uns eine derartige direkte Abbildung, den Zusammenhang zwischen dem Estelle-Text und der TLA/PT-Formel möglichst eng zu halten, damit der (menschliche) Anwender nicht den Überblick verliert.

Richten wir nun unser Augenmerk darauf, daß Estelle-Variablen bestimmte Eigenschaften besitzen, die ihnen ihre Typdeklaration vorschreibt. Die TLA/PT ist dagegen erst einmal typfrei. Aber wie Lamport in [Lam91], Kapitel 7.1.5, ausführt, hat dies lediglich zum Ziel, daß man die Überprüfung auf Typkorrektheit als Beweis in der Logik selbst ausführen kann. Sobald man die Bedingungen der Typkorrektheit in der Logik spezifiziert, kann man sie auch nachweisen (wenn sie denn erfüllt sind). Wenn der Nachweis einfach ist, kann man ihn von einer Maschine ausführen lassen, was einer Typprüfung in einer typisierten Logik entspricht, und wenn der Nachweis schwierig ist, dann ist er nur in der TLA beziehungsweise in der TLA/PT möglich. In einer typisierten Logik wäre dann die entsprechende Formel einfach nicht mehr ausdrückbar.

Daß der Wert der Variablen `x` zur Menge der natürlichen Zahlen gehört, drückt Lamport durch die Formel

$$x \in \mathbb{N}$$

aus.

Diese Idee findet sich auch bereits in [HoWi73], wo sie noch wesentlich weiter ausgeführt ist. Dort wurde die gesamte Semantik der Sprache Pascal formal definiert, und zwar auf der Basis der Hoareschen Zusicherungs-Tripel.

In [HoWi73] kommt zwar die obige Formel nur in Form einer umgangssprachlichen Beschreibung vor, da die Behandlung von Datentypen dort nicht restlos formalisiert ist, aber dafür werden die Eigenschaften der einzelnen Datentypen von Pascal ausführlich behandelt.

Als Beispiel stellen wir den Aufzählungstyp vor. Dabei definieren wir hier die Eigenschaften in TLA/PT und damit vollständig formal.

Sei ein Aufzählungstyp `aufz` folgendermaßen deklariert:

```
Type aufz = (a1, a2, ..., an)
```

Dann muß die Übersetzungsfunktion uns folgende Formel liefern:

```
□(  aufz = {a1, a2, ..., an}
  ∧ (∀i: i∈{1,...,n}:
        ( i≠n ⇒ ci+1=Succ(ci) )
      ∧ ( i≠n ⇒ ci=Pred(ci+1)) )
  ∧ (∀x: x∈aufz: (∀y: y∈aufz: (∀z: z∈aufz:
        ¬(x<x)
      ∧ ( (x<y) ∧ (y<z) ⇒ (x<z) )
      ∧ ( (x≠cn) ⇒ (x<Succ(x)) )
      ∧ ( x>y ≡ y<x )
      ∧ ( x≤y ≡ ¬(x>y) )
      ∧ ( x≥y ≡ ¬(x<y) )
      ∧ ( x≠y ≡ ¬(x=y) )))))
```

In dem oben angeführten Beispiel

```
Type
  ty2 = (el1, el2, el3)
```

ergibt sich damit:

```
□(  ty2 = {el1, el2, el3}
  ∧ el2=Succ(el1) ∧ el3=Succ(el2)
  ∧ el1=Pred(el2) ∧ el2=Pred(el3)
  ∧ (∀x: x∈ty2: (∀y: y∈ty2: (∀z: z∈ty2:
        ¬(x<x)
      ∧ ( (x<y) ∧ (y<z) ⇒ (x<z) )
      ∧ ( (x≠cn) ⇒ (x<Succ(x)) )
      ∧ ( x>y ≡ y<x )
      ∧ ( x≤y ≡ ¬(x>y) )
      ∧ ( x≥y ≡ ¬(x<y) )
      ∧ ( x≠y ≡ ¬(x=y) )))))
```

Diese Eigenschaften werden auf eine Variable übertragen, sobald sie als zu diesem Typ gehörig deklariert wird:

```
Var var2: ty2
```

Dies wird von der Übersetzungsfunktion übersetzt in:

```
□( var2 ∈ ty2)
```

Damit sind dann beide Formeln Bestandteil der Spezifikation, und man kann mit Hilfe der Schlußregeln leicht die Eigenschaften der Variablen `var2` ableiten. (Genauer gesagt: Die Eigenschaften des jeweiligen Wertes der Variablen.) Hat man etwa im Verlaufe seiner Arbeit das Prädikat

```
var2 > Succ(el1)
```

erhalten, so kann man es mit den Schlußregeln umformen in das Prädikat

```
var2 = el3
```

Den formalen Beweis überlassen wir dem Leser.

Außer Aufzählungstypen gibt es noch viele weitere Typen, in unserer Beispielspezifikation etwa die Deklaration, daß die Variable `var1` vom Typ `ty1` ist, der wiederum als äquivalent zum Typ **`Char`** deklariert ist:

```
Type
  ty1 = Char;

...

Export
  var1: ty1;
```

Hier muß die Übersetzungsfunktion folgende Formel erzeugen:

```
□(  ty1 = Char
  ∧ var1 ∈ ty1)
```

Weiterhin muß sie an alle ihre Formeln die Definition der Standardtypen anhängen. In [HoWi73] sind alle diese Definitionen beschrieben, man kann sie ohne Schwierigkeiten in TLA/PT übertragen. Da wir aber ohnehin nur das Konstruktionsprinzip der Übersetzungsfunktion zeigen wollen, ersparen wir uns diese Fleißarbeit.

Bei Verbund- und Feldtypen werden wir ein wenig von Hoares Ansatz abweichen, um die Arbeit mit Prädikatentransformatoren zu erleichtern. In Kapitel 5.3 hatten wir begründet, warum wir möglichst vermeiden wollen, Verbundvariablen ohne eine ihrer Selektorfunktionen in Formeln zu verwenden. Wir schreiben daher nicht

```
  □(∀α: α ∈ verbundtyp: ...)
∧ □(v ∈ verbundtyp)                    ,
```

sondern

```
  □(∀α: α.Typ ∈ verbundtyp: ...)
∧ □(v.Typ ∈ verbundtyp)                .
```

Die Verbundvariable `v` bekommt damit eine neue Komponente `Typ`. Die Eigenschaften dieser Komponenten lassen wir völlig offen, man kann nichts über sie ableiten, außer daß ihr Wert immer ein Element der Menge `verbundtyp` ist. Wir können höchstens indirekt auf einige Eigenschaften schließen, die die Komponente nicht hat: Wenn es zwei Verbundtypen gibt, deren Variablen Eigenschaften haben, die mit dem jeweils anderen Verbundtyp unvereinbar sind, dann müssen auch die Werte dieser Komponenten verschieden sein.

Verbundvariablen werden wir noch ausführlich anwenden (siehe Kapitel 6.4).

Der einzige Typ, der in [HoWi73] ausgelassen ist, ist der Typ `real`. Der Grund dafür liegt darin, daß die Definition dieser - in Anführungszeichen - „reellen“ Zahlen, äußerst schwierig ist. Denn schon die Algebra der mathematischen reellen Zahlen ist komplex, aber sie muß notgedrungenerweise noch verkompliziert werden durch das Zulassen einer endlichen Rechengenauigkeit, wie sie existierende Rechner haben.

Damit werden wir das Thema der Deklaration von Variablen vorläufig beschließen und verweisen als Grundlage für eine detaillierte Ausarbeitung auf [HoWi73].

6.4 Beschreibung der Modulstruktur

Wie in Kapitel 2 beschrieben, haben Estelle-Spezifikationen eine hierarchische Struktur. In diesem Kapitel werden wir zeigen, wie wir diese Struktur geeignet abbilden können. Dazu werden wir die in Kapitel 5.2 eingeführten strukturierten TLA/PT-Variablen verwenden.

Wir erinnern an Kapitel 2: In Estelle gibt es eine statische Hierarchie von Moduldefinitionen, die als generische Vorlagen dienen für die dynamische Erzeugung von Modulinstanzen. Dabei verweist nach der Erzeugung einer neuen Modulinstanz (zunächst) genau eine Modulvariable auf dieses neue Objekt. Die Beschreibung des Moduls legt die Eigenschaften der Modulinstanz fest, und die Modulvariable verweist auf das aktuell existierende Objekt, das einen bestimmten Zustand hat.

An dieser Stelle ist es notwendig, den Begriff „Objekt“ genauer zu betrachten. Um sich die Bedeutung des Begriffes „Modulinstanz“ anschaulich vorzustellen, könnte man versuchen zu sagen: „Die Modulvariable enthält ein Objekt (die Modulinstanz). Zum Beispiel durch Zuschicken einer Nachricht kann sich dieses Objekt verändern. Aber die Modulvariable enthält danach immer noch dasselbe Objekt.“

In dieser Formulierung wird recht deutlich, daß es etwas gibt, was sich mit der Zeit verändert, aber trotzdem durchgehend eine individuelle Identität besitzt. Im menschlichen Bereich erkennen wir (meistens) eine Person „als dieselbe“ wieder, auch wenn sie zwischenzeitlich die Kleidung gewechselt hat.

Aber genau mit dem eingeklammerten „meistens“ im letzen Satz ist ein tiefes Problem verbunden. Denn es stellt sich die Frage, was die Gleichheit von strukturierten Objekten bedeutet. Wenn wir einem der beschriebenen „dynamischen“ Objekte den Namen `A` geben, gilt dann noch die folgende Gleichung?

```
A = A
```

Es könnte ja sein, daß einige Komponenten des rechten `A` verschieden von denen des linken `A` sind. Bei näherem Hinsehen ergibt sich die Notwendigkeit von zwei verschiedenen Gleichheitsrelationen:

1) Alle Komponenten sind gleich.
2) Zumindest die innere „Identität“ ist gleich.

Bei diesen Betrachtungen muß man wohlgemerkt bedenken, daß diese Objekte *Werte* von Variablen sein sollen. „`A`“ soll also kein Variablenname sein, sondern Bezeichner für einen Wert, wie zum Beispiel auch „`True`“ dies ist. Und damit zeigt sich, daß in der Prädikatenlogik und der darauf basierenden temporalen Logik der Aktionen das Konzept von Objekten, die sich „im Inneren“ verändern, aber trotzdem ihre Identität behalten, nicht enthalten ist. Es ist in der TLA zwar möglich, daß man einer Variablen im Laufe der Zeit verschiedene Werte zuordnet, so daß der Wert einer Variablen in einem Zustand ungleich dem Wert der Variablen in einem anderen Zustand ist. Aber ein Wert ist immer mit sich selbst identisch. (Es gibt nur die erste Art von Gleichheit.)

Und es hat auch seinen guten Grund, daß man auf dieses Konzept verzichtet. Denn die Einfachheit der Prädikatenlogik erlaubt es, leicht Schlußfolgerungen zu ziehen. Würde man bereits an einer so grundlegenden Stelle wie der Gleichheit von Werten etwas einbauen, was in der Handhabung kompliziert ist, wäre der praktische Nutzen dieses Kalküls dahin.

Bei der Konstruktion der temporalen Logik der Aktionen ist es eines der wesentlichen Anliegen von Lamport, die einfache, gewöhnliche Prädikatenlogik als Grundlage zu behalten ([Lam91], [Lam91a]). Nur so bleibt es möglich, in der Praxis Schlußfolgerungen zu ziehen.

Um auf das vorhin erwähnte Beispiel der Wiedererkennung von Menschen zurückzukommen, die ihre Kleidung wechseln oder sich vielleicht gar völlig verkleiden: In der TLA kann man nicht von „Herrn Meyer“ sprechen, nur von „dem Mann mit Hut und Bart“. Möchte man Herrn Meyer gern Geld zurückzahlen, das man ihm schuldet, so tut man gut daran, aufgrund von weiteren Informationen abzusichern, daß die Verbindung zwischen „Hut und Bart“, der Namensbezeichnung und der Geldschuld invariant ist, damit man nicht einem Betrüger aufsitzt.

Man erkennt, daß die zugrundeliegende Logik sehr einfach ist, und daß die Identitätsprobleme mit ihrer Hilfe auf einer höheren, expliziten Ebene gelöst werden müssen. Dies kann immer noch schwer sein. Aber zumindest das Werkzeug ist leicht zu handhaben, und falls es sich nur um leichte Probleme handelt, sind sie auch leicht und ohne Ballast zu lösen.

Untersuchen wir nun also weiter, wie sich Estelle-Modulinstanzen ohne Zuhilfenahme von „dynamischen Objekten“ beschreiben lassen.

Man kann den Unterschied zwischen diesen „dynamischen“ Objekten und „gewöhnlichen“, statischen Objekten identifizieren mit einem Unterschied in der sogenannten „Referenzstufe“. Statische Objekte / statische Werte / Konstanten besitzen nach der Einteilung von [Eng88] die Referenzstufe 0. Variablen, die diese enthalten können, besitzen die Referenzstufe 1. Dies sind die gewöhnlichen (Programm-)Variablen. Variablen, die wiederum diese Variablen der Referenzstufe 1 enthalten können, besitzen die Referenzstufe 2. Diese bezeichnet man auch als Zeigervariablen, da sie auf Variablen der Stufe 1 zeigen. Die Hierarchie läßt sich beliebig fortsetzen, aber für praktische Zwecke unterscheidet man gewöhnlich Variablen höherer Stufen nicht mehr von denen der Stufe 2.

Nach dieser Einteilung hat zum Beispiel der Wert `True` die Stufe 0, eine boolsche Variable die Stufe 1 und eine Modulvariable, deren Modulinstanz aus der boolschen Variablen besteht, die Stufe 2. Die Modulvariable hat also eine analoge Semantik wie eine Zeigervariable. (Falls die referenzierte Modulinstanz weitere Modulvariablen enthält, entspricht dies einer Verkettung von Zeigern. Diese Verkettung muß aufgrund der Syntax von Estelle immer endlich und zyklenfrei sein.)

Hoare und Wirth beschreiben in ihrer axiomatischen Semantik von Pascal ([HoWi73]) Zeigertypen folgendermaßen: Ein Zeigertyp `ztypint` mit dem Grundtyp `Integer` besteht aus der beliebigen, unendlichen Menge von Werten `Nil`, $\varphi 1$, $\varphi 2$, ... Dazu wird eine Variable σ definiert, die die Komponenten $\sigma.\varphi 1$, $\sigma.\varphi 2$, ... hat, alle vom Typ `Integer`. Eine Variable ψ vom Typ `ztypint` dient dazu, das „letzte verwendete Element“ zu kennzeichnen. Für den Zugriff gilt folgende Eigenschaft:

$$x \neq \mathtt{Nil} \Rightarrow x\hat{} = \sigma.x$$

Bedauerlicherweise enthält [HoWi73] eine entscheidende Lücke: Die Zuweisungsoperation für Zeigervariablen wird nicht definiert. Und gerade sie schafft Probleme. Denn sobald man Ausdrücke der Form `x^` in Prädikaten zuläßt, erhält man das sogenannte Alias-Problem. Wenn die Zeigervariablen `x` und `y` auf dieselbe Variable zeigen und wenn `x^` ein neuer Wert zugewiesen wird, dann verändert sich ebenso der Wert von `y^`, ohne daß dies an der Struktur des Prädikates zu erkennen wäre. Daher kann die Zuweisungsoperation für Zeigervariablen nicht durch den üblichen Prädikatentransformator der einfachen Textersetzung beschrieben werden.

Beispiel: Der Ausdruck `x=y ∧ x^=5 ∧ y^=5` würde auf diese Weise durch die Operation `"x^:=7"` transformiert in die Nachbedingung `x=y ∧ x^=7 ∧ y^=5`, was offensichtlich äquivalent zu `7=5` ist.

Für die Modulvariablen von Estelle ergibt sich genau das gleiche Problem, wie fol-

gendes Beispiel zeigt: Der Modulvariablen `a` sei durch eine **`Init`**-Operation eine Modulinstanz zugeordnet worden, anschließend sei die Zuweisung `"b:=a"` ausgeführt worden. Die Modulinstanz von `a` besitze eine Variable `v`. Da beide Modulvariablen vom gleichen Typ sind, gilt dies ebenso für `b`. Nun führen wir eine Zuweisung `"a.v:=42"` aus, wodurch das Prädikat `a.v=42` wahr wird. Ebenso muß nun auch das Prädikat `b.v=42` wahr sein, obwohl `b` von der Zuweisungsoperation überhaupt nicht erwähnt worden war.

[JaEm77] hat das Problem der Referenzierung und der Zuweisung untersucht und eine Lösung angegeben, die wir hier ganz kurz andeuten wollen:

„Normale“ Logik besitzt eine einstufige Wertezuordnung, das heißt, daß den Variablen durch eine einzige Funktion die Werte zugeordnet werden. Wie gesehen behindert dies eine Darstellung von Zuweisungen mit Zeigern. Ebenso gibt es Schwierigkeiten mit bestimmten Zuweisungen auf Feldvariablen, zum Beispiel `"a[a[1]]:=2"`. Dadurch, daß ein Wert des Feldes als Index für dasselbe Feld verwendet wird, erhält man eine Art Simulation von Zeigern.

Andererseits ist [JaEm77] der Meinung, daß eine zweistufige Wertezuordnung (zum Beispiel: Variablen → Adressen, Adressen → Werte) bei einer Semantikdefinition nicht gut ist, da man damit interne Aspekte des Rechners in die Beschreibung der Bedeutung hineinbringt und man auf eine abstrakte Maschine zurückgreifen muß, die die Zuordnung von Adressen und Werten vornimmt. Der ISO-Standard für Estelle verwendet eine zweistufige Wertezuordnung, indem er eine Menge von abstrakten Speicherzellen definiert, auf die die Variablen zeigen, und die die Werte enthalten. Man erkennt schnell, daß dieses Konzept für die Verifikation sehr schlecht geeignet ist, da man in allen seinen Formeln ständig über Adressen und Speicherzellen reden muß, auch bei ganz gewöhnlichen Variablen, so daß der intuitive Zusammenhang zwischen Variablen und Werten verlorengeht.

[JaEm77] greift nun eine linguistische Arbeit von R. Montague ([Mon73]) auf. Dort wird eine sogenannte „intensionale Logik“ (IL) definiert. Die intensionale Logik ist eine Modallogik; eine Variable besitzt eine „Extension“, das heißt einen aktuellen Wert in der aktuellen Welt („R-value“), und sie besitzt eine „Intension“, eine Funktion, die diesen Wert für jede mögliche Welt angibt („L-value“). In dieser Logik wird anschließend für eine Teilmenge von ALGOL 68 die Semantik definiert. Im Grundsatz besteht die Lösung darin, daß in der Logik ein expliziter Dereferenzierungsoperator und ein expliziter Referenzierungsoperator eingeführt werden. Beispiel: Die ALGOL-Anweisung `"xx:=x"` (`x` ist vom Typ `Integer`, `xx` ist vom Typ Zeiger auf `Integer`) bewirkt, daß der Ausdruck ${}^{\mathtt{V}}xx{=}x$ wahr wird. Der Wert von ${}^{\mathtt{VV}}xx$ läßt sich nun *innerhalb* der Logik mit Hilfe des Wertes von ${}^{\mathtt{V}}x$ berechnen (denn ${}^{\mathtt{VV}}xx{=}{}^{\mathtt{V}}x$).

Insgesamt wird die Logik aber auch durch diese Erweiterung ein ganzes Stück komplizierter. Daher werden wir für die Beschreibung der Semantik von Estelle diesen Weg erst einmal nicht gehen, sondern Zeigervariablen auslassen. Da wir aber Modulvariablen auf keinen Fall auslassen können, müssen wir ihre Verwendung ein wenig einschränken:

Wir fordern, daß auf jede Modulinstanz genau eine Modulvariable zeigt.

Als Konseqenz verbieten wir damit Zuweisungen auf Modulvariablen, weil dadurch zwei Modulvariablen auf dieselbe Modulinstanz zeigen würden, und wir verbieten, daß mit Hilfe der **`Init`**-Operation eine neue Modulinstanz erzeugt und auf eine Modulvariable zugewiesen wird, solange diese noch auf eine alte Modulinstanz zeigt. Denn sonst würde auf die alte Modulinstanz hinterher gar keine Modulvariable mehr zeigen.

Im Ergebnis können wir Modulvariablen damit recht ähnlich wie normale Verbundvariablen behandeln. Allerdings sollen die Typdefinitionen für die Komponenten einer solchen Variablen (zum Beispiel Hauptzustand, Warteschlangen, weitere Variablen) nur gelten, wenn zu der Modulvariablen auch gerade eine Modulinstanz „existiert“. Wenn sie nicht „existiert“, darf jeder Zugriff auf eine Komponente, der trotzdem stattfindet, völlig unvorhersehbare Konsequenzen haben.

Damit haben wir die Begriffe der „Existenz“ und der „Nicht-Existenz“ einer Modulinstanz berührt. In einem gegebenen Zustand existiert zu jeder Modulvariablen eine Modulinstanz, oder sie existiert nicht. In der TLA/PT dagegen sind die Variablen und die Werte „ewig“, oder anders ausgedrückt, die Bedeutungsfunktion ordnet einem Paar aus einem Zustand und einer Variablen einen Wert zu. Wenn es in einem Zustand einen Wert für eine Variable gibt, gibt es dies in allen Zuständen. Ebenso ist die Menge der Werte unabhängig vom Zustand.

Daher beschreiben wir die „Existenz“ einer Modulinstanz *als Teil* des Zustandes. Eine Modulinstanz kann in zwei möglichen Grundzuständen sein, entweder sie existiert oder nicht. Dies drücken wir für eine Modulinstanz α durch eine boolsche Variable α`.Exists` aus.

Wenn eine Modulinstanz „nicht existiert“, heißt dies im Rahmen der TLA/PT, daß über sie in diesem Zustand keinerlei weitere Eigenschaften ableitbar sein sollen.

Die Übersetzungsfunktion muß damit eine Moduldefinition in etwas übersetzen, was einer Typdefinition für variante Verbunde entspricht. Die „Selektorkomponente“ α`.Exists` bestimmt alle weiteren Typeigenschaften. Sie müssen nur gelten, wenn dieser „Selektor“ den Wert `True` besitzt. (Den Typ des Selektors selbst müssen wir nicht weiter festlegen. Es ist egal, ob er den Wert `False` oder den Wert 0 hat. Allerdings wird er in einer korrekten Spezifikation immer vom Typ `Boolean` sein, da er nur zwischen `False` und `True` wechseln wird.)

Anmerkung: Wir hatten auch erwogen, die Existenz durch ein Enthaltensein in einer „Menge der existierenden Modulinstanzen“ darzustellen, wie es auch der ISO-Standard tut. Logisch wäre dies gleichwertig, aber die jetzige Lösung erwies sich in der Handhabung als günstiger. (Genau das gleiche wird sich bei der Darstellung der „zum Schalten ausgewählten Estelle-Transitionen“ ergeben, siehe Kapitel 6.7.)

Und man sollte festhalten: Auch wenn wir auf die eben beschriebene Weise nur eine etwas eingeschränkte Semantik der Modulvariablen beschreiben, bleibt es prinzipiell noch möglich, die Ideen von [JaEm77] später zu integrieren und die Semantikbeschreibung damit vollständig zu machen.

Nach diesen Überlegungen können wir nun angeben, wie wir konkret Module und Modulinstanzen beschreiben wollen.

Zur Gesamtspezifikation existiert immer genau eine Modulinstanz. Ihr können wir daher einen festen Namen geben, wir wählen den festen Namen `Sys` (einen besonders kurzen Namen, da er in den Formeln besonders häufig auftreten wird). Die Wahl eines festen Namens hat, wie sich zeigen wird, den Vorteil, daß das Ausführungsmodell sich so leichter auf den Gesamtsystemzustand beziehen kann. Denn aufgrund der hierarchischen Schachtelung der Modulinstanzen gibt es eine äußerste, oberste Modulinstanz, die das Gesamtsystem darstellt und alle weiteren Modulinstanzen direkt oder mittelbar als Sohn-Instanzen enthält.

Mit der Verbundvariablen `Sys` (siehe zu Verbundvariablen in der TLA/PT Kapitel 5.2) beschreiben wir den Zustand dieser speziellen Modulinstanz. Dafür geben wir ihr mehrere Komponenten. Für das Beispiel aus Kapitel 6.2 beschreiben wir mit dem Ausdruck `Sys.mv1` die (einzige) Modulvariable. Und mit `Sys.m1` beschreiben wir die Moduldefinition `m1`. Daher muß die Übersetzungsfunktion zum Beispiel die

Teilformel erzeugen:

```
□( (∀α: α.Typ ∈ Sys.m1: α.Exists ⇒ ...)
∧ Sys.mv1.Typ ∈ Sys.m1)
```

Wenn dieses Modul mit einem Modulrumpf `b1` instanziiert ist, dann muß es als Unterstruktur eine Modulvariable `mv2` geben, so daß die Übersetzungsfunktion erzeugen muß:

```
□( (∀α: α.Typ ∈ Sys.m1: α.Exists
    ⇒ ((α.Body = Sys.b1) ⇒ (α.mv2.Typ ∈ α.m2))
∧ Sys.mv1.Typ ∈ Sys.m1)
```

Wie man sieht, kennzeichnen wir durch den Zustand der Variablen `Sys.mv1.Body`, mit welchem Modulrumpf die Modulvariable instantiiert ist. Sämtliche Typeigenschaften der weiteren Variablen werden nur verlangt, solange diese Instantiierung anhält. Sollte die Modulinstanz vernichtet werden, so sind die Eigenschaften dieser Variablen undefiniert. Dies läßt Raum für verschiedene Arten der Implementierung eines Zugriffs auf eine solche Variable. Es ist möglich, eine Fehlerbehandlung wie bei einer Division durch Null einzubauen, es ist aber auch möglich, eine Instanz durch einen Speicherbereich darzustellen, der auch nach der Freigabe weiter existiert und auf den weiterhin zugegriffen werden kann, der aber später wieder für eine neue Instantiierung verwendet wird.

Einen weiteren wichtigen Punkt können wir an dieser Stelle beleuchten: Indem wir eine spezielle TLA/PT-Variable `xxx.Body` einführen, schaffen wir prinzipiell eine Möglichkeit zur Kollision mit dem Übersetzungsergebnis der vom Benutzer definierten Estelle-Variablen. In diesem Falle kann eine solche Kollision noch nicht passieren, weil `Body` ein reserviertes Schlüsselwort ist, das nicht für Variablenbezeichner verwendet werden darf, aber wir werden noch weitere feste Variablennamen einführen, bei denen dieses Problem in der Tat auftreten könnte.

Unsere Lösung besteht darin, daß die Übersetzungsfunktion alle Bezeichner des Benutzers in Kleinbuchstaben umwandeln muß. Dies ist ohne Probleme möglich, da Estelle (wie Pascal) nicht zwischen Groß- und Kleinschreibung unterscheidet. Alle von uns definierten Bezeichner schreiben wir dagegen im ersten Buchstaben groß. Und wir unterscheiden in unserer Logik zwischen Groß- und Kleinbuchstaben. (In den früheren Kapiteln haben wir aus diesem Grunde alle von uns dort eingeführten Bezeichner bereits im ersten Buchstaben groß geschrieben.) So kann es nicht mehr zu Namenskollisionen kommen.

Außer den Eigenschaften einer Modulinstanz, die vom aktuellen Modulrumpf abhängen, gibt es auch Eigenschaften, die davon unabhängig für jede Modulinstanz dieses Modultyps gelten. Ein Beispiel dafür ist die in Kapitel 2.1 erläuterte Einteilung in eine der Klassen **`SystemProcess`**, **`Process`**, **`System Activity`** oder **`Activity`**. Hier muß die Übersetzungsfunktion für jede Moduldefinition entsprechend Zeilen der Art

```
□( (∀α: α.Typ ∈ Sys.m1:
    α.Exists  ⇒  α.Class = SystemProcess))
```

erzeugen. Die Klassen kann sie direkt dem Spezifikationstext entnehmen. Falls kein Attribut spezifiziert ist, soll sie stattdessen `NonAttrib` einsetzen. (Zur Verwendung der Klassenattributierung siehe Kapitel 7.1.)

Damit wir das Ausführungsmodell definieren können, muß die Übersetzungsfunk-

tion zu den obigen Formeln noch eine weitere Information dazuliefern. Denn die Definition des Ausführungsmodells muß sich auf die Menge aller Sohn-Modulinstanzen zu einer Modulinstanz beziehen können. Die Moduldefinitionen, also sozusagen die Typen, dieser Menge sind die Vereinigung aller Moduldefinitionen in dieser Modulinstanz:

$\Box$(... $\wedge$ α.LocModDefs = α.moddef1 $\cup$ ... $\cup$ α.moddefn)

Im Beispiel aus Kapitel 6.2 reduziert sich dies zu:

$\Box$(Sys.LocModDefs = Sys.m1)
$\wedge$ $\Box$(... $\wedge$ α.LocModDefs = α.m2)

Damit wird dann im Ausführungsmodell folgende Quantifizierung über alle Sohnmodulinstanzen möglich:

... $\wedge$ ($\forall\beta$: β.Typ $\in$ α.LocModDefs $\wedge$ β.Exists: ...)

Schließlich müssen wir uns noch um den Anfangszustand kümmern. In den Kapiteln 6.7 und 7 werden wir die Initialisierungstransitionen behandeln. Hier sei schon einmal erwähnt, daß im allerersten Zustand, dem sogenannten „präinitialen“ Zustand, noch keine Initialisierungstransition geschaltet hat und somit noch keinerlei Modulinstanz existiert. Wie bereits oben erwähnt, existiert allein die spezielle Modulinstanz des Gesamtsystems immer, und zwar mitsamt ihrer (Modul-)Variablen. Dies drücken wir dadurch aus, daß ihre Variablen von keinem „.Exists“-Selektor abhängen.

Um den ersten Zustand zu beschreiben, muß die Übersetzungsfunktion zu jeder dieser Modulvariablen mv die Teilformel

mv.Exists = False

erzeugen. Sie spezifiziert für den ersten Zustand, daß zu dieser Modulvariablen keine Modulinstanz existiert. (Man beachte, daß kein $\Box$-Operator vor der Teilformel steht.)

Bei der hierarchischen Strukturierung von Variablen- und Typdefinition kann ein allgemeines Problem entstehen: In einem inneren Block wird eine Variable verwendet, die in einem viel weiter außen liegenden Block definiert wird. Für unsere Notation ist dies keine Schwierigkeit, wir können einfach schreiben:

a.b.c.d.e.f $\in$ a.typ

Hier sind die früher erwähnten kontextsensitiven Eigenschaften der Übersetzungsfunktion gefordert, sie muß die korrekte Zuordnung von Typbezeichnern zu ihrer Definition herausfinden. Die Zuordnung ist statisch, und als Lösungsverfahren kommen all die Algorithmen in Betracht, die auch in herkömmlichen Compilern für diesen Zweck verwendet werden. Aber da wir die genauen Eigenschaften der Übersetzungsfunktion hier nicht definieren, geben wir auch keine Spezifikation der notwendigen Eigenschaften des Verfahrens an.

Um unsere Beschreibung der Modulstruktur anschaulicher zu machen, zeigen wir an dem Beispiel aus Kapitel 6.2, wie die Modulstruktur übersetzt werden soll:

```
□( Sys.Class = NonAttrib
∧ (∀α: α.Typ ∈ Sys.m1:
      α.Exists
    ⇒ ( α.Class = SystemProcess
      ∧ (α.Body = Sys.b1
         ⇒ ( α.con1 = 0
           ∧ (∀β: β.Typ ∈ α.m2:
                β.Exists
              ⇒ ( β.Class = Activity
                ∧ (β.Body = α.b2
                   ⇒ (...))
                ∧ β.LocModDefs = {}))
           ∧ α.mv2.Typ ∈ α.m2
           ∧ α.LocModDefs = α.m2))))
∧ Sys.mv1.Typ ∈ Sys.m1
∧ Sys.LocModDefs = Sys.m1)
```

Hieraus läßt sich sofort ableiten:

```
□( Sys.Class = NonAttrib
∧ Sys.mv1.Typ ∈ Sys.m1
∧ (Sys.mv1.Exists
    ⇒ ( Sys.mv1.Class = SystemProcess
      ∧ (Sys.mv1.Body = Sys.b1
         ⇒ ( Sys.mv1.con1 = 0
           ∧ Sys.mv1.mv2.Typ ∈ Sys.mv1.m2
           ∧ (Sys.mv1.mv2.Exists
              ⇒ ( Sys.mv1.mv2.Class = Activity
                ∧ (Sys.mv1.mv2.Body = Sys.mv1.b2
                   ⇒ (...))
                ∧ Sys.mv1.mv2.LocModDefs = {}))
           ∧ Sys.mv1.LocModDefs = Sys.m2))))
∧ Sys.LocModDefs = Sys.m1)
```

Ein derartiges Ableiten ist sinnvoll, wenn man die Eigenschaften der Variablen direkt braucht. Die Eigenschaften bei der Definition des Typs zu sammeln ist insbesondere sinnvoll, um die Formel kurz zu halten, wenn mehrere Variablen des gleichen Typs in ihr vorkommen.

6.5 Beschreibung von Interaktionspunkten

In Estelle können Modulinstanzen miteinander kommunizieren. Außer durch einen sehr eingeschränkten Zugriff auf gemeinsame Variablen geschieht dies vor allem durch Nachrichtenaustausch. Das grundlegende Konzept des „Interaktionspunktes“ beschreibt dabei die Schnittstelle, an der die Nachrichten ausgetauscht werden. Eine allgemeine Diskussion der Semantik dieses Konzepts findet sich in [Got92a], zusammen mit der Feststellung, daß über dieses architekturelle Konzept offensichtlich noch kein allgemeiner Konsens gefunden worden ist.

In Estelle ist die Auffassung die folgende: Eine Nachricht wird mit Hilfe eines Paares von Interaktionspunkten übertragen, das dazu durch einen Kanal verbunden sein muß. Die Interaktion ist asynchron, jedem Interaktionspunkt ist eine (FIFO-)Warteschlange zugeordnet, die die noch nicht vom Empfänger abgeholten Nachrichten aufbewahrt. Die Verbindungen zwischen den Interaktionspunkten, die Kanäle, können dynamisch verändert werden. Dabei beschreiben die Kanäle im wesentlichen nur, nach wohin eine Nachricht zu transportieren ist. Da die Übertragung atomar geschieht, gibt es in den Kanälen keine „unterwegs befindlichen“ Nachrichten, Nachrichten gehören niemals zum Zustand der Kanäle. Somit kann man den Zustand der Kanäle als simple Relation zwischen den aktuell verbundenen Interaktionspunkten beschreiben.

Der Zustand eines Interaktionspunktes besteht in der Warteschlange der noch nicht abgeholten Nachrichten (sowie einiger Verwaltungsinformationen über ihre Herkunft). In [Got92a] werden die Eigenschaften eines Interaktionspunktes in einer temporalen Logik, aber eigenschaftsorientiert, beschrieben. Unsere Beschreibung wird ein wenig operationaler ausfallen, da unsere formale Grundlage, die TLA/PT, außerdem auch die operationalen Aspekte von Estelle beschreiben können muß (siehe Kapitel 4.1). Wir werden den inneren Zustand eines Interaktionspunktes nicht indirekt mit den Mitteln seines in der Vergangenheit beobachtbaren äußeren Verhaltens, sondern direkt durch TLA/PT-Variablen beschreiben. Dafür erlaubt uns dies auch, ohne Aufwand die „Inspektion“ eines Interaktionspunktes (ohne Entnahme einer Nachricht) und die Estelle-spezifischen „Common Queues“ zu beschreiben, also Warteschlangen, die mehreren Interaktionspunkten gemeinsam sind.

In diesem Kapitel soll es zunächst darum gehen, den Zustand von Interaktionspunkten in TLA/PT zu beschreiben. Im nächsten Kapitel, 6.6, wird dann die Verbindungsstruktur in TLA/PT beschrieben, und in Kapitel 6.7 wird schließlich unter anderem erläutert, wie die Operationen für den Nachrichtenaustausch definiert werden können. Daher interessieren wir uns vorläufig nicht dafür, wie es zu Änderungen des Zustandes kommt.

Ein Interaktionspunkt ist jeweils einer Modulinstanz zugeordnet. Wir sehen seinen Zustand daher als Teil des Zustandes dieser Modulinstanz an. Eine Estelle-Kanalvereinbarung legt die Eigenschaften der zum Kanal gehörenden Interaktionspunkte fest, zum Beispiel die Menge der möglichen Nachrichten.

Aus diesem Grunde können wir Interaktionspunkte wiederum als eine besondere Art von Variablen (der Modulinstanz) betrachten und die Kanalvereinbarungen als die dazugehörigen Typdefinitionen.

Als Namen für die Variable verwenden wir den Namen des Interaktionspunktes, womit auch intuitiv ein enger Zusammenhang zwischen der Variablen und dem Interaktionspunkt hergestellt wird. Ein solcher enger Zusammenhang ist ganz allgemein anzustreben, damit man beim Verifizieren einen möglichst guten Überblick behält über das, was man eigentlich gerade nachweist.

Zur Vererbung von Kanalvereinbarungen an innere Blöcke gilt das Gleiche wie zur Vererbung von Typen an innere Blöcke allgemein: Die kontextsensitive Übersetzungsfunktion muß diese Zuordnung auflösen. (Siehe voriges Kapitel.)

Der zustandsabhängige Informationsgehalt eines Interaktionspunktes besteht in der Information über den aktuellen Zustand der zugeordneten Warteschlange für die angekommenen, aber noch nicht abgeholten Nachrichten. Daher ist unsere Variable, die den Interaktionspunkt beschreibt, vom Typ „unbegrenzt lange Warteschlange“. Je ein Eintrag in der Schlange enthält je eine Nachricht und außerdem noch einige weitere Informationen, die unter anderem für die Umsortierung der Nachrichten beim Auflösen von Kommunikationsverbindungen wichtig sind. Die genaue Definition dieses Typs und der darauf definierten Operationen ist natürlich noch vorzunehmen.

Eine Sonderbehandlung müssen wir nur für die als **Common Queue** definierten Interaktionspunkte durchführen, da bei ihnen nur eine einzige Warteschlange für alle derartigen Interaktionspunkte einer ganzen Modulinstanz existiert. Hierzu definieren wir für jede Modulinstanz eine Variable mit dem Namen `LocalCommonQueue`, die diese Warteschlange beschreibt.

Die Estelle-Spezifikation in Kapitel 6.2 enthält folgende Kanalvereinbarung:

```
Channel ch1(user, provider);
  By user:
    conreq;
    datreq;
  By provider:
    conconf;
    disind;
```

Die Übersetzungsfunktion soll nun hierzu eine Typdefinition erzeugen. Zuerst müssen wir sicherstellen, daß die Nachrichten jedes Kanals Konstanten und wohlunterschieden sind:

```
  □ Unchanged Sys.conreq
∧ □ Unchanged Sys.datreq
∧ □(Sys.conreq  ≠ Sys.datreq)
∧ □ Unchanged Sys.conconf
∧ □ Unchanged Sys.disind
∧ □(Sys.conconf ≠ Sys.disind)
```

Anmerkung: Daß es sich um Konstanten handeln soll, ließe sich auch durch die Verwendung von starren Variablen ausdrücken. Aber dazu müßten diese deklariert werden können, wozu wir die Erweiterungen der TLA$^+$ benötigen würden.

Die restliche Typdefinition kann die Übersetzungsfunktion durch den folgenden Ausdruck beschreiben:

```
□ Channel(Sys.ch1,Sys.user,     {Sys.conreq, Sys.datreq},
                  Sys.provider,{Sys.conconf,Sys.disind})
```

Um den Ausdruck zu strukturieren, haben wir die jetzt folgenden Definitionen verwendet. Erläuterungen dazu folgen dann anschließend.

```
Channel(ch, role1, msgset1, role2, msgset2)
≜
  ch.Roles = {role1, role2}
∧ OnewayChannel(role1, msgset2)
∧ OnewayChannel(role2, msgset1)
```

```
OnewayChannel(role, opp_msgset)
≜
  (∀π: Ip_id.π ∈ role  ∧  Ip_id.π ∈ IndividualQueue:
       CompSet(π) ⊆ {Ip_id.π} × role × opp_msgset )
∧ (∀α:: (∀ψ: Ip_id.α.ψ ∈ role  ∧  Ip_id.α.ψ ∈ CommonQueue:
             (∀ω: ω ∈ CompSet(α.LocalCommonQueue):
                  Dest_ip(ω) = Ip_id.α.ψ
                  ⇒ ω ∈ {Ip_id.α.ψ} × role × opp_msgset )))

CompSet( <x1,...,xn> ) ≜ {x1} ∪ ... ∪ {xn}

Dest_ip( (d_ip_id,e_ip_id, msg) ) ≜ d_ip_id
```

Um die kompakten Formeln leichter verständlich zu machen, wollen wir ihren Inhalt noch einmal erläutern. Die Definition `Channel()` liefert für eine Kanalvereinbarung die komplette Typdefinition, abgesehen von der obigen Definition der Nachrichten. Der erste Parameter der Definition ist der Name der (strukturierten) TLA/PT-Variablen, die den Kanal beschreiben soll. Der zweite Parameter beschreibt die erste der beiden „Rollen"-Variablen. Eine Rollen-Variable beschreibt die Menge der Kennungen derjenigen Interaktionspunkte, die diese Rolle übernehmen können. (Sozusagen die Interaktionspunkte mit dem passenden Typ.) Der dritte Parameter beschreibt die Menge der für diese Rolle sendbaren Nachrichten.

Der vierte und fünfte Parameter entsprechen dem zweiten und dritten, nur für die entgegengesetzte Rolle.

In der Definition von `Channel()` werden die erste Rolle und die zweite Nachrichtenmenge einander zugeordnet und umgekehrt, da jeder Interaktionspunkt genau die sendbaren Nachrichten der entgegengesetzten Rolle empfangen und zwischenspeichern kann.

In dieser Definition sieht man weiterhin, daß die Komponente `ch.Roles` festgelegt wird. Damit kann das Ausführungsmodell später über alle beiden Rollen eines Kanals quantifizieren. Die weitere Definition des Kanals ist spiegelsymmetrisch für die beiden Rollen, weshalb wir die Definition für eine Richtung mit der nächsten Definition `OnewayChannel()` herausgezogen haben.

Für `OnewayChannel()` wiederum werden zwei kleine Hilfsdefinitionen benötigt: `Dest_ip()` liefert die Selektorfunktion, die von einem Tripel die erste Komponente auswählt. Wie wir gleich sehen werden, enthalten die Warteschlangen Tripel, und deren erste Komponente beschreibt den Ziel-Interaktionspunkt für eine Nachricht. Die Definition `CompSet()` nimmt als Parameter ein Tupel mit einer beliebigen Anzahl von Komponenten (weshalb wir diese Art von Tupeln in spitzen Klammern notieren), und liefert eine Menge, die als Elemente eben diese Komponenten enthält. Beispiel:

```
CompSet(<a,b,a,c,b,d,e>) = {a,b,c,d,e}
```

Dies wird benötigt, um über die Menge der in einer Warteschlange wartenden Nachrichten reden zu können. Damit werden dann Typ-Aussagen über den Warteschlangeninhalt möglich.

Kommen wir aber nun zur Definition `OnewayChannel()` zurück:

Die strukturierte Variable `Ip_id` enthält als Selektoren die Namen der Interaktionspunkte, und ihre Werte sind die Identifikatoren ebendieser. (Da eine Interaktionspunkt-Variable bereits eine Warteschlange beschreibt, können wir sie nicht mehr als strukturierte Variable definieren und ihr eine Komponente `.Id` geben. Daher

diese „Umkehrung“.) Der Ausdruck „`Ip_id.`$\pi \in$ `role`“ beschreibt, daß ein Interaktionspunkt eine bestimmte Rolle für eine bestimmte Art von Kanal spielen muß.

Weiterhin muß die Übersetzungsfunktion Teilformeln erzeugen, die sicherstellen, daß alle Interaktionspunkt-Identifikatoren `Ip_id.xxx` Konstanten und jederzeit verschieden sind, analog zu den Ausdrücken, die wir oben für die Wohlunterschiedenheit der Nachrichten angegeben hatten.

Nun erkennen wir in der Definition von `OnewayChannel()` die Unterscheidung zwischen Interaktionspunkten mit eigener Warteschlange und Interaktionspunkten der Art **`Common Queue`**. Zu jeder Vereinbarung eines Interaktionspunktes soll die Übersetzungsfunktion entweder eine Teilformel der Art

$\Box$`(Ip_id.Sys.ip42` $\in$ `IndividualQueue)`

oder eine Teilformel der Art

$\Box$`(Ip_id.Sys.ip42` $\in$ `CommonQueue)`

erzeugen. Je nach Art des Interaktionspunktes ergeben sich damit aus der Typdefinition dann jeweils die richtigen Eigenschaften. Diese Eigenschaften bestehen in der Festlegung der Typen der Einträge der Warteschlange. Die Definition `CompSet()` gibt dabei jeweils die Menge der in einer Warteschlange aktuell vorhandenen Einträge an, und diese Menge wird Bedingungen unterworfen.

Für Interaktionspunkte der Art **`Individual Queue`** ist ein Eintrag der Warteschlange ein Tripel, dessen erste Komponente immer die Kennung dieses Interaktionspunktes ist (also eine redundante Information), dessen zweite Komponente die Kennung eines Interaktionspunktes mit der selben Rolle ist (zu deren Sinn wir gleich noch kommen werden), und dessen dritte Komponente eine der jeweils erlaubten Nachrichten ist.

Für Interaktionspunkte der Art **`Common Queue`** ist die Definition etwas komplizierter. Zu jeder Modulinstanz α existiert eine Warteschlange α`.LocalCommonQueue`. Diese gemeinsame Warteschlange enthält ebenfalls Tripel. In der ersten Komponente ist der Ziel-Interaktionspunkt nun nicht mehr redundant. Für alle aktuellen Einträge in der Warteschlange (beschrieben durch „`(`$\forall\omega$`:` $\omega \in$ `CompSet(`α`.LocalCommonQueue): ...)`“) muß gelten, daß die zur Rolle des jeweiligen Ziel-Interaktionspunktes passenden Nachrichten und Interaktionspunkte im dritten und zweiten Parameter vorhanden sind.

Die Warteschlangeneinträge werden für beide Arten von Interaktionspunkten gleich definiert, obwohl die erste Komponente des Tripels für die erste Art redundant ist, da auf diese Weise eine einheitliche Behandlung möglich wird. Die Definition der Estelle-Operationen, die die Warteschlangen manipulieren, müssen so keine Fallunterscheidung machen.

Die zweite Komponente eines Warteschlangeneintrages wird wichtig, wenn Interaktionspunkte einer Vater- und einer Sohn-Modulinstanz mit der Operation **`Attach`** verbunden und mit der Operation **`Detach`** wieder getrennt werden. Denn wenn die Warteschlange eines Interaktionspunktes `ip5` einen Nachrichteneintrag enthält und dieser Interaktionspunkt mittels **`Attach`** mit einem externen Interaktionspunkt `ip6` einer Sohnmodulinstanz verbunden wird, so wird dieser Eintrag aus der Warteschlange von `ip5` entfernt und an die Warteschlange von `ip6` angefügt. Wird danach aber diese Verbindung wieder aufgelöst, so wandert der Eintrag wieder zurück, die Modulhierarchie hinauf.

Dabei soll er aber bei weiteren Vebindungsauflösungen niemals weiter in einer **`Attach`**-Hierarchie hinaufgereicht werden, als er ursprünglich in diese Hierarchie über eine durch die Operation **`Connect`** hergestellte Verbindung „eingestiegen“ ist.

Hierfür ist nun die zweite Komponente des Tripels vorgesehen. Sie kennzeichnet den „Einstiegs"-Interaktionspunkt.

Die genaue, formale Beschreibung dieser Nachrichtentransport- und Nachrichtenumsortierungsvorgänge ist dann allerdings Teil der Beschreibung der einzelnen Estelle-Operationen und findet deshalb hier nicht statt.

Die vollständige Definition des Typs „Warteschlange" samt der Operationen darüber werden wir hier ebenfalls nicht ausführen, sondern nur so viel darüber sagen, wie wir für unsere Argumentation benötigen. Geeignete Definitionen von Warteschlangen sind in vielen Lehrbüchern nachzulesen. Die Übersetzungsfunktion muß in die Ergebnisformel eine Beschreibung des Typs Warteschlange einfügen, wie wir dies schon in Kapitel 6.3 beschrieben haben. Wir notieren eine solche Warteschlange, die gerade die Einträge `x1`, `x2`, `x3` und `x4` enthält, als `<x1,x2,x3,x4>`. Dabei wird `x1` als erster Eintrag abgeholt, `x4` war der zuletzt hinzugefügte Eintrag. (Die Warteschlange mit ihren Operationen soll FIFO-Verhalten zeigen.) Formal ist eine Warteschlange ein Tupel mit gleichartigen Komponenten, wobei die Anzahl der Komponenten von Zustand zu Zustand schwanken kann. Um diese besondere Eigenschaft deutlicher zu machen, notieren wir das Tupel mit spitzen Klammern und nicht wie sonst mit runden Klammern.

Der ISO-Standard verwendet zur Definition der Semantik von Estelle ebenfalls Warteschlangen, und er nimmt diese samt ihrer Grundoperationen als bekannt an. (Kapitel 9.2, S.65)

Ansonsten geht der ISO-Standard aber etwas anders vor als wir. (Vergleiche dort S. 96f.) In der Definition des ISO-Standards senden externe Interaktionspunkte Tripel:

```
(Absender-IP, Nachricht, Parameter-Vektor)
```

Diese Tripel werden nicht „sofort" zugestellt, sondern sie werden erst in einer Sammelaktion am Ende einer Transition zugestellt, damit sie nach außen hin atomar sichtbar werden. Nach Anwendung des Zustellalgorithmus′ (siehe Kapitel 9.5.4) werden aus den Tripeln Quadrupel:

```
(Ziel-IP, aktueller Connected-IP, Nachricht, Parameter-Vektor)
```

Interne Interaktionspunkte senden dagegen gleich diese Quadrupel, die bei ihnen dem Ziel-Interaktionspunkt sofort zugestellt werden. Insgesamt ist die Beschreibung der Nachrichtenübertragung im ISO-Standard äußerst unübersichtlich.

Aufgrund der operationalen Beschreibung des Pascal-Anteils der Estelle-Transitionen ist die Darstellung der Atomizität von Transitionen recht aufwendig. Denn die operationale Darstellung bringt es mit sich, daß die Estelle-Transitionen in einer Art Mikroschritte abgearbeitet werden, daß aber die Wirkung dieser Mikroschritte sich nicht mit der Wirkung der Mikroschritte von Estelle-Transitionen in anderen Modulinstanzen verzahnen darf (S. 82 o.). Dies ist der Grund dafür, daß im ISO-Standard die Nachrichten zuerst gesammelt werden müssen, bevor sie von externen Interaktionspunkten an andere Modulinstanzen verschickt werden können.

Wenn nämlich zwei Modulinstanzen jeweils zwei Nachrichten versenden, können alle vier Nachrichten in derselben Warteschlange abgelegt werden. Dies kann dann geschehen, wenn die Empfänger-Interaktionspunkte von der Art **`Common Queue`** sind und eine gemeinsame Warteschlange besitzen. In diesem Falle sind nicht alle Ankunftsreihenfolgen erlaubt, da das weitere Verhalten der Empfänger-Modulinstanz von der Reihenfolge abhängen kann. Beispiel:

Erlaubt:	Verboten:
<a1,a2,b1,b2>	<a1,b1,b2,a2>
<b1,b2,a1,a2>	<b1,a1,b2,a2>

Wenn jede der vier Nachrichten von einer eigenen Estelle-Transition an einem eigenen Interaktionspunkt akzeptiert wird, dann kann der Zustand der Empfänger-Modulinstanz schließlich durchaus von der Reihenfolge abhängen, in der seine vier Estelle-Transitionen geschaltet haben.

Auf den ersten Blick scheint es, als sei das Problem auf Warteschlangen der Art **Common Queue** beschränkt. Aber man kann auch Beispiele konstruieren, in denen es bei getrennten Warteschlangen zu Unterschieden im Verhalten kommt: Nehmen wir an, daß eine Estelle-Transition zwei Nachrichten aussendet, und daß diese durch das Schalten von zwei anderen Estelle-Transitionen empfangen werden können, die einer gemeinsamen Modulinstanz angehören. Durch ein **Priority**-Konstrukt habe dabei die erste Vorrang vor der zweiten, und wenn eine von beiden geschaltet habe, sei aufgrund entsprechender `To`-Klauseln keine von ihnen mehr schaltbereit. Wenn beide Nachrichten atomar und gleichzeitig eintreffen, dann kann die zweite Estelle-Transition niemals schalten. Wenn aber dazwischen noch mindestens ein Zustand liegt, kann auch die zweite möglicherweise zum Zuge kommen. Ob sie es tatsächlich tut, ist einer Implementation (und ihrer Geschwindigkeit) freigestellt, aber es würden offensichtlich auch hier Ausführungsfolgen möglich, die nicht zulässig sein sollen. (Das Beispiel mit **Common Queue** ist allerdings augenfälliger, man kann die unterschiedliche Reihenfolge der Nachrichten in der gemeinsamen Warteschlange schön anschaulich und kurz notieren.)

Ohne Nachrichten und Warteschlangen würde es in Estelle keinen Unterschied für die weitere Entwicklung des Zustandes machen, ob die Wirkung von Estelle-Transitionen atomar eintritt oder ob sie in vielen kleinen Schritten eintritt. Denn die Vorrangregelung zwischen Vater- und Sohn-Modulinstanzen sorgt zusammen mit den Einschränkungen für den Gebrauch von gemeinsamen Variablen auf den Zugriff auf Sohn-Variablen dafür, daß sonst keine „Zwischenzustände“ einer Implementation inspiziert werden können.

Daß die Übertragung von Nachrichten so viel zusätzlichen Aufwand für eine Implementation notwendig macht, liegt daran, daß diese über die Grenzen von „Systemprozessen“ und „Systemaktivitäten“ hinaus übertragen werden können. Hierdurch ist erst die volle uneingeschränkte Parallelität möglich, und diese fordert natürlich auch uneingeschränkte Atomizität „ohne Tricks“.

Immerhin läßt sich diese gesamte Problematik mit Hilfe unserer Prädikatentransformatoren, die atomar schaltende Aktionen ausdrücken, wesentlich eleganter beschreiben als durch die operationale Methode, die im ISO-Standard verwandt wird.

Damit kommen wir nun auch wieder zu unserer eigenen Beschreibung der Semantik von Estelle zurück.

Wir machen die obige Unterscheidung zwischen „sofortiger“ und „nicht sofortiger“ Zustellung nicht, da unsere Beschreibung der Estelle-Transitionen durch Prädikatentransformatoren ein atomares Schalten ohnehin direkt ausdrückt. Wir bleiben damit auf der konzeptuellen Ebene und brauchen keinen Algorithmus anzugeben, mit dem die Atomizität einer Implementation sichergestellt werden kann. Hier eine geeignete Wahl zu treffen, überlassen wir der Freiheit des Implementators.

In der vorhin angeführten Definition von `OnewayChannel()` beschrieben wir die Einträge einer Warteschlange immer als Tripel. Im Falle eines als **Individual Queue** deklarierten Interaktionspunktes π hieß dies:

```
CompSet(π) ⊆ {Ip_id.π} × role × opp_msgset
```

(Dabei beschrieb die Menge `CompSet`, wie oben definiert, die *Menge* der zur Zeit in der Warteschlange enthaltenen Einträge.)

Die dritte Komponente des Tripels enthielt jeweils die eigentliche Nachricht. Um auch parametrisierte Nachrichten darstellen zu können, legen wir nun zusätzlich fest, daß eine Nachricht auch strukturiert sein kann. Ein Beispiel: (Es ist nicht aus Kapitel 6.2 entnommen.) Die Nachrichtenvereinbarung

```
c_count_result(c: Char; num: Integer)
```

hat zur Folge:

```
  c_count_result ∈ msgset
∧ c_count_result.c ∈ Char
∧ c_count_result.num ∈ Integer
```

Zum Abschluß dieses Kapitels stellen wir alle betreffenden Teilformeln zusammen, die sich aus dem Beispiel in Kapitel 6.2 ergeben:

```
  □ Unchanged Sys.conreq
∧ □ Unchanged Sys.datreq
∧ □(Sys.conreq  ≠ Sys.datreq)
∧ □ Unchanged Sys.conconf
∧ □ Unchanged Sys.disind
∧ □(Sys.conconf ≠ Sys.disind)
∧ □ Channel(Sys.ch1,Sys.user,    {Sys.conreq, Sys.datreq},
                    Sys.provider,{Sys.conconf,Sys.disind})
∧ □(Ip_id.Sys.ip0 ∈ Sys.user)
∧ □(Ip_id.Sys.ip0 ∈ IndividualQueue)
∧ □(∀α: α ∈ Sys.m1:
      Ip_id.Sys.α.ip1 ∈ Sys.provider
    ∧ Ip_id.Sys.α.ip1 ≠ Ip_id.Sys.ip0
    ∧ Ip_id.Sys.α.ip1 ∈ IndividualQueue
    ∧ ( α.Body = Sys.b1
      ⇒ ( Unchanged α.m_datreq
        ∧ Unchanged α.m_datind
        ∧ Channel(α.ch2,α.user,    {α.m_datreq},
                        α.provider,{α.m_datind})
        ∧ Ip_id.Sys.α.ip11 ∈ Sys.provider
        ∧ Ip_id.Sys.α.ip11 ∉ {Ip_id.Sys.ip0, Ip_id.Sys.α.ip1}
        ∧ Ip_id.Sys.α.ip11 ∈ CommonQueue
        ∧ (∀β: β ∈ α.m2:
             Ip_id.Sys.β.ip2 ∈ α.user
           ∧ Ip_id.Sys.β.ip2 ∉ {Ip_id.Sys.ip0, Ip_id.Sys.α.ip1,
                                Ip_id.Sys.α.ip11}
           ∧ Ip_id.Sys.β.ip2 ∈ IndividualQueue)))
    ∧ α.mv2 ∈ α.m2)
∧ □(Sys.mv1 ∈ Sys.m1)
```

Setzt man die Definition für `Channel()` ein, kann man unter anderem sofort daraus ableiten:

```
  □ Unchanged Sys.conreq
∧ □ Unchanged Sys.datreq
∧ □(Sys.conreq  ≠ Sys.datreq)
∧ □ Unchanged Sys.conconf
∧ □ Unchanged Sys.disind
∧ □(Sys.conconf ≠ Sys.disind)
∧ □(Sys.ch1.Roles = {Sys.user, Sys.provider})
∧ □(Ip_id.Sys.ip0 ∈ Sys.user)
∧ □(Ip_id.Sys.ip0 ∈ IndividualQueue)
∧ □(CompSet(ip0)
    ⊆ {Ip_id.Sys.ip0} × Sys.user × {Sys.conconf, Sys.disind})
∧ □(∀α: α ∈ Sys.m1:
      Ip_id.Sys.α.ip1 ∈ Sys.provider
    ∧ Ip_id.Sys.α.ip1 ≠ Ip_id.Sys.ip0
    ∧ Ip_id.Sys.α.ip1 ∈ IndividualQueue
    ∧ CompSet(α.ip1)
      ⊆ {Ip_id.Sys.α.ip1} × Sys.provider × {Sys.conreq, Sys.datreq}
    ∧ ( α.Body = Sys.b1
      ⇒ ( Unchanged α.m_datreq
        ∧ Unchanged α.m_datind
        ∧ α.ch2.Roles = {α.user, α.provider}
        ∧ Ip_id.Sys.α.ip11 ∈ Sys.provider
        ∧ Ip_id.Sys.α.ip11 ∉ {Ip_id.Sys.ip0, Ip_id.Sys.α.ip1}
        ∧ Ip_id.Sys.α.ip11 ∈ CommonQueue
        ∧ (∀ω: ω ∈ CompSet(α.LocalCommonQueue):
              Dest_ip(ω) = Ip_id.Sys.α.ip11
              ⇒ ω ∈ {Ip_id.Sys.α.ip11} × Sys.provider
                    × {Sys.conreq, Sys.datreq} )
        ∧ (∀β: β ∈ α.m2:
              Ip_id.Sys.β.ip2 ∈ α.user
            ∧ Ip_id.Sys.β.ip2 ∉ {Ip_id.Sys.ip0, Ip_id.Sys.α.ip1,
                                 Ip_id.Sys.α.ip11}
            ∧ Ip_id.Sys.β.ip2 ∈ IndividualQueue
            ∧ CompSet(β.ip2)
              ⊆ {Ip_id.Sys.β.ip2} × α.user × {α.m_datind})))
    ∧ α.mv2 ∈ α.m2)
∧ □(Sys.mv1 ∈ Sys.m1)
```

6.6 Beschreibung der Kommunikationsstruktur

Zum Zustand eines Estelle-Systems gehört die Information, wie die einzelnen Interaktionspunkte gerade aktuell verbunden sind. Estelle erlaubt hier eine dynamische Veränderung der Verbindungen.

Insofern müssen wir hier weniger beschreiben, was die Übersetzungsfunktion aus dem Spezifikationstext erzeugen soll, sondern mehr, wie wir den Zustand beschreiben wollen, den dann die Estelle-Operationen verändern. Da aber eine enge Verwandtschaft mit der Definition der Interaktionspunkte besteht, lassen wir diese Beschreibung nun direkt im Anschluß folgen.

In jedem Zustand ist die Kommunikationsstruktur eine Relation zwischen den vorhandenen Interaktionspunkten. Daher ist es in unserem formalen Rahmen sinnvoll, sie auch als Relation, das heißt als Menge von Paaren, darzustellen. Um allgemeinere Aussagen zu erhalten, kann man dann darüber Prädikate bilden.

Es stellt sich die Frage, ob es sinnvoller ist, für jede Modulinstanz hierarchisch absteigend jeweils eine eigene Relation in Form einer TLA/PT-Variablen zu definieren oder besser eine gesamte Relation über die Gesamtspezifikation. Da es für viele Operationen (zum Beispiel **`Output`**) notwendig ist, End-zu-End-Verbindungen ausdrücken zu können, stellt es sich als günstiger heraus, eine gesamte Relation für alle Modulinstanzen zusammen zu definieren.

Bei genauerem Hinsehen müssen wir allerdings zwei Relationen definieren. Denn es gibt zwei verschiedene Arten von Kommunikationsverbindungen. Erstens gibt es Verbindungen zwischen zwei Interaktionspunkten, die entgegengesetzte Rollen spielen (wobei diese sowohl interne Interaktionspunkte der jeweiligen Modulinstanz sein können als auch externe Interaktionspunkte von direkten Sohn-Modulinstanzen). Und zweitens gibt es Verbindungen zwischen zwei Interaktionspunkten, die die gleiche Rolle spielen (und die zu zwei Modulinstanzen gehören, von denen die eine ein Sohn der anderen ist). Die erste Verbindungsart wird durch die **`Connect`**-Operation erzeugt, die zweite durch die **`Attach`**-Operation.

In TLA/PT beschreiben wir diese beiden Relationen folglich durch die beiden TLA/PT-Variablen `Connect` und `Attach`, die beide Mengen von Paaren sein werden. Die einzelnen Paare enthalten dabei jeweils die Kennungen der verbundenen Interaktionspunkte. Beispiel:

```
(Ip_id.ip1, Ip_id.ip2) ∈ Connect
(Ip_id.ip2, Ip_id.ip1) ∈ Connect
(Ip_id.ip3, Ip_id.ip4) ∈ Attach
```

Sobald man alle Operationen definiert hat, die die Relation `Connect` verändern, kann man sehen und nachweisen, daß diese Relation symmetrisch und irreflexiv ist. (Wir können diesen Nachweis hier nicht formal führen, da wir in dieser Arbeit nicht alle Estelle-Operationen formal definieren werden. Diese Eigenschaft ist aber auch intuitiv einleuchtend.) Hierauf basierend werden wir daher die Relation `Connect` zukünftig als Menge von ungeordneten Paaren beziehungsweise als Menge von zweielementigen Mengen darstellen. Beispiel:

```
{Ip_id.ip1, Ip_id.ip2} ∈ Connect
```

Die Relation `Attach` wird sich ebenfalls als irreflexiv herausstellen, aber nicht als symmetrisch. Im Gegenteil, sie ist antisymmetrisch. Als Hilfe für die Definition der Estelle-Operationen ist die irreflexive, transitive Hülle der Relation `Attach` interessant. Wir nennen sie

```
TransitiveAttach
```

und definieren sie als die kleinste Menge, für die gilt:

```
Attach ⊆ TransitiveAttach
∧ (∀Ip_id.ip1, Ip_id.ip2, Ip_id.ip3::
      ( Ip_id.ip1 TransitiveAttach Ip_id.ip2
      ∧ Ip_id.ip2 TransitiveAttach Ip_id.ip3)
   ⇒ Ip_id.ip1 TransitiveAttach Ip_id.ip3)
```

Weiterhin interessieren uns Paare von Interaktionspunkten, die jeweils an den Enden einer Kette von `Attach`-, `Connect`- und wiederum `Attach`-Relationen stehen:

```
Link ≜
{{Ip_id.ip1, Ip_id.ip2} |
     ¬(∃Ip_id.ip0:: ( (Ip_id.ip1, Ip_id.ip0) ∈ Attach
                    ∨ (Ip_id.ip2, Ip_id.ip0) ∈ Attach))
   ∧ (∃Ip_id.ip3, Ip_id.ip4::
          ( Ip_id.ip1 = Ip_id.ip3
          ∨ (Ip_id.ip1, Ip_id.ip3) ∈ TransitiveAttach)
        ∧ ( Ip_id.ip2 = Ip_id.ip4
          ∨ (Ip_id.ip2, Ip_id.ip4) ∈ TransitiveAttach)
        ∧ {Ip_id.ip3, Ip_id.ip4} ∈ Connect ) }
```

Diese Verbindungen sind es, über die die Nachrichten transportiert werden; die **`Output`**- und die **`When`**-Operation werden von der Relation `Link` Gebrauch machen.

Die Relation `Link` ist, wie man leicht zeigen kann, symmetrisch, so daß die Notation wiederum als Menge von zweielementigen Mengen erfolgt.

Nach Definition aller Estelle-Operationen wird man außerdem zeigen können, daß die Relation `Link` irreflexiv ist, und insbesondere auch, daß sie sogar eine partielle Funktion ist, so daß wir die Notation

```
Link(Ip_id.ip1) = Ip_id.ip2
Link(Ip_id.ip2) = Ip_id.ip1
```

verwenden können, um zu einem Interaktionspunkt den Gegenpart zu erhalten, sofern wir wissen, daß dieser existiert.

Für Beweise über Estelle-Spezifikationen, in denen die Kommunikationsstruktur (nach der ersten Initialisierung) statisch ist, empfiehlt es sich, als einen der ersten Beweisschritte die Relation `Link` explizit als Aufzählung zu bestimmen und ihre Invarianz formal nachzuweisen. Dadurch vermeidet man den Umgang mit den obigen, aufwendigen Formeln. Denn die Definitionen aller Prädikatentransformatoren zur Nachrichtenübertragung werden sich auf die Relation `Link` beziehen.

Anderenfalls ist es nützlich, nach Möglichkeit wenigstens Abschnitte der Spezifikation zu identifizieren, für die die Kommunikationsstruktur statisch ist, und den Schritt auf sie anzuwenden.

Ist die Kommunikationsstruktur aber voll dynamisch und ändert sich im Laufe einer Ausführung ständig, so wird auch das Beweisen einer Aussage über die Zustellung einer Nachricht recht aufwendig. Aber dies ist nicht erstaunlich, da diese Arbeit den Aufwand wiederspiegelt, der zur Verifikation der Operationen notwendig ist, die die Kommunikationsstruktur verändern. Und tun sie dies in einer komplexen Weise, so wird auch der Beweis schwieriger.

Schließlich wollen wir noch ein Wort zu dem Anfangszustand der Kommunikationsstruktur verlieren, auch wenn man dies später vielleicht noch dem Ausführungsmo-

dell zuordnen sollte. Vor dem Schalten der ersten Initialisierungstransition existiert noch keinerlei Kommunikationsverbindung, es gilt:

```
  Attach  = {}
∧ Connect = {}
```

6.7 Beschreibung des Transitionsteils

Das Schalten von Estelle-Transitionen hat im allgemeinen Auswirkungen auf den Zustand des Systems (Estelle-Variablen, Warteschlangen, ...). Daher wäre es denkbar, den Vorgang des Schaltens in TLA/PT durch die Auswirkungen auf den Zustand zu beschreiben. Aber dies wäre keine gute Lösung. Einmal könnten verschiedene Estelle-Transitionen mit identischen Wirkungen nicht mehr unterschieden werden. Solche Transitionen wären zwar vermutlich ein Entwurfsfehler in der Spezifikation, aber aus konzeptuellen Gründen sollte man auch hierüber reden können. Am wichtigsten ist aber folgender Grund: Das Ausführungsmodell spricht vom Auswählen und Schalten bestimmter, einzelner Estelle-Transitionen, und dies müssen wir in TLA/PT ausdrücken können.

Damit stellt sich die Frage, wie wir über diese Vorgänge reden können, die nicht direkt Veränderungen des Zustands des bisher betrachteten Systems sind. Bei näherem Hinsehen stellt man fest, daß sich der Zustand des Systems sehr wohl ändert, wenn eine Estelle-Transition schaltet, auch wenn sich weder der Zustand von Estelle-Variablen noch von Warteschlangen verändert. Denn es verändert sich eine interne Verwaltungsinformation, auf die das Ausführungsmodell Bezug nimmt. Für jede Estelle-Transition jeder Modulinstanz kennt das Ausführungsmodell zwei wichtige Zustandsübergänge. Die Estelle-Transition kann „unwiderruflich zum Schalten ausgewählt werden“, und sie kann „schalten“. Die Bedingungen und Konsequenzen des Auswählens sind Teil des Ausführungsmodells, ebenso der Zeitpunkt des Schaltens.

Es ist eine grundlegende Eigenschaft des Ausführungsmodells, daß eine Estelle-Transition nur dann schalten kann, wenn sie vorher dazu ausgewählt worden ist, und daß das Auswählen und das Schalten immer streng abwechselnd stattfinden müssen.

Daher benötigen wir zur Beschreibung dieser Zustandsübergänge für jede Estelle-Transition zwei Zustände, „nicht ausgewählt“ und „ausgewählt“. Der Zustandsübergang von „nicht ausgewählt“ zu „ausgewählt“ drückt den Vorgang des Auswählens aus, der umgekehrte Zustandsübergang das Schalten.

Im ISO-Standard wird diese Zustandsinformation durch das Enthaltensein oder Nichtenthaltensein der betreffenden Estelle-Transition in einer Menge von ausgewählten Estelle-Transitionen beschrieben.

Wir könnten diesen Weg ebenfalls beschreiten, aber im Zusammenhang mit den TLA/PT-Aktionen erscheint es uns günstiger, nicht eine große (Mengen-)Variable zu definieren, sondern für jede Transition eine eigene boolsche Variable. So werden Zustandsübergänge leichter ausdrückbar. Sie bestehen einfach im „Umkippen“ der boolschen Variablen und nicht darin, daß sich die Anzahl der Elemente einer Menge ändert.

Der ISO-Standard ordnet die Zustandsinformation über die ausgewählten Estelle-Transitionen jeweils den einzelnen Modulinstanzen zu. Dies ist nicht zwingend, aber sehr naheliegend, und so werden wir es ebenfalls tun. Denn mit der Erzeugung einer Modulinstanz werden ebenso Instanzen der zugehörigen Estelle-Transitionen erzeugt, so daß es zu einer Estelle-Transitions-Definition mehrere Instanzen geben kann, die unterschieden werden müssen. Und weiterhin ist es sinnvoll, die Modulinstanzen, da sie erweiterte endliche Automaten darstellen, als aktive Komponenten anzusehen, womit natürlich diesen auch ihre zugehörige Verwaltungsinformation zugeordnet werden sollte. (Letztere Betrachtung ist allerdings rein pragmatisch, vom mathematischen Konzept her beschreibt eine Estelle-Spezifikation lediglich einen einzigen, großen abstrakten Automaten, der einen einzigen, strukturierten Gesamt-

zustand besitzt.)

Sei also α eine Modulinstanz und `trnam` der Name einer Estelle-Transition. (In Kapitel 2.2 hatten wir gefordert, daß jeder Estelle-Transition mit Hilfe des **`Name`**-Konstrukts ein Name zugeordnet wird, was eine rein syntaktische Maßnahme ohne semantische Konsequenzen ist.) Dann bezeichnen wir mit

α`.trnam.Selected`

die der Estelle-Transition zugeordnete boolsche TLA/PT-Variable.

Anmerkung: Wie man sieht, haben wir nicht einfach α`.trnam` gewählt, sondern die Variable noch weiter strukturiert. Dies hat zwei Gründe: Erstens wird so intuitiv die Bedeutung einer Formel klarer: Der Ausdruck $\neg\alpha$`.trnam.Selected` sagt deutlicher als $\neg\alpha$`.trnam`, daß die Estelle-Transition zur Zeit gerade nicht ausgewählt ist. Und zweitens kann eine Estelle-Transition zum Beispiel auch lokale Variablen besitzen, die als weitere Unterstruktur von α`.trnam` dargestellt werden können müssen.

Nachdem wir nun für jede Estelle-Transition eine eigene boolsche Variable haben, ist es für das Ausführungsmodell leicht, in TLA/PT das Auswählen und das Schalten mit Bedingungen und Wirkungen zu verknüpfen. Die Aktion des Auswählens drücken wir folgendermaßen aus:

$\neg\alpha$`.trnam.Selected` $\wedge$ α`.trnam.Selected`$'$

Und die Aktion des Schaltens:

α`.trnam.Selected` $\wedge$ $\neg\alpha$`.trnam.Selected`$'$

Bevor wir nun allerdings zur Verknüpfung mit den Bedingungen und Wirkungen kommen können, müssen wir etwas ausholen und diskutieren, wie wir die TLA/PT-Formel über die gesamten Eigenschaften eines Systems am günstigsten aufbauen. Um Beispielsysteme zu spezifizieren, verwendet Lamport in seinem TLA-Report meist Formeln von beispielsweise folgender Form:

$\Box[(A1 \wedge B1 \wedge C1) \vee A2 \vee A3 \vee (A4 \wedge B4)]_f$

`A1` bis `B4` sind hierbei Aktionen und `f` eine Zustandsfunktion. Lamport möchte damit ausdrücken, daß jeweils die konjunktiv verbundenen Aktionen nur zusammen stattfinden können und daß die disjunktiv verbundenen Gruppen Alternativen darstellen. Wie in Kapitel 3.3 beschrieben, läßt die Notation „$\Box[\ ]_f$“ immer auch die zusätzliche (Stotterschritt-)Alternative $f' = f$ zu.

Auch in unserer Formel sollen gewisse Teilaktionen stets zusammen stattfinden, und zwar alle diejenigen, die mit einer bestimmten Estelle-Transition verbunden sind. Ebenso gibt es Alternativen, die nur wahlweise stattfinden sollen, zum Beispiel die Auswahl zwischen verschiedenen Estelle-Transitionen. Insofern ist diese Grundstruktur auch für unsere Arbeit durchaus geeignet. Allerdings schafft die obige Form ein Problem. Die Teilformeln, die sich auf eine Estelle-Transition beziehen, möchten wir gern unabhängig voneinander aufschreiben, denn wir wollen einen möglichst kompositionellen Weg beschreiten und die Gesamtsystembeschreibung aus vielen Teileigenschaften zusammensetzen.

Einerseits sollen die Eigenschaften einer Estelle-Transition, die sich aus dem Ausführungsmodell ergeben, in einem Teil der Formel für alle Estelle-Transitionen gemeinsam aufgeschrieben werden. Andererseits sollen in einem separaten anderen Teil der Formel diejenigen Eigenschaften aufgeschrieben werden, die sich direkt aus dem Spezifikationstext ergeben. Dies soll mit einer möglichst ähnlichen Struktur wie in dem Spezifikationstext geschehen, also noch weiter in Teilformeln untergliedert. Die Verbindung zwischen allen diesen Teilen sollen die boolschen Transitionsvariablen herstellen.

Daher formen wir die obige Formel etwas um:

```
  □[(A1 ∧ B1 ∧ C1) ∨ A2 ∨ A3 ∨ (A4 ∧ B4)]_f
≡ { Def. Notation „□[ ]_f“, Aussagenlogik, Schlußregel E1 }
  (∃v1:: □( (¬(v1 ∧ ¬v1′) ∨ (A1 ∧ B1 ∧ C1))
          ∧ ((v1 ∧ ¬v1′) ∨ A2 ∨ A3 ∨ (A4 ∧ B4) ∨ (f′ = f))))
≡ { Aussagenlogik }
  (∃v1:: □( (¬(v1 ∧ ¬v1′∧ v1≠v1′) ∨ (A1 ∧ B1 ∧ C1))
          ∧ (A2 ∨ A3 ∨ (A4 ∧ B4) ∨ (v1 ∧ ¬v1′) ∨ (f′ = f))))
≡ { Aussagenlogik }
  (∃v1:: □( (((v1 ∧ ¬v1′) ⇒ A1) ∨ v1=v1′)
          ∧ (((v1 ∧ ¬v1′) ⇒ B1) ∨ v1=v1′)
          ∧ (((v1 ∧ ¬v1′) ⇒ C1) ∨ v1=v1′)
          ∧ (A2 ∨ A3 ∨ (A4 ∧ B4) ∨ (v1 ∧ ¬v1′) ∨ (f′ = f))))
≡ { Dasselbe wie für v1 auch für v2, v3 und v4 }
  (∃v1:: (∃v2:: (∃v3:: (∃v4::
    □( (((v1 ∧ ¬v1′) ⇒ A1) ∨ v1=v1′)
     ∧ (((v1 ∧ ¬v1′) ⇒ B1) ∨ v1=v1′)
     ∧ (((v1 ∧ ¬v1′) ⇒ C1) ∨ v1=v1′)
     ∧ (((v2 ∧ ¬v2′) ⇒ A2) ∨ v2=v2′)
     ∧ (((v3 ∧ ¬v3′) ⇒ A3) ∨ v3=v3′)
     ∧ (((v4 ∧ ¬v4′) ⇒ A4) ∨ v4=v4′)
     ∧ (((v4 ∧ ¬v4′) ⇒ B4) ∨ v4=v4′)
     ∧ ( (v1 ∧ ¬v1′) ∨ (v2 ∧ ¬v2′) ∨ (v3 ∧ ¬v3′) ∨ (v4 ∧ ¬v4′)
       ∨ (f′ = f)))))))
≡ { Schlußregel STL5, Def. Notation „□[ ]_g“ }
  (∃v1:: (∃v2:: (∃v3:: (∃v4::
      □[(v1 ∧ ¬v1′) ⇒ A1]_v1
    ∧ □[(v1 ∧ ¬v1′) ⇒ B1]_v1
    ∧ □[(v1 ∧ ¬v1′) ⇒ C1]_v1
    ∧ □[(v2 ∧ ¬v2′) ⇒ A2]_v2
    ∧ □[(v3 ∧ ¬v3′) ⇒ A3]_v3
    ∧ □[(v4 ∧ ¬v4′) ⇒ A4]_v4
    ∧ □[(v4 ∧ ¬v4′) ⇒ B4]_v4
    ∧ □[(v1 ∧ ¬v1′) ∨ (v2 ∧ ¬v2′) ∨ (v3 ∧ ¬v3′) ∨ (v4 ∧ ¬v4′)]_f))))
```

Durch diese Äquivalenzumformungen haben wir nunmehr eine völlig konjunktive Form erreicht, wie wir sie schon gegen Ende von Kapitel 6.1 angekündigt hatten.

Als nächstes werden wir diese Form so verändern, daß die Aktionen in Beziehung mit den boolschen Transitionsvariablen gesetzt werden.

Nehmen wir an, daß

- die Aktionen `A1`, `B1` und `C1` die Bedingungen und Wirkungen des Schaltens einer Estelle-Transition α`.trnam1` beschreiben,
- die Aktion `A2` die Bedingungen und Wirkungen des Auswählens von α`.trnam1` beschreibt,
- die Aktion `A3` die Bedingungen und Wirkungen des Schaltens einer Estelle-Transition α`.trnam2` beschreibt und
- die Aktionen `A4` und `B4` die Bedingungen und Wirkungen des Auswählens von α`.trnam2` beschreiben.

(Dies ist nur ein Demonstrationsbeispiel. Für eine sinnvolle, vollständige Systembeschreibung wären selbstverständlich sehr viel mehr Aktionen notwendig.)

Um die eben aufgezählten Eigenschaften darzustellen, verändern wir die obige Formel ein wenig, indem wir die Existenzquantoren fortlassen und ihre Variablen `v1`

bis v4 in der folgenden Weise mit den Transitionsvariablen identifizieren:

$$\begin{aligned}
\texttt{v1} &\triangleq \alpha.\texttt{trnam1.Selected},\\
\texttt{v2} &\triangleq \neg\alpha.\texttt{trnam1.Selected},\\
\texttt{v3} &\triangleq \alpha.\texttt{trnam2.Selected},\\
\texttt{v4} &\triangleq \neg\alpha.\texttt{trnam2.Selected}
\end{aligned}$$

Damit ergibt sich (*nicht* durch eine Äquivalenzumformung) die folgende, endgültige Grundform für unsere Formeln zur Beschreibung von Systemen, die in Estelle spezifiziert sind:

$$\begin{aligned}
&\Box[(\alpha.\texttt{trnam1.Selected} \wedge \neg\alpha.\texttt{trnam1.Selected}') \Rightarrow \texttt{A1}]_{\alpha.\texttt{trnam1.Selected}}\\
\wedge\ &\Box[(\alpha.\texttt{trnam1.Selected} \wedge \neg\alpha.\texttt{trnam1.Selected}') \Rightarrow \texttt{B1}]_{\alpha.\texttt{trnam1.Selected}}\\
\wedge\ &\Box[(\alpha.\texttt{trnam1.Selected} \wedge \neg\alpha.\texttt{trnam1.Selected}') \Rightarrow \texttt{C1}]_{\alpha.\texttt{trnam1.Selected}}\\
\wedge\ &\Box[(\neg\alpha.\texttt{trnam1.Selected} \wedge \alpha.\texttt{trnam1.Selected}') \Rightarrow \texttt{A2}]_{\alpha.\texttt{trnam1.Selected}}\\
\wedge\ &\Box[(\alpha.\texttt{trnam2.Selected} \wedge \neg\alpha.\texttt{trnam2.Selected}') \Rightarrow \texttt{A3}]_{\alpha.\texttt{trnam2.Selected}}\\
\wedge\ &\Box[(\neg\alpha.\texttt{trnam2.Selected} \wedge \alpha.\texttt{trnam2.Selected}') \Rightarrow \texttt{A4}]_{\alpha.\texttt{trnam2.Selected}}\\
\wedge\ &\Box[(\neg\alpha.\texttt{trnam2.Selected} \wedge \alpha.\texttt{trnam2.Selected}') \Rightarrow \texttt{B4}]_{\alpha.\texttt{trnam2.Selected}}\\
\wedge\ &\Box[\ \ (\alpha.\texttt{trnam1.Selected} \neq \alpha.\texttt{trnam1.Selected}')\\
&\quad\ \vee (\alpha.\texttt{trnam2.Selected} \neq \alpha.\texttt{trnam2.Selected}')]_{\texttt{f}}
\end{aligned}$$

Hiermit haben wir nicht nur eine konjunktive Darstellungsform unserer Formel erreicht, bei der sich die Formel durch Aneinanderreihen von Konjunktoren stückweise zusammensetzen und umgekehrt auch wieder analysieren läßt, sondern wir haben auch die Verbindung der einzelnen Teilformeln über die boolschen Transitionsvariablen erreicht. Wir können den Vorgang des Schaltens einer Estelle-Transition auf natürliche Weise ausdrücken und seine Darstellung sowohl in der Spezifikation des Ausführungsmodells als auch später in Beweisen über Estelle-Spezifikationen verwenden.[33] (Praktisch jeder Beweis wird sich auf das Ausführungsmodell beziehen müssen.)

Kommen wir nun zu den weiteren Eigenschaften, die eine Estelle-Transition besitzen kann. Mit einer **From-** und einer **To-**Klausel kann ein Übergang des Hauptzustandes spezifiziert werden, mit **Provided** und **When** können Bedingungen für die Auswahl der Estelle-Transition angegeben werden, mit dem **Priority-**Konstrukt können Prioritäten vor anderen Estelle-Transitionen gesetzt werden, mit **Delay** können Verzögerungszeiten spezifiziert werden, mit **Any** können mehrere Estelle-Transitionen zusammengefaßt werden, und schließlich kann ein Transitionsrumpf eine Aktion spezifizieren.

Wie dieses alles auf die Entwicklung des Zustandes wirkt, ist Sache des Ausführungsmodells und wird in Kapitel 7 beschrieben. An dieser Stelle müssen wir allerdings erörtern, wie die Übersetzungsfunktion diese Klauseln in TLA/PT übersetzen soll, so daß das in TLA/PT formulierte Ausführungsmodell Zugriff darauf hat.

Für die **Provided-**Klausel ist dies einfach: Sie enthält ein Prädikat, das direkt in TLA/PT übersetzt werden kann, wir müssen nur einen Namen vergeben:

$$\Box(\ \ldots \wedge \alpha.\texttt{trnam.Provided = boolterm})$$

(In diesem Beispiel für eine Modulinstanz α, einen Transitionsnamen `trnam` und ein Prädikat `boolterm`.)

Auch für die **From-**, **To-**, **Priority-** und **Delay-**Klauseln wird entsprechend erzeugt:

[33] Wie man bei einer Verifikation eine Formel, die Transitionsvariablen enthält, in Beziehung setzt zu einer abstrakteren Eigenschaftsspezifikation, die diese Variablen nicht enthält, ist in Kapitel 3.3 bei der Behandlung der Verfeinerungs-Abbildungen beschrieben.

```
□( ... ∧ α.trnam.From = ...)
□( ... ∧ α.trnam.To = ...)
□( ... ∧ α.trnam.Priority = ...)
□( ... ∧ α.trnam.Delay1 = ...)
□( ... ∧ α.trnam.Delay2 = ...)
```

(Als zweiter Parameter α.`trnam.Delay2` der **`Delay`**-Klausel ist auch der Text „*“ erlaubt, der eine unendliche Verzögerungszeit anzeigt. Er soll in den speziellen Wert `Infinity` übersetzt werden, auf dessen Eigenschaften wir in Kapitel 7.2 zurückkommen werden.)

Die Übersetzungsfunktion muß also einfach nur die zu den Klauseln gehörigen Terme in TLA/PT umsetzen, ohne sie weiter zu interpretieren. Um die Auswirkung auf den Zustand wird sich das Ausführungsmodell kümmern.

Die **`Any`**-Klausel ist im Gegensatz zu den anderen nur eine syntaktische Abkürzung. Estelle-Transitionen mit **`Any`**-Klausel können mit einem einfachen Algorithmus in mehrere einfache Estelle-Transitionen expandiert werden ([AmPrSc88]). Ebenso wie der ISO-Standard werden wir daher voraussetzen, daß alle derartigen Estelle-Transitionen bereits vor Anwendung der Übersetzungsfunktion expandiert worden sind.

Für den Transitionsrumpf muß die Übersetzungsfunktion einen Prädikatentransformator spezifizieren, der die zugehörige Operation beschreibt. Damit das Ausführungsmodell sich ohne Umstände auf diesen Prädikatentransformator beziehen kann, lassen wir die Übersetzungsfunktion eine *Definition* erzeugen:

```
□( ... ∧ α.trnam.Body_Effect_PT ≜ ...)
```

In Kapitel 5.4 haben wir dieses Konzept informal zur TLA/PT hinzugenommen, da uns eine Möglichkeit zur Strukturierung unserer Formeln für die Lesbarkeit notwendig erscheint. Es ist auch nur ein rein syntaktisches Konzept, alle Definitionen könnten aus einer Formel wieder eliminiert werden, was aber ihre Lesbarkeit und Verständlichkeit verschlechtern würde.

So müßten wir eigentlich auch die „Definition“ für die TLA/PT definieren. Aber da auch die entsprechende Erweiterung der TLA zur TLA$^+$ zum Zeitpunkt unserer Arbeit noch nicht formal ausgearbeitet ist ([Lam91b]), und da die Schlußregeln der TLA/PT bereits nicht ganz ausgearbeitet sind (Kapitel 5.3), unterlassen wir dies und bleiben an diesem Punkt ein wenig informell. Große Probleme sollten sich nicht ergeben, da [Lam91b] sagt, daß ein Algorithmus existiert, auf dessen Basis die Erweiterung definiert werden kann.

Nun bleibt lediglich noch die **`When`**-Klausel zu betrachten, sie bedarf etwas mehr Überlegung als die anderen Klauseln. Diese Klausel spielt eine Doppelrolle. Einerseits spezifiziert sie eine Bedingung für das Schalten, indem sie fordert, daß eine bestimmte Warteschlange eine bestimmte Nachricht an erster Stelle enthalten muß. Andererseits spezifiziert sie auch die Entnahme dieser Nachricht für den Fall, daß die Estelle-Transition schaltet.

Daher muß die Übersetzungsfunktion einerseits eine Gleichung

```
□( ... ∧ α.trnam.When = ...)
```

erzeugen, die die richtige Bedingung für die Warteschlange ausdrückt (siehe zu Warteschlangen auch Kapitel 6.5), andererseits aber auch eine Definition für einen Prädikatentransformator

```
□( ... ∧ α.trnam.When_Effect_PT ≜ ...)
```

Es kann bei der **`When`**-Klausel sogar der Fall sein, daß Parameter von Nachrichten

übergeben werden. In diesem Falle werden entsprechende lokale Variablen für diese Estelle-Transition definiert, dies soll auf die gleiche Weise wie für andere Variablen auch geschehen (siehe Kapitel 6.3).

Um die Formulierung des Ausführungsmodells übersichtlicher zu gestalten, fassen wir alle Wirkungen einer Estelle-Transition, also die des Transitionsrumpfes, die der **When**-Klausel und die der **To**-Klausel, in einem einzigen Prädikatentransformator „α.trnam.Effect_PT“ zusammen. Es soll daher für jede Transition die Definition erzeugt werden:

```
(α.trnam.Effect_PT).X ≜
  (α.trnam.Body_Effect_PT).(α.trnam.When_Effect_PT).
    ( (α.trnam.To = α.Same ⇒ X)
    ∧ (α.trnam.To ≠ α.Same ⇒ (α.State:=α.trnam.To).X))
                                                  für alle X
```

Wie man sieht, spezifiziert auch die **To**-Klausel eine Aktion, und zwar eine Veränderung der Variablen α.State. Die Begründung hierfür folgt in Kapitel 7.1 über das Ausführungsmodell.

Im übrigen ist zu der letzten Definition anzumerken, daß die Reihenfolge der Verkettung der drei Wirkungsanteile willkürlich gewählt wurde, da sie sich jeweils auf unterschiedliche Variablen beziehen und daher kommutativ sind. (Nach einem entsprechenden Beweis wäre eine Umformung, die die Reihenfolge vertauscht, zulässig.)

Um die Sache etwas anschaulicher zu machen, geben wir ein kurzes Beispiel, wie der (um irrelevante Teile vereinfachte) Prädikatentransformator für einen einfachen Transitionsrumpf aussieht:

```
Trans
Name incr:
  To done
    Begin
      x := x+1;
      y := y+1
    End
```

```
(α.incr.Effect_PT).X ≜
  (α.x:=α.x+1).(α.y:=α.y+1).(α.State:=α.done).X      für alle X
```

Schließlich ist noch zu bedenken, daß nicht für jede Estelle-Definition alle genannten Klauseln syntaktisch im Spezifikationstext vorkommen müssen. Mit dieser Abwesenheit wird natürlich auch ein bestimmtes Verhalten spezifiziert, zum Beispiel die niedrigste Priorität von allen oder die Schaltbedingung, die immer wahr ist. Dies muß das Ausführungsmodell auch beachten, und daher muß die Übersetzungsfunktion in diesen Fällen Ersatzformeln einsetzen, zum Beispiel:

```
□( ... ∧ α.trnam.Provided = True)
```

Für das Ausführungsmodell muß die Übersetzungsfunktion noch eine weitere Information bereitstellen: Das Ausführungsmodell muß zu jeder Modulinstanz die Menge ihrer Estelle-Transitionen kennen. Damit kann es diese dann zum Beispiel auf Schaltbereitschaft untersuchen und entsprechende Aktionen spezifizieren.

Hierfür benötigen wir einen Aufzählungstyp (siehe dazu Kapitel 6.3), der uns genügend viele verschiedene Identifikatoren bereitstellt. Daher muß uns die Übersetzungsfunktion für jede Moduldefinition eine Definition eines Aufzählungstyps

```
α.tr1.Id, ..., α.trn.Id
```

liefern. (Diese Definition legt insbesondere fest, daß sich die Identifikatoren unterscheiden. Siehe auch Kapitel 6.3.)

Und dann muß die Übersetzungsfunktion für jede Moduldefinition die Menge aller Estelle-Transitionsdefinitionen beschreiben:

$$\Box(\ \ldots \wedge \alpha.\texttt{LocTrans} = \{\alpha.\texttt{tr1.Id}, \ldots, \alpha.\texttt{trn.Id}\})$$

Außerdem muß sie die Menge aller Estelle-Transitionsdefinitionen von allen Sohn-Modulinstanzen (und allen weiteren Nachkommen) beschreiben:

$$\begin{aligned}\Box(\ \ldots \wedge \alpha.\texttt{Trans} = \alpha.\texttt{LocTrans}\\ \cup\ (\textstyle\bigcup\beta\colon\ \beta.\texttt{Typ} \in \alpha.\texttt{LocModDefs} \wedge \beta.\texttt{Exists}:\\ \beta.\texttt{Trans})))\end{aligned}$$

(Zur Definition der Menge aller Sohn-Moduldefinitionen α.`LocModDefs` siehe Kapitel 6.4)

Auch von einer Modulinstanz zu einer anderen Modulinstanz müssen sich die Identifikatoren von Estelle-Transitionen unterscheiden lassen (obwohl bei einer Moduldefinition erst einmal nicht klar ist, wieviele Modulinstanzen es dazu geben wird). Dies können wir zum Beispiel dadurch spezifizieren, daß die Übersetzungsfunktion jede obige Gleichung für α.`Trans` durch die Forderung ergänzt, daß bei der Mengenvereinigung die Zahl der Elemente der Vereinigungsmenge gleich der Summe der Elementzahlen der vereinigten Mengen ist:

$$\begin{aligned}\Box(\ \ldots \wedge |\alpha.\texttt{Trans}| = |\alpha.\texttt{LocTrans}|\\ +\ (\textbf{Summe}\ \beta\colon\ \beta.\texttt{Typ} \in \alpha.\texttt{LocModDefs} \wedge \beta.\texttt{Exists}:\\ |\beta.\texttt{Trans}|)))\end{aligned}$$

Damit sind die Identifikatoren α.`trnam.Id` für verschiedene Sohn-Modulinstanzen verschieden, und durch den hierarchischen Aufbau der Modulinstanzen ergibt sich dann das Gewünschte.

Mit dieser Unterstützung durch die Übersetzungsfunktion können wir nun zum Beispiel in der Spezifikation des Ausführungsmodells die folgende Quantifikation schreiben:

$$\Box(\ \ldots \wedge (\forall \texttt{tr}\colon\ \texttt{tr.Id} \in \alpha.\texttt{LocTrans}\colon\ \ldots))$$

Und wir beschreiben damit genau die Transitionen `tr`, die zur aktuellen Modulinstanz α gehören.

Ein weiteres Thema, das mit den gewöhnlichen Estelle-Transitionen eng verwandt ist, wollen wir in diesem Kapitel ebenfalls noch erörtern. Dies sind die **`Initialize`**-Transitionen.

`Initialize`-Transitionen sind spezielle Estelle-Transitionen. Wenn eine Modulinstanz mit einer **`Init`**-Operation erzeugt wird, ist nach dieser Aktion diese Modulinstanz in einem Zustand, der aus dem Schalten einer dieser speziellen Estelle-Transition resultiert (sofern diese Modulinstanz überhaupt eine **`Initialize`**-Transition besitzt). Es wird, grob gesagt, in einer einzigen Aktion sowohl die Modulinstanz erzeugt, als auch die **`Initialize`**-Transition geschaltet. Die Schalteffekte dieser Estelle-Transition sind also Teil der Schalteffekte der Estelle-Transition, die die **`Init`**-Operation enthält.

Daher fordern wir, daß die Übersetzungsfunktion aus einer **`Initialize`**-Transition die gleichen Definitionen erzeugt wie aus allen anderen Estelle-Transitionen[34], insbesondere natürlich den Prädikatentransformator über die Schaltwirkungen. Diese Definitionen werden allerdings im wesentlichen von der Definition der **`Init`**-Opera-

tion „verwendet“ werden.

Die **Init**-Operation muß allerdings „wissen“, welche **Initialize**-Transitionen es zu einer Modulinstanz gibt. Daher muß die Übersetzungsfunktion, ganz ähnlich wie für die gewöhnlichen Estelle-Transitionen, eine entsprechende Menge von Identifikatoren definieren, und die Elemente dieser Menge müssen verschieden von den Elementen der Menge der gewöhnlichen Transitionen sein:

$$\begin{aligned}\Box(\ \ldots\ &\wedge\ \alpha.\texttt{IniTrans} = \{\alpha.\texttt{itrnam1.Id},\ \ldots,\ \alpha.\texttt{itrnamn.Id}\}\\ &\wedge\ |\alpha.\texttt{IniTrans} \cup \alpha.\texttt{LocTrans}|\\ &\quad = |\alpha.\texttt{IniTrans}| + |\alpha.\texttt{LocTrans}|)\end{aligned}$$

Alles weitere und die eigentliche Semantik von Estelle-Transitionen werden wir in Kapitel 7 über das Ausführungsmodell beschreiben.

[34] Abgesehen von den **Delay**-, **When**-, **From**- und **Provided**-Klauseln, die hier syntaktisch nicht zulässig sind.

7. Definition des Ausführungsmodells

Nachdem wir in Kapitel 6 angegeben haben, wie die Übersetzungsfunktion den Text einer Estelle-Spezifikation in der TLA/PT darstellen soll, können wir nun das Ausführungsmodell dazu angeben, das die TLA/PT-Spezifikation vervollständigen soll.

Estelle beruht auf einem einfachen Grundmodell, aber es wurden eine ganze Reihe von Konzepten hinzugefügt, so daß das vollständige Ausführungsmodell recht umfangreich ist. Daher werden wir unsere TLA/PT-Spezifikation des Ausführungsmodells schrittweise einführen, wobei wir mit den einfacheren Eigenschaften beginnen.

7.1 Das Ausführungsmodell ohne quantitative Zeitaspekte

In Kapitel 6.7 haben wir eine Estelle-Transition durch eine Verbundvariable dargestellt. Sie war selbst wiederum Komponente ihrer Modulinstanz und trug als Komponentennamen den Namen, der ihr mit dem **Name**-Konstrukt gegeben worden sein mußte. Alle Identifikatoren von Transitionsvariablen waren jeweils Elemente der speziellen Menge α.`LocTrans`. (Mit α haben wir die jeweilige Modulinstanz bezeichnet.)

Mit Hilfe der Komponenten `Selected` einer solchen Variablen `tr` haben wir beschrieben[35], wie sie endgültig zum Schalten ausgewählt wird, und wie sie schaltet.

Mit der Aktion

$\neg$`tr.Selected` $\wedge$ `tr.Selected`$'$

haben wir ausgedrückt, daß die Estelle-Transition von der Modulinstanz endgültig ausgewählt wird. Das Schalten ist damit unvermeidlich, aber die Schaltwirkung ist in diesem Zustand noch nicht eingetreten.

Einem Implementator steht es frei, von diesem Zustand ab mit der Berechnung des Ergebnisses zu beginnen, solange die Wirkung noch nicht nach außen sichtbar wird. Es läßt sich zeigen, daß durch die Vorrangregeln von Estelle, zu denen wir noch kommen werden, sichergestellt wird, daß sich der für die Estelle-Transition sichtbare Teil des Zustandes nicht mehr ändert, und er daher in die Berechnungen bereits einbezogen werden kann.

Die verschiedenen in Estelle formulierten Bedingungen (**Provided**, **When**, ...), unter denen eine Estelle-Transition schalten kann, sind in Kapitel 6.7 für jede Estelle-Transition in Prädikate übersetzt worden. Daher können wir sie jetzt einfach als Bedingung zur endgültigen Auswahl verwenden. Am einfachsten ist es für die **Provided**- und die **When**-Klausel:

```
(∀tr: (tr.Id ∈ α.LocTrans):
  (¬tr.Selected ∧ tr.Selected′)  ⇒  (tr.Provided ∧ tr.When) )
```

(Die Menge der Identifikatoren der direkt zu einer Modulinstanz gehörenden Estelle-Transitionen α.`LocTrans` haben wir in Kapitel 6.7 definiert.)

[35] Wir werden im folgenden die Variable `tr` als Platzhalter für eine Transitionsvariable, zum Beispiel α.`trnam`, verwenden, und dann auch entsprechend über `tr` quantifizieren.

Die Aktion des Auswählens muß also das Bedingungsprädikat implizieren. Wenn das Prädikat nicht erfüllt ist, ist die Aktion folglich nicht zulässig.

Anmerkung: Den obigen Ausdruck haben wir mit dem Implikationszeichen „$\Rightarrow$“ notiert, so wie wir es in Kapitel 6.7 angekündigt haben, wobei allerdings das Prädikat ein Spezialfall einer Aktion ist.

Die **From**-Klausel fragt den sogenannten Hauptzustand einer Modulinstanz ab. Dies ist ein besonders wichtiger Teil des Zustandes der Modulinstanz, er wird nicht wie die übrigen Variablen mit dem **Var**-Konstrukt (oder **Export**-Konstrukt) deklariert, sondern mit dem **State**-Konstrukt. Wie wir bereits am Ende von Kapitel 6.1 angemerkt haben, stellen wir alle Teile des Systemzustandes durch TLA/PT-Variablen dar, und entsprechend stellen wir auch den Hauptzustand einer Modulinstanz α als spezielle Variable α.State dar. Die **From**-Klausel muß von der Übersetzungsfunktion in eine Menge von Hauptzuständen übersetzt werden, so daß das Ausführungsmodell als Schaltbedingung fordern kann:

```
(∀tr: (tr.Id ∈ α.LocTrans):
  (¬tr.Selected ∧ tr.Selected′)  ⇒  α.State ∈ tr.From )
```

Für die weiteren Zwecke beschreiben wir die bisherigen Auswahlbedingungen zusammengefaßt durch das Prädikat tr.Enab:

```
(∀tr: tr.Id ∈ α.LocTrans:
   tr.Enab = tr.Provided ∧ tr.When ∧ α.State ∈ tr.From)
```

Auch die Prioritäten der Estelle-Transitionen müssen beachtet werden. Eine Estelle-Transition tr darf nur dann zum Schalten ausgewählt werden, solange keine andere Estelle-Transition tr_2 dazu bereit ist, die eine höhere[36] Priorität besitzt:

```
(∀tr: (tr.Id ∈ α.LocTrans):
    (¬tr.Selected ∧ tr.Selected′)
  ⇒ (∀tr₂: tr₂.Id ∈ α.LocTrans ∧ tr₂.Priority < tr.Priority:
      ¬tr₂.Enab))
```

Von den möglichen Klauseln bleibt jetzt nur noch die **Delay**-Klausel zu behandeln. Da sie quantitative Zeitaspekte ins Spiel bringt, werden wir auf sie aus Gründen der Übersichtlichkeit erst später in Kapitel 7.2 zurückkommen.

Dafür fassen wir unsere bisherigen Schaltbedingungen noch einmal zusammen.

```
  (∀tr: tr.Id ∈ α.LocTrans:
    tr.Enab = tr.Provided ∧ tr.When ∧ α.State ∈ tr.From)
∧ (∀tr: tr.Id ∈ α.LocTrans:
     (¬tr.Selected ∧ tr.Selected′)
   ⇒ ( tr.Enab
     ∧ (∀tr₂: tr₂.Id ∈ α.LocTrans ∧ tr₂.Priority < tr.Priority:
         ¬tr₂.Enab)))
```

Als nächstes betrachten wir das Schalten einer Estelle-Transition. Wir beschreiben es (siehe auch Kapitel 6.7) durch:

```
tr.Selected ∧ ¬tr.Selected′
```

Die Wirkung tritt atomar ein. Diese Atomizität ist insbesondere für die Warteschlangen der Nachrichtenkanäle wichtig. Alle Wirkungen auf Warteschlangen müssen in einem Schritt eintreten.

Welche Wirkungen dies jeweils sind, wird jeweils von dem Prädikatentransformator

[36] Höhere Prioritäten werden in Estelle durch kleinere Zahlen beschrieben.

`tr.Effect_PT` beschrieben. Dieser wird für jede Estelle-Transition von der Übersetzungsfunktion definiert (Kapitel 6.7).

Um aus einem Prädikatentransformator `PT` eine TLA/PT-Aktion zu machen, müssen wir ihn auf eine Zustandsfunktion `f` beziehen: „$\mathtt{PT}_{\mathtt{f}}$“ (Kapitel 5.1). Diese Zustandsfunktion `f` beschreibt den Teil des Zustandes, über den die TLA/PT-Aktion eine Aussage macht. (Dies ist notwendig wegen der Forderung, daß alle betroffenen Variablen auch explizit genannt werden sollen). Und um aus den TLA/PT-Aktionen eine TLA/PT-Formel zu machen, müssen wir außerdem angeben, welche Zustandsfunktion `g` unverändert bleiben soll, falls die Aktionen nicht stattfinden, sondern an ihrer Stelle ein „Stotterschritt“: $\Box[A \wedge B \wedge \ldots]_{\mathtt{g}}$. Diese Zustandsfunktionen `f` und `g` müssen wir nun bestimmen.

In Kapitel 4.1 haben wir uns entschieden und dies begründet, die Semantik von Estelle als „Interleaving“-Semantik darzustellen. Bei ihr schalten jeweils zwei Estelle-Transitionen niemals „gleichzeitig“ in einem einzigen Zustandsübergang, sondern dies geschieht immer in zwei verschiedenen Übergängen (bei der Reihenfolge kann allerdings eine nichtdeterministische Wahl möglich sein).

Somit ist offensichtlich, daß sich der Prädikatentransformator auf den gesamten Systemzustand beziehen kann, der einfacheren Beschreibung halber wird er dies auch tun. Ebenso bleibt bei einem Stotterschritt des gesamten Systems dieselbe Zustandsfunktion über das Gesamtsystem unverändert.

Den Zustand der obersten Modulinstanz, die alle weiteren Modulinstanzen enthält, haben wir in Kapitel 6.4 durch die Verbundvariable `Sys` beschrieben. Sie muß damit in dieser Zustandsfunktion enthalten sein. Die weiteren Teile des Systems, die ebenfalls in der Zustandsfunktion enthalten sein müssen, sind die einzelnen Teile die Kommunikationsstruktur, beschrieben durch die Relationen `Attach` und `Connect` (siehe Kapitel 6.6). Damit ergibt sich als Zustandsfunktion das Tripel:

$$\mathtt{(Sys,\ Attach,\ Connect)}$$

Und die unter anderem durch den Prädikatentransformator `PT` beschriebene Gesamtformel würde lauten:

$$\Box[(\ldots \Rightarrow \mathtt{PT}_{\mathtt{(Sys,Attach,Connect)}}) \wedge \ldots]_{\mathtt{g}}$$

Um dies etwas kürzer notieren zu können, definieren wir die Zustandsfunktion:

$$\mathtt{All} \triangleq \mathtt{(Sys,\ Attach,\ Connect)}$$

Damit kann man dann schreiben:

$$\Box[(\ldots \Rightarrow \mathtt{PT}_{\mathtt{All}}) \wedge \ldots]_{\mathtt{g}}$$

Die ersten drei Auslassungspunkte stehen für die Verbindung zum Schalten der Estelle-Transition. Wie in Kapitel 6.7 ausführlich begründet, spezifizieren wir sie, und zwar gleich für alle Estelle-Transitionen:

$$\begin{array}{l}
\Box((\forall \mathtt{tr}:\ (\mathtt{tr.Id} \in \alpha.\mathtt{LocTrans}): \\
\quad [\ \ (\mathtt{tr.Selected} \wedge \neg \mathtt{tr.Selected}') \\
\quad\ \ \Rightarrow ((\mathtt{tr.Effect_PT}).(\mathtt{Change}.\{\mathtt{tr.Selected}\}))_{\mathtt{All}}\ ]_{\mathtt{tr.Selected}})) \\
\wedge\ \ldots
\end{array}$$

Wir haben dabei den Prädikatentransformator `(tr.Effect_PT)` mit dem Prädikatentransformator `(Change.{tr.Selected})` zu dem Prädikatentransformator in der dritten Zeile verkettet, um zu spezifizieren, daß die Variable `tr.Selected` nach der Aktion einen beliebigen Wert annehmen darf, insbesondere, daß sie nicht den alten Wert behalten muß. Zusammen mit der Zeile davor haben wir damit spezifiziert, daß diese boolsche Variable beim Schalten von `True` nach `False` wechselt,

und dies völlig unabhängig davon, was der Prädikatentransformator `tr.Effect_PT` aussagt. Denn üblicherweise sagt er aus, daß alle Variablen bis auf bestimmte ihren alten Wert behalten müssen, was sonst auch für die Variable `tr.Selected` gelten würde.

Nun bleibt noch die oben erwähnte Zustandsfunktion `g` zu behandeln, die aussagt, was alles bei einem Stotterschritt unverändert bleiben muß. Wie ebenfalls bereits erwähnt, muß bei einem Stotterschritt der gesamte Systemzustand, also `All`, unverändert bleiben, so daß sich nur noch die Frage stellt, wo diese Zustandsfunktion im Gesamt-Ausführungsmodell erwähnt wird. In der letzten Formel haben wir entsprechend unserer Diskussion in Kapitel 6.7 nur `tr.Selected` an die eckigen Klammern geschrieben, wir müssen daher noch angeben, was sich nicht verändern soll. Dies tun wir ebenfalls in der in Kapitel 6.7 diskutierten Form. Dort hatten wir als letztes Konjunktionsglied zur Systemgesamtbeschreibung die folgende Teilformel hinzugefügt, die eine Disjunktion über alle dortigen Transitionsvariablen enthält:

```
...
∧ □[  (α.trnam1.Selected ≠ α.trnam1.Selected′)
    ∨ (α.trnam2.Selected ≠ α.trnam2.Selected′)]f
```

Daher fügen wir hier zum Gesamt-Ausführungsmodell ganz analog die folgende Teilformel hinzu, bei der eine Existenzquantifikation für die Disjunktion über alle Transitionsvariablen sorgt:

```
...
∧ □[(∃tr: tr.Id ∈ Sys.Trans:
      tr.Selected ≠ tr.Selected′)]All
```

Nachdem geklärt ist, welche Aktionen mit dem Schalten einer Estelle-Transition verbunden sein sollen und was geschehen soll, wenn die Transitionsvariablen unverändert bleiben, fehlt noch eine Aussage über die Aktionen, die mit der Auswahl von Estelle-Transitionen zum Schalten verbunden sind. Denn bisher wurde die Auswahl zwar mit Bedingungen an den Vor-Zustand verknüpft, aber über den Nach-Zustand wurde noch keine Aussage gemacht; bisher dürfen sich alle Variablen außer den Transitionsvariablen beliebig verhalten. Natürlich wollen wir, daß alle anderen Variablen des Systems ihre Werte bei der Auswahl von Estelle-Transitionen behalten. (Wegen der Interleaving-Semantik schließen wir ein gleichzeitiges Schalten von Estelle-Transitionen aus anderen (System-)Modulinstanzen aus.) Daher spezifizieren wir, wieder unter Zuhilfenahme des Prädikatentransformators `Change`:

```
...
∧ (∀tr: tr.Id ∈ α.LocTrans:
    [  (¬tr.Selected ∧ tr.Selected′)
     ⇒ (Change.(⋃tr2: tr2.Id ∈ Sys.Trans: {tr2.Selected}))All]tr.Selected
```

Weiterhin muß das Gesamt-Ausführungsmodell auch etwas über den Anfangszustand des Systems aussagen. Eine Ausnahme unter den **`Initialize`**-Transitionen (siehe auch Kapitel 6.7) bilden die der äußersten Modulinstanz, der Gesamtspezifikation. Diese Instanz wird nicht durch eine **`Init`**-Operation erzeugt, sie existiert „von Anfang an“. Hier ist es nicht möglich, eine Aktion zu spezifizieren, die den ersten Zustand des Systems als Nachfolgezustand einer Modulinstanz-Initialisierung festlegt, denn der Anfangszustand dieser Operation müßte vor dem ersten Zustand des Systems liegen, was ein Widerspruch in sich ist. Daher beschreiben wir die Semantik dieser Initialisierung nach dem selben Verfahren wie der ISO-Standard.

Der ISO-Standard beschreibt den ersten Zustand des Gesamtsystems als sogenann-

ten präinitialen (englisch: preinitial) Zustand, in dem die äußerste Modulinstanz bereits existiert, aber in dem alle Variablen noch undefinierte Werte haben (außer den Transitions- und Modulvariablen, zu letzteren siehe dafür Kapitel 6.4). Den Zustandsübergang vom präinitialen Zustand zum Zustand nach der Initialisierung beschreiben wir durch eine besondere Transition, wir nennen sie `Sys.Ini`. Sie hat als Wirkung die Wirkung einer der **`Initialize`**-Transitionen der Gesamtspezifikation, ganz analog wie die Definition der Wirkung einer **`Init`**-Operation einer Vater-Modulinstanz aussehen würde. Diese besondere Transition `Sys.Ini` muß im präinitialen Zustand zum Schalten ausgewählt sein, und sie schaltet in einem der darauffolgenden Zustände genau wie eine gewöhnliche Estelle-Transition. Anschließend kann sie nie wieder ausgewählt werden. Formal spezifizieren wir:

$$\begin{aligned}
&\Box(\texttt{Sys.Ini.Selected} \in \texttt{Boolean})\\
\wedge\ &\texttt{Sys.Ini.Selected}\\
\wedge\ &\Box[\neg(\neg\texttt{Sys.Ini.Selected} \wedge \texttt{Sys.Ini.Selected}')]_{\texttt{Sys.Ini.Selected}}
\end{aligned}$$

Man beachte, daß vor der zweiten Zeile kein $\Box$-Operator steht, es wird eine Eigenschaft nur für den ersten Zustand spezifiziert.

Alle anderen Estelle-Transitionen sollen nicht zum Schalten ausgewählt sein oder ausgewählt werden, solange die Initialisierung durch `Sys.Ini` nicht stattgefunden hat:

$$\begin{aligned}
\Box(\ &\texttt{Sys.Ini.Selected}\\
&\Rightarrow (\forall\texttt{tr: tr.Id} \in \texttt{Sys.Trans: } \neg\texttt{tr.Selected}))
\end{aligned}$$

(Diese anderen Estelle-Transitionen müssen damit natürlich zur äußersten Modulinstanz gehören, da andere Modulinstanzen im präinitialen Zustand noch nicht existieren.)

Weiterhin müssen wir noch spezifizieren, daß das Schalten der Systeminitialisierung schließlich auch tatsächlich geschehen (sein) muß

$$\Diamond(\neg\texttt{Sys.Ini.Selected})$$

und daß bei ihrem Schalten der richtige Effekt eintritt. Als richtiger Effekt kommt dabei der Effekt einer der Initialisierungstransitionen in Frage. Gibt es davon mehrere, wird eine nichtdeterministische Auswahl getroffen, gibt es gar keine, so ändert sich als einziges der Wert der Transitionsvariablen:

$$\begin{aligned}
\Box[&(\texttt{Sys.Ini.Selected} \wedge \neg\texttt{Sys.Ini.Selected}')\\
&\Rightarrow(\ (\ \texttt{Sys.IniTrans} \neq \{\}\\
&\qquad\Rightarrow (\exists\texttt{tr: (tr.Id} \in \texttt{Sys.IniTrans):}\\
&\qquad\quad((\texttt{tr.Effect_PT}).(\texttt{Change.}\{\texttt{Sys.Ini.Selected}\}))_{\texttt{All}}))\\
&\quad\wedge(\ \texttt{Sys.IniTrans} = \{\}\\
&\qquad\Rightarrow (\texttt{Change.}\{\texttt{Sys.Ini.Selected}\})_{\texttt{All}}))]_{\texttt{Sys.Ini.Selected}}
\end{aligned}$$

Nachdem wir die spezielle Transition `Sys.Ini` eingeführt haben, müssen wir die weiter oben angeführte Formel über die Stotterschritte, in denen nichts geschieht, um diese zusätzliche Transition ergänzen:

$$\begin{aligned}
&\ldots\\
\wedge\ &\Box[(\exists\texttt{tr: tr.Id} \in \texttt{Sys.Trans} \cup \{\texttt{Sys.Ini.Id}\}:\\
&\quad\texttt{tr.Selected} \neq \texttt{tr.Selected}')]_{\texttt{All}}
\end{aligned}$$

Als nächsten Schritt zum Ausführungsmodell nehmen wir die hierarchische Modulstruktur in unsere Betrachtungen auf. Wie wir bereits gesehen haben, ist eine Estelle-Spezifikation hierarchisch in Module gegliedert. Hiermit läßt sich die Möglichkeit einer verteilten Implementierung des spezifizierten Systems andeuten. Man kann das

Gesamtsystem zwar als einen großen abstrakten Automaten aufassen, der nacheinander globale Zustandsübergänge durchführt, aber für die Hauptanwendungszwecke von Estelle sinnvoller ist die Betrachtungsweise, daß die einzelnen Modulinstanzen relativ autonom jeweils für sich einen Fortschritt erzielen.

Die Studienarbeit [Pet89] von D. Peter, die die Möglichkeiten untersucht, Estelle-Spezifikationen verteilt auszuführen, kommt zu dem Ergebnis, daß die kleinste Einheit, die sinnvoll für die Ausführung verteilt werden kann, die Modulinstanz ist. (In der Diplomarbeit [Pet91] wurde diese Verteilung dann realisiert. Eine Zusammenfassung der Arbeiten findet sich zum Beispiel in [AnBrHi91].)

Wir teilen diese Ansicht, und entsprechend werden wir unsere mathematische Beschreibung des Ausführungsmodells so strukturieren, daß eine Implementierung mit Modulinstanzen als eigenständige, verteilte Agenten, die sich in gewissem Umfange absprechen, erleichtert wird.

Aus diesem Grunde definieren wir das Ausführungsmodell rekursiv aus lokalen Teil-Ausführungsmodellen, und zwar rekursiv über die jeweils existierenden Modulinstanzen. Dazu müssen wir uns das Konzept der Definition aus der TLA$^+$ leihen (siehe Kapitel 5.4):

$$\texttt{LocAusführungsmodell}(\alpha) \triangleq \ldots$$

Dabei soll α eine Modulinstanz bezeichnen.

Das Teil-Ausführungsmodell gilt immer für die äußerste Modulinstanz des Gesamtsystems, da sie immer existiert:

$$\Box\texttt{LocAusführungsmodell(Sys)}$$

Ebenso gilt es rekursiv für alle jeweils existierenden Sohn-Modulinstanzen (siehe dazu auch Kapitel 6.4):

$$\begin{array}{l}\texttt{LocAusführungsmodell}(\alpha) \triangleq \\ \quad \ldots \\ \quad \wedge\ (\forall\beta\texttt{:}\ \beta\texttt{.Typ} \in \alpha\texttt{.LocModDefs} \wedge \beta\texttt{.Exists: LocAusführungsmodell}(\beta))\end{array}$$

(Die Rekursion ist immer endlich, da auch die Struktur der Moduldefinitionen, hier beschrieben durch den Ausdruck α`.LocModDefs`, endlich und sogar statisch ist. Die weitere Einschränkung „$\wedge$ β`.Exists`“ verkleinert den Rekursionsbaum höchstens noch weiter.)

Zusammengefaßt ergibt sich aus dem gesamten bisher Beschriebenen:

Zuerst die Definitionen

$$\texttt{All} \triangleq \texttt{(Sys, Attach, Connect)}$$

und

```
LocAusführungsmodell(α) ≜
    (∀β: β.Typ ∈ α.LocModDefs ∧ β.Exists: LocAusführungsmodell(β))
  ∧ (∀tr: tr.Id ∈ α.LocTrans:
      tr.Enab = tr.Provided ∧ tr.When ∧ α.State ∈ tr.From)
  ∧ (∀tr: tr.Id ∈ α.LocTrans:
       [  (¬tr.Selected ∧ tr.Selected′)
        ⇒ ( tr.Enab
          ∧ (∀tr2: tr2.Id ∈ α.LocTrans ∧ tr2.Priority < tr.Priority:
              ¬tr2.Enab)
          ∧ (Change.(⋃tr2: tr2.Id ∈ Sys.Trans:
                     {tr2.Selected}))All)]tr.Selected
     ∧ [  (tr.Selected ∧ ¬tr.Selected′)
        ⇒ ((tr.Effect_PT).(Change.{tr.Selected}))All]tr.Selected)
  ∧ ...
```

und

```
IniDefs ≜
    □(Sys.Ini.Selected ∈ Boolean)
  ∧ Sys.Ini.Selected
  ∧ □[¬(¬Sys.Ini.Selected ∧ Sys.Ini.Selected′)]Sys.Ini.Selected
  ∧ □(  Sys.Ini.Selected
      ⇒ (∀tr: tr.Id ∈ Sys.Trans: ¬tr.Selected))
  ∧ ◇(¬Sys.Ini.Selected)
  ∧ □[ (Sys.Ini.Selected ∧ ¬Sys.Ini.Selected′)
      ⇒( (  Sys.IniTrans ≠ {}
          ⇒ (∃tr: (tr.Id ∈ Sys.IniTrans):
              ((tr.Effect_PT).(Change.{Sys.Ini.Selected}))All))
        ∧(  Sys.IniTrans = {}
          ⇒ (Change.{Sys.Ini.Selected})All))]Sys.Ini.Selected
```

Und darauf aufbauend die Spezifikation des Gesamt-Ausführungsmodells:

```
AusführungsmodellOhneLiveness ≜
    □LocAusführungsmodell(Sys)
  ∧ IniDefs
  ∧ □[(∃tr: tr.Id ∈ Sys.Trans ∪ {Sys.Ini.Id}:
        tr.Selected ≠ tr.Selected′)]All
```

Als nächstes fügen wir die Beschreibung des Zyklus′ aus Auswählen und Schalten hinzu.

Jede Modulinstanz durchläuft für sich immer wieder diesen Zyklus. Er beginnt mit der sogenannten „Managementphase“. Sie ist dadurch gekennzeichnet, daß keine Estelle-Transition dieser Modulinstanz oder einer ihrer Sohn-Modulinstanzen zum Schalten ausgewählt ist. (Dies ist auch der Zustand nach der Erzeugung einer neuen Modulinstanz.) Bis zum Ende der Managementphase werden die Estelle-Transitionen bestimmt, die endgültig zum Schalten ausgewählt werden sollen, und das Ende selbst ist durch den Vorgang des endgültigen Auswählens (einer oder mehrerer Estelle-Transitionen) gekennzeichnet.

Eine Implementation wird die Managementphase üblicherweise dazu nutzen, für die Modulinstanz und ihre Nachfahren die Wahrheitswerte der Schaltbedingungen der Estelle-Transitionen zu berechnen, um die Schaltbereitschaft feststellen zu können. Anschließend sollte ein Auswahlalgorithmus, der die im folgenden beschriebenen Auswahlbedingungen implementiert, die Entscheidung der endgültigen

Auswahl treffen.

Außerhalb der Managementphase können keine weiteren Estelle-Transitionen ausgewählt werden. In dieser Schaltphase können die ausgewählten Estelle-Transitionen in einer beliebigen Reihenfolge schalten. Sobald alle ausgewählten dies getan haben, beginnt eine neue Managementphase.

Formal beschreiben wir dies alles durch:

```
(∀tr: tr.Id ∈ α.LocTrans:
    (¬tr.Selected ∧ tr.Selected′)
  ⇒ (∀tr2: tr2.Id ∈ α.Trans: ¬tr2.Selected))
```

Wenn man die Sohn-Modulinstanzen beiseite läßt, dann darf für jede Modulinstanz immer höchstens eine Estelle-Transition pro Zyklus zum Schalten ausgewählt werden:

```
(∀tr: tr.Id ∈ α.LocTrans:
    (¬tr.Selected ∧ tr.Selected′)
  ⇒ (∀tr2: tr2.Id ∈ α.LocTrans\{tr.Id}:
      ¬(¬tr2.Selected ∧ tr2.Selected′)))
```

Oder etwas kürzer formuliert, unter Ausnutzung der eben spezifizierten Eigenschaft, daß vor dem Auswählen einer Estelle-Transition überhaupt keine Estelle-Transition ausgewählt war:

```
(∀tr: tr.Id ∈ α.LocTrans:
    (¬tr.Selected ∧ tr.Selected′)
  ⇒ (∀tr2: tr2.Id ∈ α.LocTrans\{tr.Id}: ¬tr2.Selected′))
```

Wenn in einer Modulinstanz in der Managementphase keine Estelle-Transition zum Auswählen bereit ist, können Estelle-Transitionen ihrer Sohn-Modulinstanzen ausgewählt werden. Dies ist die Vorrangregelung zwischen Vater- und Sohn-Modulinstanzen. Der Vater hat immer Vorrang. Auf diese Weise vermeidet man in Implementationen beim Schalten Konflikte zum Beispiel beim Zugriff auf die Variablen der Söhne.

Die auswahlbereiten Estelle-Transitionen hatten wir oben bereits durch das Prädikat `tr.Enab` beschrieben. Daher spezifizieren wir nun:

```
  (∃tr: tr.Id ∈ α.LocTrans: tr.Enab)
⇒ (∀tr: tr.Id ∈ α.Trans\α.LocTrans:
    ¬(¬tr.Selected ∧ tr.Selected′))
```

(Zur Erinnerung: Die Menge α`.LocTrans` enthält die Identifikatoren aller Estelle-Transitionen der Modulinstanz α, die Menge α`.Trans` die Vereinigung hieraus mit den entsprechenden Mengen aller Sohn-Modulinstanzen und weiteren Nachfahren. Siehe Kapitel 6.7.)

Außer der Vater-Vorrangregelung unterliegen die Sohn-Modulinstanzen noch weiteren Beschränkungen, zu denen wir nun kommen.

In Kapitel 6.4 hatten wir festgelegt, daß die Übersetzungsfunktion zu jeder Modulinstanz α eine Aussage über die Klasse α`.Class` machen muß, die gleich einer der folgenden Klassen sein muß:

- `SystemProcess`
- `Process`
- `SystemActivity`
- `Activity`
- `NonAttrib`

Die Syntax von Estelle schrieb dabei fünf Bedingungen für die Klassenattributierung vor ([BuDe87]):

1) Eine Instanz mit Estelle-Transitionen darf keine `NonAttrib`-Instanz sein.
2) `SystemProcess`- oder `SystemActivity`-Instanzen können höchstens eine `NonAttrib`-Instanz als Vater haben.
3) `Process`- oder `Activity`-Instanzen müssen eine `SystemProcess`- oder `SystemActivity`-Instanz als Vater haben.
4) `Process`- oder `SystemProcess`-Instanzen können nur `Process`- oder `Activity`-Instanzen als Söhne haben.
5) `Activity`- oder `SystemActivity`-Instanzen können nur `Activity`-Instanzen als Söhne haben.

Folglich brauchen wir für Modulinstanzen der Klasse `NonAttrib` kein besonderes Auswahlverhalten vorzuschreiben, da sie niemals Estelle-Transitionen besitzen.

Modulinstanzen der Klassen `SystemProcess` und `SystemActivity` haben, wenn sie denn überhaupt eine Vater-Instanz besitzen, nur `NonAttrib`-Instanzen als Ahnen. (Daher werden sie von diesen niemals bei der Transitionsauswahl blockiert.) Auch sonst ist die Semantik von Instanzen dieser Klassen so, daß ihr Auswahlvorgang ohne irgendwelche weiteren Einschränkungen stattfinden kann. Formal spezifizieren wir dies, indem wir keine weitere Konjunktion für diesen Fall zur Definition des Ausführungsmodells hinzufügen.

Nun endlich können wir zu den angekündigten weiteren Einschränkungen kommen. Gehört eine Instanz zur Klasse SystemProcess oder Process, dann müssen mit dem Ende der Managementphase *alle* Söhne[37] (und rekursiv auch die weiteren Nachfahren) eine ihrer Estelle-Transitionen zum Schalten auswählen, sofern sie eine schaltbereite besitzen und das Auswählen nicht gegen die Vater-Sohn-Vorrangregelung verstößt. Für Instanzen der Klasse `SystemActivity` oder `Activity` muß genau *eine* Estelle-Transition der Söhne oder weiteren Nachfahren[38] ausgewählt werden (sofern überhaupt schaltbereit). Formal heißt dies:

```
(∀tr: tr.Id ∈ α.Trans:
    (¬tr.Selected ∧ tr.Selected′)
  ⇒ (  ( α.Class ∈ {SystemProcess, Process}
       ⇒ (∀β: β.Typ ∈ α.ModDefs ∧ β.Exists:
            (∃tr2: tr2.Id ∈ β.LocTrans:
              Enabled ( ¬tr2.Selected ∧ tr2.Selected′
                        ∧ LocAusführungsmodell(β)))
          ⇒ (∃tr2: tr2.Id ∈ β.LocTrans:
              ¬tr2.Selected ∧ tr2.Selected′))
     ∧ ( α.Class ∈ {SystemActivity, Activity}
       ⇒ ¬(∃tr2: tr2.Id ∈ α.Trans\{tr.Id}:
            ¬tr2.Selected ∧ tr2.Selected′))))
```

In dieser Formel erscheint der Ausdruck α.`ModDefs`, der bisher noch nicht spezifiziert wurde. Dies wollen wir daher gleich nachholen: Den Ausdruck α.`LocModDefs` hatten wir in Kapitel 6.4 als die Vereinigungsmenge aller direkten Sohn-Modul-

[37] die der Klasse `Process` oder `Activity` angehören müssen
[38] die der Klasse `Activity` angehören müssen

definitionen definiert. Auf ihrer Basis spezifizieren wir rekursiv die Menge aller Moduldefinitionen aller Nachfahren α.ModDefs:

$$\Box(\alpha.\texttt{ModDefs} = \alpha.\texttt{LocModDefs} \cup (\bigcup\beta\colon \beta.\texttt{Typ} \in \alpha.\texttt{LocModDefs}\colon \beta.\texttt{ModDefs}))$$

Weiterhin findet man etwa in der Mitte der Formel den Ausdruck

$$\textit{Enabled}\ (\neg \texttt{tr}_2.\texttt{Selected} \wedge \texttt{tr}_2.\texttt{Selected}' \wedge \texttt{LocAusführungsmodell}(\beta)) \quad .$$

Er sagt aus[39], daß es zum aktuellen Zustand einen Nachzustand geben muß, zu dem hin die Estelle-Transition $\texttt{tr}_2$ ausgewählt wird, und der mit dem Ausführungsmodell verträglich ist. Oder anders ausgedrückt: Es darf nicht verboten sein, daß $\texttt{tr}_2$ ausgewählt wird, zum Beispiel aufgrund einer entsprechenden Modulattributierung oder einer sonstigen Einschränkung des Ausführungsmodells. Da diese Aussage auf der linken Seite einer Implikation steht, wird daraus eine Bedingung, und die rechte Seite sagt dazu aus, daß $\texttt{tr}_2$ dann (das heißt, wenn es möglich ist) auch auf jeden Fall ausgewählt werden muß. Ohne die Implikation wäre dies lediglich erlaubt. Die Implikation bedeutet allerdings keinerlei Liveness-Zusage. Denn sie ist selbst wiederum nur Teil der Konsequenz des Implikationszeichens in der dritten Zeile, und dessen Bedingung besteht darin, daß überhaupt (mindestens) eine Estelle-Transition ausgewählt wird.

Nachdem nun bereits große Teile der Safety-Eigenschaften des Ausführungsmodells spezifiziert sind, wird es Zeit für Zusicherungen, daß nicht nur nichts Falsches geschieht, sonders daß auch überhaupt etwas geschieht, also für Liveness-Zusagen. Hier soll das Ausführungsmodell zwei Eigenschaften zusichern:

1) Wenn eine Modulinstanz in ihrer Managementphase ist und wenn sie dabei lange genug eine Estelle-Transition zum Schalten auswählen kann, dann muß sie dies schließlich auch tun. (Die Auswahlbereitschaft einer Estelle-Transition kann auch wieder aufgehoben werden, nämlich sobald plötzlich die Auswahlbedingungen einer Estelle-Transition der Vater-Modulinstanz erfüllt werden.)
2) Wenn eine Estelle-Transition zum Schalten ausgewählt ist, dann muß sie schließlich auch schalten.

Mit diesen Eigenschaften wird abgesichert, daß das Gesamtsystem einen Fortschritt erzielt, sofern dies möglich ist. Estelle macht aber *keine Zusagen über Fairness* bei der nichtdeterministischen Entscheidung zwischen der Auswahl verschiedener Fortschrittsmöglichkeiten.

Wenn zum Beispiel in einer Modulinstanz für zwei Estelle-Transitionen immer die Schaltbedingungen erfüllt sind, ist es einer Implementation erlaubt, jedesmal eine der beiden vorzuziehen. Ebenso dürfen einzelne von mehreren Sohn-Modulinstanzen einer `Activity`-Instanz ständig benachteiligt werden.

Die einzige Fairness-Eigenschaft, die im obigen Punkt 1) enthalten ist, ist die, daß `SystemProcess`- und `SystemActivity`-Instanzen als eigene Teilsysteme gelten, die jedes für sich schließlich einen Fortschritt erzielen müssen. (Da sie nicht von Vater-Instanzen blockiert werden können.)

---

[39] Die Definition von „*Enabled* A“ findet sich in Kapitel 3.3. Hilfreicher ist an dieser Stelle vielleicht aber die alternative Definition, die Lamport in [Lam91] in seinem Kapitel 3.7 gibt. Man kann dieses Prädikat auch wie folgt syntaktisch definieren: Wenn $\texttt{v}_1,\ldots,\texttt{v}_\texttt{n}$ sämtliche Variablen sind, die in der Aktion A vorkommen, dann sei

$$\textit{Enabled}\ A \triangleq (\exists \texttt{c}_1,\ldots,\texttt{c}_\texttt{n}\colon A(\texttt{c}_1/\texttt{v}_1',\ldots,\texttt{c}_\texttt{n}/\texttt{v}_\texttt{n}')) \quad ,$$

wobei $A(\texttt{c}_1/\texttt{v}_1',\ldots,\texttt{c}_\texttt{n}/\texttt{v}_\texttt{n}')$ die Formel bezeichnet, die man erhält, wenn man die starren Variablen $\texttt{c}_\texttt{i}$ für alle Vorkommen der $\texttt{v}_\texttt{i}'$ in A ersetzt.

Formal spezifizieren wir die Liveness-Eigenschaften so:

```
Ausführungsmodell ≜
    AusführungsmodellOhneLiveness
  ∧ (∀tr: tr.Id ∈ Sys.Trans:
        WF_tr.Selected(  ¬tr.Selected ∧ tr.Selected′
                       ∧ AusführungsmodellOhneLiveness)
      ∧ tr.Selected ⇝ ¬tr.Selected)
```

Die dritte bis fünfte Zeile spezifizieren formal unsere Forderung aus Punkt 1). Der „schwache Fairness"-Operator „$\mathtt{WF_f}(A)$" ist eine „zusätzliche Notation" der TLA und damit auch der TLA/PT, er ist definiert als:

$$\mathtt{WF_f}(A) \triangleq (\Box\Diamond\langle A\rangle_f) \vee (\Box\Diamond\neg \mathit{Enabled}\ \langle A\rangle_f)$$

Halbformal, dafür aber anschaulicher, kann man ihn erklären als:

Schwache Fairness: $\Box((\Diamond$ ausgeführt$) \vee (\Diamond$ unmöglich$))$

Im Gegensatz dazu stünde die

Starke Fairness: $\Box((\Diamond$ ausgeführt$) \vee (\Diamond\Box$ unmöglich$))$

mit

$$\mathtt{SF_f}(A) \triangleq (\Box\Diamond\langle A\rangle_f) \vee (\Diamond\Box\neg \mathit{Enabled}\ \langle A\rangle_f)$$

Anmerkung: Innerhalb des oben spezifizierten `WF`-Ausdrucks bedeutet der Zusatz

```
... ∧ AusführungsmodellOhneLiveness
```

zu

```
¬tr.Selected ∧ tr.Selected′      ,
```

daß dieses Auswählen einer Estelle-Transition verträglich mit dem sonstigen Ausführungsmodell sein muß. (Weiter oben hatten wir im Zusammenhang mit dem *Enabled*-Operator bereits etwas Ähnliches beschrieben.)

Die Eigenschaft aus Punkt 2) wird mit dem „leads-to"-Operator „$\rightsquigarrow$" spezifiziert. In Kapitel 5.1 wurde diese „zusätzliche Notation" aus der TLA in die TLA/PT übernommen, sie war definiert als:

$$\mathtt{F} \rightsquigarrow \mathtt{G} \triangleq \Box(\mathtt{F} \Rightarrow \Diamond\mathtt{G})$$

Mit diesen weiteren Spezifikationen ist bereits ein vereinfachtes Ausführungsmodell beisammen. Es fehlen lediglich noch die quantitativen Zeitaspekte. (Die Definition der einzelnen Operationen in den Transitionsrümpfen, zum Beispiel die Definition der Modulinstanz-Initialisierung mittels der Operation **`Init`**, gehört zur detaillierten Ausarbeitung der Übersetzungsfunktion für die einzelnen Estelle-Konstrukte aus Kapitel 6, aber nicht zum Ausführungsmodell.)

Daher fassen wir die bisherige Arbeit zu einem Ausführungsmodell zusammen, in dem lediglich noch die quantitativen Zeitaspekte fortgelassen sind:

Zuerst wiederum die Definitionen

```
All ≜ (Sys, Attach, Connect)
```

und

```
LocAusführungsmodell(α) ≜
    (∀β: β.Typ ∈ α.LocModDefs ∧ β.Exists: LocAusführungsmodell(β))
  ∧ (α.ModDefs =   α.LocModDefs
                 ∪ (⋃β: β.Typ ∈ α.LocModDefs: β.ModDefs))
  ∧ (∀tr: tr.Id ∈ α.LocTrans:
      tr.Enab = tr.Provided ∧ tr.When ∧ α.State ∈ tr.From)
  ∧ (∀tr: tr.Id ∈ α.LocTrans:
       [  (¬tr.Selected ∧ tr.Selected′)
        ⇒ ( tr.Enab
          ∧ (∀tr2: tr2.Id ∈ α.Trans: ¬tr2.Selected)
          ∧ (∀tr2: tr2.Id ∈ α.LocTrans\{tr.Id}: ¬tr2.Selected′)
          ∧ (∀tr2: tr2.Id ∈ α.LocTrans ∧ tr2.Priority < tr.Priority:
              ¬tr2.Enab)
          ∧ (Change.(⋃tr2: tr2.Id ∈ Sys.Trans:
                      {tr2.Selected}))All)]tr.Selected
     ∧ [  (tr.Selected ∧ ¬tr.Selected′)
        ⇒ ((tr.Effect_PT).(Change.{tr.Selected}))All]tr.Selected)
  ∧ ( (∃tr: tr.Id ∈ α.LocTrans: tr.Enab)
    ⇒ (∀tr: tr.Id ∈ α.Trans\α.LocTrans:
        [¬(¬tr.Selected ∧ tr.Selected′)]tr.Selected)
  ∧ (∀tr: tr.Id ∈ α.Trans:
      [  (¬tr.Selected ∧ tr.Selected′)
       ⇒ (  ( α.Class ∈ {SystemProcess, Process}
            ⇒ (∀β: β.Typ ∈ α.ModDefs ∧ β.Exists:
                 (∃tr2: tr2.Id ∈ β.LocTrans:
                   Enabled ( ¬tr2.Selected ∧ tr2.Selected′
                             ∧ LocAusführungsmodell(β)))
              ⇒ (∃tr2: tr2.Id ∈ β.LocTrans:
                   ¬tr2.Selected ∧ tr2.Selected′))
          ∧ ( α.Class ∈ {SystemActivity, Activity}
            ⇒ ¬(∃tr2: tr2.Id ∈ α.Trans\{tr.Id}:
                 ¬tr2.Selected ∧ tr2.Selected′)))]tr.Selected)
```

und

```
IniDefs ≜
    □(Sys.Ini.Selected ∈ Boolean)
  ∧ Sys.Ini.Selected
  ∧ □[¬(¬Sys.Ini.Selected ∧ Sys.Ini.Selected′)]Sys.Ini.Selected
  ∧ □(  Sys.Ini.Selected
      ⇒ (∀tr: tr.Id ∈ Sys.Trans: ¬tr.Selected))
  ∧ ◇(¬Sys.Ini.Selected)
  ∧ □[ (Sys.Ini.Selected ∧ ¬Sys.Ini.Selected′)
      ⇒( (  Sys.IniTrans ≠ {}
          ⇒ (∃tr: (tr.Id ∈ Sys.IniTrans):
              ((tr.Effect_PT).(Change.{Sys.Ini.Selected}))All))
        ∧(  Sys.IniTrans = {}
          ⇒ (Change.{Sys.Ini.Selected})All))]Sys.Ini.Selected
```

und

$$
\begin{array}{l}
\texttt{AusführungsmodellOhneLiveness} \triangleq \\
\quad \Box\texttt{LocAusführungsmodell(Sys)} \\
\;\wedge\; \texttt{IniDefs} \\
\;\wedge\; \Box[(\exists \texttt{tr: tr.Id} \in \texttt{Sys.Trans} \cup \{\texttt{Sys.Ini.Id}\}: \\
\qquad \texttt{tr.Selected} \neq \texttt{tr.Selected}')]_{\texttt{All}}
\end{array}
$$

Und darauf aufbauend die eigentliche Spezifikation des Ausführungsmodells:

$$
\begin{array}{l}
\texttt{Ausführungsmodell} \triangleq \\
\quad \texttt{AusführungsmodellOhneLiveness} \\
\;\wedge\; (\forall \texttt{tr: tr.Id} \in \texttt{Sys.Trans}: \\
\qquad \mathrm{WF}_{\texttt{tr.Selected}}(\;\neg\texttt{tr.Selected} \wedge \texttt{tr.Selected}' \\
\qquad\qquad\qquad\qquad \wedge\; \texttt{AusführungsmodellOhneLiveness}) \\
\qquad \wedge\; \texttt{tr.Selected} \leadsto \neg\texttt{tr.Selected})
\end{array}
$$

7.2 Das Ausführungsmodell mit quantitativen Zeitaspekten

Nun sind nur noch die quantitativen Zeitaspekte zu ergänzen. Gegen Ende von Kapitel 4.1 („Auswahl einer grundsätzlichen Semantikdefinitionsmethode für Estelle“) haben wir uns bereits für ein Grundkonzept zur Zeitdarstellung entschieden.

Das an allen Orten gleichmäßige Voranschreiten der Zeit sollte durch einen einzigen, unabhängigen „Zeit-Prozeß“ dargestellt werden. Die Estelle-Spezifikation sollte (mit Hilfe der **Delay**-Klauseln) Zugriff lediglich auf Zeitdifferenzen haben, nicht aber auf absolute Zeitpunkte. Damit wurde die notwendige gleichmäßige Verringerung von Rest-Verzögerungszeiten spezifizierbar, ohne für eine Implementation mehr annehmen zu müssen als die Existenz der universellen physikalischen Zeit nach Newton.

Den Zeit-Prozeß stellen wir durch die separate TLA/PT-Variable `Clock` dar, die eine reelle Zahl als Wert haben soll. Damit ordnen wir jedem Gesamtsystemzustand einen Zeitpunkt zu. In Kapitel 4.1 wurde die Forderung des ISO-Standards übernommen, daß jede Berechnungsfolge in die Zukunft führt, niemals aber zurück in die Vergangenheit:

$$\begin{array}{l} \quad \Box \mathtt{Clock} \in \mathtt{Real} \\ \wedge \ \Box[\mathtt{Clock}' > \mathtt{Clock}]_{\mathtt{Clock}} \end{array}$$

Wenn sich also der Wert der Variablen `Clock` ändert, dann kann er nur größer werden. Weiterhin war gefordert, daß er größer werden *muß*, wenn ein Berechnungsschritt stattfindet, der nicht nur ein Stotterschritt ohne Veränderungen ist:

$$\Box[\mathtt{Clock}' > \mathtt{Clock}]_{\mathtt{All}}$$

(Es wächst der Wert von `Clock`, oder das Gesamtsystem `All` bleibt gleich. Und wenn sich `All` nicht verändert, dann darf der Wert von `Clock` ebenfalls zunehmen, muß es aber nicht.)

Damit ist die Spezifikation des unabhängigen Zeitprozesses vollständig. Die Aufmerksamkeit kann sich nun auf die Verzögerungszeiten von Estelle-Transitionen richten.

Eine **Delay**-Klausel drückt erstens aus, daß die Estelle-Transition nicht gleich nach dem Beginn ihrer sonstigen Auswahlbereitschaft ausgewählt werden darf. Vorher muß erst eine gewisse Zeitspanne verstreichen, in der ihre anderen Auswahlbedingungen weiterhin ununterbrochen erfüllt sein müssen. Genauer gesagt diejenigen Bedingungen, die aus der **When**-, der **Provided**- und der **From**-Klausel entstehen. Die Länge dieses Zeitintervalls war für eine Estelle-Transition `tr` durch das Ergebnis der Übersetzungsfunktion, wie in Kapitel 6.7 beschrieben, gleich `tr.Delay1`.

Solange keine **Delay**-Klausel für die Estelle-Transitionen spezifiziert ist, gilt die in Kapitel 7.1 spezifizierte Liveness-Forderung, daß eine Estelle-Transition schließlich zum Schalten ausgewählt werden muß, wenn dies nur lange genug möglich ist. Nun gibt es aber noch einen zweiten durch die **Delay**-Klausel festgelegten Wert, den Wert `tr.Delay2`, der größer oder gleich `tr.Delay1` sein muß. Erst wenn diese Zeitspanne verstrichen ist, soll die Liveness-Forderung in Kraft treten, daß die Estelle-Transition schließlich ausgewählt werden muß, wenn dies möglich ist. Vorher ist es einer Implementation erlaubt zu handeln, aber sie ist nicht dazu verpflichtet. `tr.Delay2` kann auch den Wert Unendlich besitzen, in welchem Falle die Liveness-Forderung niemals wirksam werden soll.

Um feststellen zu können, wielange die Bedingungen aus den drei oben genannten Klauseln schon erfüllt sind, brauchen wir nur das Prädikat `tr.Enab` für eine Estelle-

Transition `tr` zu betrachten.

Nicht enthalten im Prädikat `tr.Enab` sind die Auswahlrestriktionen, die sich daraus ergeben, ob sich zum Beispiel die Modulinstanz gerade nicht in einer Managementphase befindet oder ob eine Estelle-Transition der Vater-Modulinstanz Vorrang hat. Laut dem ISO-Standard, Kapitel 9.6.3 bis 9.6.5, und auch [BuDe87] beeinflußt dies das Ablaufen der Verzögerungsuhren nicht. (Der Sinn dieser Regelung ist, sogenannte „Timeouts“ einfach spezifizieren zu können. Und diese Verzögerungen werden üblicherweise bestimmt durch die Bedingungen, die in den drei genannten Klauseln spezifiziert werden.)

Nachdem wir die Bedingungen für den Start der Zeitverzögerung aufgestellt haben, können wir zur Verzögerung selbst kommen. Im ISO-Standard werden jeder Estelle-Transition zwei „Eieruhren“ zugeordnet, die entweder mit dem Fortschreiten der Zeit langsam ablaufen können, oder die in einem Zustand auch nicht aktiviert sein können. Die Beschreibung dieser Vorgänge ist recht aufwendig, unter anderem auch, weil ausführlich spezifiziert werden muß, daß alle „Eieruhren“ gleichschnell laufen sollen, wenn sie laufen. Und ebenso aufgrund vieler Fallunterscheidungen, weil den Uhren entweder ein nichtnegativer reeller Wert für die verbleibende Zeit, oder aber ein spezieller Wert „!“ für das Nicht-Aktiviert-Sein zugeordnet wird.

Da unsere Semantikdefinition inhaltlich möglichst ähnlich zu der im ISO-Standard sein sollte, sehen wir uns gezwungen, die aufwendige Semantik der **`Delay`**-Klauseln zu übernehmen. (Auch wenn uns die TLA/PT-Formalisierung erlauben wird, einiges zusammenzufassen und damit zu vereinfachen; siehe unten.) Diese aufwendige Semantik ist ein deutliches Handicap bei der Verifikation von Estelle-Spezifikationen, da eine so grundlegende Sache wie das Auswählen einer Estelle-Transition zum Schalten recht aufwendig wird. Argumentationen über eine ganze Kette von schaltenden Estelle-Transitionen werden damit zu entsprechend sehr umfangreichen Ausdrücken führen. Ein Ausweg für die Praxis wird darin bestehen, sich möglichst auf Estelle-Spezifikationen ohne **`Delay`**-Klauseln zu beschränken und in einem ersten Schlußfolgerungsschritt die Formel für das Ausführungsmodell zu vereinfachen. (Mit einem Ergebnis etwa wie die bisher eingeführte Formel vom Ende des Kapitels 7.1.)

Daß die **`Delay`**-Klausel einen derartigen Aufwand erzeugt, dürfte prinzipbedingt sein. Die Verifikation von Systemen mit quantitativen zeitlichen Eigenschaften macht generell große Schwierigkeiten, da der „Zeit-Prozeß“ für die einzelnen Aktionen auf eine sehr komplexe Weise die möglichen Reihenfolgen teilweise einschränkt. Insofern ist es nicht erstaunlich, daß Estelle-Spezifikationen für die Verifikation schwer zu handhaben werden, sobald quantitative Zeitaspekte ins Spiel kommen. Bedauerlich ist lediglich, daß die Grundsatzentscheidung bei der Entwicklung von Estelle, quantitative Zeitaspekte mit aufzunehmen, jede Definition eines vollständigen Ausführungsmodells erheblich unübersichtlicher macht. Der einzige Lichtblick ist, daß es uns die TLA/PT erlaubt, mit Hilfe von Schlußfolgerungsregeln das Ausführungsmodell für eine bestimmte Estelle-Spezifikation zu vereinfachen, wenn die entsprechenden Eigenschaften niemals zum Zuge kommen können.

Was die beiden genannten speziellen Punkte betrifft, die die Übersichtlichkeit der Definition im ISO-Standard besonders beeinträchtigen, so werden wir versuchen, durch eine etwas andere Lösung eine Verbesserung zu erzielen.

Im ISO-Standard wird eine nicht laufende Verzögerungsuhr durch den speziellen Zeitwert „!“ dargestellt. Hier definieren wir stattdessen ein spezielles Prädikat, das ausdrückt, ob die Uhr läuft. (Und wenn sie nicht läuft, dann machen wir einfach keine Vorgaben für den Wert der Uhr.)

Weiterhin benötigt der ISO-Standard eine halbe Seite Formeln, um zu spezifizieren,

daß alle Verzögerungsuhren mit der gleichen relativen Geschwindigkeit ablaufen, und daß sie überhaupt und in die richtige Richtung laufen. Dies vereinfachen wir durch den Bezug auf den bereits eingeführten „Zeit-Prozeß“ `Clock`.

Ansonsten werden wir uns im folgenden aber sehr dicht an Kapitel 9.6.5 des ISO-Standards halten, um möglichst keine Differenzen in der Bedeutung aufkommen zu lassen.

Jeder Estelle-Transition sind in jedem Zustand, sofern ihre Uhr läuft, zwei Verzögerungswerte zugeordnet. In TLA/PT stellen wir dies durch eine Verbundvariable `Timer` mit den drei Komponenten `Running`, `1` und `2` dar. Jede dieser drei Komponenten besitzt die gleiche Unterstruktur. Und zwar sollen es jeweils Feldvariablen sein, deren Indexmenge die Menge der Identifikatoren der Estelle-Transitionen umfaßt. Formal spezifizieren wir:

```
□(∀tr: tr.Id ∈ Sys.Trans:
    Timer.Running[tr.Id] ∈ Boolean
  ∧ ( Timer.Running[tr.Id]
    ⇒ (  Timer.1[tr.Id] ∈ Real
       ∧ Timer.2[tr.Id] ∈ Real ∪ {Infinity})))
```

Dabei beschreibt Real die Menge der reellen Zahlen. `Infinity` ist ein spezieller Zeitwert, der eine unendlich lange Restverzögerungszeit spezifiziert. Es muß gelten:

```
□(∀x: x ∈ Real: x < Infinity  ∧  x + Infinity = Infinity)
```

(Damit folgt, daß gilt: `□(Infinity ∉ Real)`)

Die Verzögerungsuhren haben wir nicht als eine weitere Komponente der großen Verbundvariablen `Sys` spezifiziert, da wir bereits spezifiziert haben, daß sie sich nicht verändert, solange nicht wenigstens eine Estelle-Transition ausgewählt wird oder schaltet. Dies ist aber keine notwendige Bedingung dafür, daß die Verzögerungsuhren gestartet oder gestoppt werden, wie wir gleich noch sehen werden.

Die Komponenten der Verbundvariablen `Timer` müssen gewisse grundsätzliche Eigenschaften erfüllen, wie sie im ISO-Standard, Kapitel 9.6.4, Abschnitt (a), beschrieben sind:

```
□(∀tr: tr.Id ∈ Sys.Trans:
    Timer.Running[tr.Id]
  ⇒ Timer.1[tr.Id] ≤ Timer.2[tr.Id])
```

(Die restlichen Bedingungen aus diesem Abschnitt sind bereits durch unsere veränderte Darstellung der Uhren automatisch erfüllt.) Auch diese Eigenschaft wird sich aus den im folgenden spezifizierten Eigenschaften ableiten lassen, so daß wir sie nicht mehr explizit aufführen müssen.

In Abschnitt (b) dieses Kapitels des ISO-Standards wird gefordert, daß im präinitialen Zustand des Systems alle Uhren abgeschaltet sind:

```
(∀tr: tr.Id ∈ Sys.Trans: ¬Timer.Running[tr.Id])
```

Es ist dort nicht weiter erwähnt, aber diese Forderung muß selbstverständlich auch entsprechend für jede später neu erzeugte Modulinstanz gelten. (Allerdings gehört die Modulinstanzerzeugung mittels der **`Init`**-Operation wie erwähnt bereits nicht mehr zur Thematik des Aufführungsmodells, also von Kapitel 7.)

In Abschnitt (c) wird das An- und Abschalten sowie das Laufen der Uhren beschrieben:

(1) Außerhalb der Managementphase werden keine Uhren angehalten oder gestartet:

```
□(  ...
  ∧ (∀tr: tr.Id ∈ α.Trans ∧ α.Class ∈ {SystemProcess,
                                       SystemActivity}:
        tr.Selected
      ⇒ (∀tr₂: tr₂.Id ∈ α.Trans: Unchanged Timer.Running[tr₂.Id])))
```

(Anmerkung: Auch dies ist eine TLA/PT-Formel der Form $\Box[A]_g$, da
(□ *Unchanged* f) = (□ f′=f) = (□[False]$_f$))

(2) In der Managementphase gilt:
(I) Wenn eine Uhr angestellt wird, dann war die Estelle-Transition schaltbereit, wurde aber nicht ausgewählt, und die Verzögerungswerte sind zulässig:

```
□(  ...
  ∧ (∀tr: tr.Id ∈ α.Trans ∧ α.Class ∈ {SystemProcess,
                                       SystemActivity}:
      [  (¬Timer.Running[tr.Id] ∧ Timer.Running[tr.Id]′)
       ⇒ ( tr.Enab
         ∧ ¬tr.Selected′
         ∧ 0 ≤ tr.Delay1 ≤ tr.Delay2
         ∧ Timer.1[tr.Id]′ = Clock + tr.Delay1
         ∧ Timer.2[tr.Id]′ = Clock + tr.Delay2)]Timer.Running[tr.Id]))
```

(Anmerkung: Die Bedingung, daß dies nur in der Management-Phase geschehen kann, folgt wegen (1) bereits daraus, daß sich die Zustandsfunktion `Timer.Running[tr.Id]` verändert.)

(II) Wenn eine Uhr abgestellt wird, dann war die Estelle-Transition entweder nicht mehr schaltbereit, oder sie wurde ausgewählt:

```
□(  ...
  ∧ (∀tr: tr.Id ∈ α.Trans ∧ α.Class ∈ {SystemProcess,
                                       SystemActivity}:
      [  (Timer.Running[tr.Id] ∧ ¬Timer.Running[tr.Id]′)
       ⇒ ( ¬tr.Enab
         ∨ tr.Selected′)]Timer.Running[tr.Id]))
```

Zusammen mit der von der Übersetzungsfunktion gelieferten Aussage, daß sich die Variablen `tr.Delay1` und `tr.Delay2` während der „Lebenszeit“ einer Modulinstanz nicht ändern können, können wir die Formeln aus (I) und (II) zusammenfassen zu:

```
□(  ...
  ∧ (∀tr: tr.Id ∈ α.Trans ∧ α.Class ∈ {SystemProcess,
                                       SystemActivity}:
        [  Timer.Running[tr.Id]′
         = ( tr.Enab
           ∧ ¬tr.Selected′
           ∧ 0 ≤ tr.Delay1 ≤ tr.Delay2)]Timer.Running[tr.Id]
      ∧ [  (¬Timer.Running[tr.Id] ∧ Timer.Running[tr.Id]′)
         ⇒ ( Timer.1[tr.Id]′ = Clock + tr.Delay1
           ∧ Timer.2[tr.Id]′ = Clock + tr.Delay2)]Timer.Running[tr.Id]))
```

Dabei sagt die erste Hälfte der Formel: *Wenn* sich die Variable `Timer.Running[tr.Id]` ändert, dann muß sie den richtigen Wert annehmen. Solange die Managementphase nicht endet, ist die Variable allerdings zu keiner Änderung verpflichtet. (Dies ergibt sich dadurch, daß die Eigenschaft als TLA/PT-Formel spezifiziert wurde, weshalb die Aktion immer auch durch einen Stotterschritt ersetzt

werden kann.)

Die zweite Hälfte der Formel sagt unverändert, daß bei einem Einschalten der Uhr die Verzögerungswerte richtig gesetzt werden müssen. Dieser Teil mit den Wirkungen des Einschaltens ist nun von den Bedingungen für das Einschalten getrennt.

(3) Am Ende einer Managementphase, also bei dem Auswählen von Estelle-Transitionen, gilt:
(I) Wenn die Uhr abgeschaltet war, wenn die Estelle-Transition schaltbereit war, wenn sie aber nicht ausgewählt wird und wenn die Verzögerungswerte korrekt sind, dann muß die Uhr gestartet werden:

```
□(  ...
  ∧ (∀tr: tr.Id ∈ α.Trans ∧ α.Class ∈ {SystemProcess,
                                       SystemActivity}:
      [  (¬tr.Selected ∧ tr.Selected′)
       ⇒ (∀tr₂: tr₂.Id ∈ α.Trans:
              ( ¬Timer.Running[tr₂.Id]
              ∧ tr₂.Enab
              ∧ ¬tr₂.Selected′
              ∧ 0 ≤ tr₂.Delay1 ≤ tr₂.Delay2)
            ⇒ Timer.Running[tr₂.Id]′)]tr.Selected))
```

(Die Startwerte für die Uhr haben wir bereits bei Punkt (2) spezifiziert. Der ISO-Standard muß hier aufgrund seiner ungünstigeren Darstellung eine Aussage doppelt spezifizieren.)

(II) Wenn die Estelle-Transition nicht schaltbereit war, oder sie nun ausgewählt wird, dann muß die Uhr abgeschaltet werden:

```
□(  ...
  ∧ (∀tr: tr.Id ∈ α.Trans ∧ α.Class ∈ {SystemProcess,
                                       SystemActivity}:
      [  (¬tr.Selected ∧ tr.Selected′)
       ⇒ (∀tr₂: tr₂.Id ∈ α.Trans:
              ( ¬tr₂.Enab
              ∨ tr₂.Selected′)
            ⇒ ¬Timer.Running[tr₂.Id]′)]tr.Selected)))
```

Ebenso wie bei Punkt (2) können wir Punkt (3)(I) und (3)(II) zusammenfassen:

```
□(  ...
  ∧ (∀tr: tr.Id ∈ α.Trans ∧ α.Class ∈ {SystemProcess,
                                       SystemActivity}:
      [  (¬tr.Selected ∧ tr.Selected′)
       ⇒ (∀tr₂: tr₂.Id ∈ α.Trans:
              Timer.Running[tr₂.Id]′
            = ( tr₂.Enab
              ∧ ¬tr₂.Selected′
              ∧ 0 ≤ tr₂.Delay1 ≤ tr₂.Delay2))]tr.Selected))
```

Man erkennt plötzlich, daß es sich um fast dieselbe Aussage handelt wie bei dem ersten Teil von Punkt (2), nur daß hier das Ende der Managementphase als zusätzliche Bedingung auftritt und daß in diesem Falle die Aktion nicht mehr durch einen Stotterschritt ersetzt werden darf. Wir fassen daher auch Punkt (2) und (3) insgesamt zusammen:

```
□(  ...
  ∧ (∀tr: tr.Id ∈ α.Trans ∧ α.Class ∈ {SystemProcess,
                                       SystemActivity}:
        (∀tr2: tr2.Id ∈ α.Trans:
          [  ( (Timer.Running[tr2.Id] ≠ Timer.Running[tr2.Id]′)
             ∨ (¬tr.Selected ∧ tr.Selected′))
           ⇒ (  Timer.Running[tr2.Id]′
              = ( tr2.Enab
                ∧ ¬tr2.Selected′
                ∧ 0 ≤ tr2.Delay1 ≤ tr2.Delay2))
          ]_(Timer.Running[tr2.Id],tr.Selected))
      ∧ [  (¬Timer.Running[tr.Id] ∧ Timer.Running[tr.Id]′)
         ⇒ ( Timer.1[tr.Id]′ = Clock + tr.Delay1
           ∧ Timer.2[tr.Id]′ = Clock + tr.Delay2)]_Timer.Running[tr.Id]))
```

Die erste Hälfte der Formel sagt nun: *Falls* der „Uhrenschalter“ `Timer.Running[tr.Id]` innerhalb der Managementphase seine Stellung ändert, dann muß er dabei die richtige Stellung einnehmen. Und am Ende der Managementphase *muß* es das auf jeden Fall tun.

Die zweite Hälfte der Formel sagt unverändert, daß bei einem Einschalten der Uhr die Verzögerungswerte richtig gesetzt werden müssen.

Punkt (4) dieses Kapitels des ISO-Standards spezifiziert ausführlich das gleichmäßige Laufen aller Uhren. Wir müssen aufgrund unserer Darstellung mit einer allgemeinen Zeit `Clock` lediglich spezifizieren, daß die von den Uhr-Variablen festgehaltenen Endzeitpunkte nicht verändert werden dürfen, solange die Uhren laufen:

```
□(∀tr: tr.Id ∈ Sys.Trans:
     Timer.Running[tr.Id]
  ⇒ ( Unchanged Timer.1.[tr.Id]
    ∧ Unchanged Timer.2.[tr.Id]))
```

Punkt (d) spezifiziert eine Liveness-Zusage, nach der die Zeit schließlich voranschreiten muß. Dies haben wir bereits dadurch in TLA/PT spezifiziert, daß einerseits das System einen Fortschritt erzielen muß und andererseits die Zeit bei einem Systemfortschritt ebenfalls voranschreiten muß.

Nun endlich können wir die Spezifikation des Auswählens einer Estelle-Transition um die Verzögerungsbedingung ergänzen.

Sie darf ausgewählt werden, wenn einerseits die normalen Bedingungen erfüllt sind und andererseits erstens der erste Verzögerungswert `tr.Delay1` gleich Null ist oder zweitens der erste Verzögerungszeitpunkt `Timer.1[tr.Id]` vorbei ist. Daher müssen wir in unserem bisherigen, vereinfachten Ausführungsmodell den Ausdruck `tr.Enab` ersetzen durch `tr.DelayEnab`:

```
  tr.DelayEnab
=   tr.Enab
  ∧ ( (tr.Delay1 = 0 ∧ 0 ≤ tr.Delay2)
    ∨ (Timer.Running[tr.Id] ∧ Timer.1[tr.Id] ≤ Clock))
```

Anmerkung: An dieser Stelle kann man beim Verifizieren seine Formeln entscheidend vereinfachen, wenn für einige oder noch besser alle Estelle-Transitionen gilt, daß `tr.Delay1 = 0` (und `0 ≤ tr.Delay2`) ist. Damit kann man die letzten beiden Zeilen eliminieren, wodurch der gesamte Bezug auf die Uhren abgekoppelt

wird.

Als nächstes betrachten wir die Liveness-Zusagen. Eine davon muß im Lichte des zweiten Verzögerungswertes leicht revidiert werden, denn eine Estelle-Transition soll nicht zwangsweise ausgewählt werden, solange ihre zweite Uhr nicht abgelaufen ist.

Bisher hatten wir die folgende „schwache Fairness"-Bedingung spezifiziert:

```
(∀tr: tr.Id ∈ Sys.Trans:
  WF_tr.Selected(  ¬tr.Selected ∧ tr.Selected′
                 ∧ AusführungsmodellOhneLiveness))
```

Daher spezifizieren wir dies nun mit der zusätzlichen Randbedingung:

```
(∀tr: tr.Id ∈ Sys.Trans:
  WF_tr.Selected(  ¬tr.Selected ∧ tr.Selected′
                 ∧ AusführungsmodellOhneLiveness
                 ∧ Timer.Running[tr.Id] ∧ Timer.2[tr.Id] ≤ Clock))
```

Weiterhin benötigen wir noch eine zusätzliche Liveness-Bedingung, die sicherstellt, daß eine Verzögerungsuhr schießlich auch gestartet wird, wenn dies lange genug möglich ist:

```
(∀tr: tr.Id ∈ Sys.Trans:
  WF_Timer.Running[tr.Id](  ¬Timer.Running[tr.Id] ∧ Timer.Running[tr.Id]′
                          ∧ AusführungsmodellOhneLiveness))
```

Ohne diese Bedingung könnte ein System sonst in der Managementphase „hängenbleiben", wenn es zwar keine Estelle-Transition auswählen kann, aber zumindest eine Verzögerungsuhr dafür starten könnte.

Nun können wir endlich alle Ergebnisse des Kapitels 7 zusammenfassen:

Zuerst wiederum die Definitionen

```
All ≜ (Sys, Attach, Connect)
```

und

```
LocTimerDefs(α) ≜
    (∀tr: tr.Id ∈ α.Trans ∧ α.Class ∈ {SystemProcess,
                                       SystemActivity}:
        (  tr.Selected
         ⇒ (∀tr_2: tr_2.Id ∈ α.Trans: Unchanged Timer.Running[tr_2.Id]))
      ∧ (∀tr_2: tr_2.Id ∈ α.Trans:
          [  ( (Timer.Running[tr_2.Id] ≠ Timer.Running[tr_2.Id]′)
             ∨ (¬tr.Selected ∧ tr.Selected′))
           ⇒ (  Timer.Running[tr_2.Id]′
              = ( tr_2.Enab
                ∧ ¬tr_2.Selected′
                ∧ 0 ≤ tr_2.Delay1 ≤ tr_2.Delay2))
          ](Timer.Running[tr2.Id],tr.Selected))
      ∧ [  (¬Timer.Running[tr.Id] ∧ Timer.Running[tr.Id]′)
         ⇒ ( Timer.1[tr.Id]′ = Clock + tr.Delay1
           ∧ Timer.2[tr.Id]′ = Clock + tr.Delay2)]_Timer.Running[tr.Id])
```

und

```
LocAusführungsmodell(α) ≜
    LocTimerDefs(α)
  ∧ (∀β: β.Typ ∈ α.LocModDefs ∧ β.Exists: LocAusführungsmodell(β))
  ∧ (α.ModDefs =   α.LocModDefs
                 ∪ (⋃β: β.Typ ∈ α.LocModDefs: β.ModDefs))
  ∧ (∀tr: tr.Id ∈ α.LocTrans:
       tr.Enab = tr.Provided ∧ tr.When ∧ α.State ∈ tr.From
     ∧ (  tr.DelayEnab
        =   tr.Enab
          ∧ ( (tr.Delay1 = 0 ∧ 0 ≤ tr.Delay2)
            ∨ (Timer.Running[tr.Id] ∧ Timer.1[tr.Id] ≤ Clock))))
  ∧ (∀tr: tr.Id ∈ α.LocTrans:
       [  (¬tr.Selected ∧ tr.Selected′)
        ⇒ ( tr.DelayEnab
          ∧ (∀tr2: tr2.Id ∈ α.Trans: ¬tr2.Selected)
          ∧ (∀tr2: tr2.Id ∈ α.LocTrans\{tr.Id}: ¬tr2.Selected′)
          ∧ (∀tr2: tr2.Id ∈ α.LocTrans ∧ tr2.Priority < tr.Priority:
              ¬tr2.DelayEnab)
          ∧ (Change.(⋃tr2: tr2.Id ∈ Sys.Trans:
                     {tr2.Selected}))All)]tr.Selected
     ∧ [  (tr.Selected ∧ ¬tr.Selected′)
        ⇒ ((tr.Effect_PT).(Change.{tr.Selected}))All]tr.Selected)
  ∧ ( (∃tr: tr.Id ∈ α.LocTrans: tr.DelayEnab)
    ⇒ (∀tr: tr.Id ∈ α.Trans\α.LocTrans:
        [¬(¬tr.Selected ∧ tr.Selected′)]tr.Selected)
  ∧ (∀tr: tr.Id ∈ α.Trans:
      [  (¬tr.Selected ∧ tr.Selected′)
       ⇒ (  ( α.Class ∈ {SystemProcess, Process}
            ⇒ (∀β: β.Typ ∈ α.ModDefs ∧ β.Exists:
                 (∃tr2: tr2.Id ∈ β.LocTrans:
                   Enabled ( ¬tr2.Selected ∧ tr2.Selected′
                             ∧ LocAusführungsmodell(β)))
               ⇒ (∃tr2: tr2.Id ∈ β.LocTrans:
                   ¬tr2.Selected ∧ tr2.Selected′))
          ∧ ( α.Class ∈ {SystemActivity, Activity}
            ⇒ ¬(∃tr2: tr2.Id ∈ α.Trans\{tr.Id}:
                 ¬tr2.Selected ∧ tr2.Selected′)))]tr.Selected)
```

und

```
TimerDefs ≜
    □Clock ∈ Real
  ∧ □[Clock′ > Clock]Clock
  ∧ □[Clock′ > Clock]All
  ∧ □(∀tr: tr.Id ∈ Sys.Trans:
        Timer.Running[tr.Id] ∈ Boolean
      ∧ ( Timer.Running[tr.Id]
        ⇒ (  Timer.1[tr.Id] ∈ Real
           ∧ Timer.2[tr.Id] ∈ Real ∪ {Infinity}
           ∧ Unchanged Timer.1.[tr.Id]
           ∧ Unchanged Timer.2.[tr.Id]))
  ∧ □(∀x: x ∈ Real: x < Infinity  ∧  x + Infinity = Infinity)
  ∧ (∀tr: tr.Id ∈ Sys.Trans: ¬Timer.Running[tr.Id])
```

und

```
IniDefs ≜
    □(Sys.Ini.Selected ∈ Boolean)
  ∧ Sys.Ini.Selected
  ∧ □[¬(¬Sys.Ini.Selected ∧ Sys.Ini.Selected′)]_Sys.Ini.Selected
  ∧ □(  Sys.Ini.Selected
      ⇒ (∀tr: tr.Id ∈ Sys.Trans: ¬tr.Selected))
  ∧ ◇(¬Sys.Ini.Selected)
  ∧ □[ (Sys.Ini.Selected ∧ ¬Sys.Ini.Selected′)
      ⇒( (  Sys.IniTrans ≠ {}
          ⇒ (∃tr: (tr.Id ∈ Sys.IniTrans):
              ((tr.Effect_PT).(Change.{Sys.Ini.Selected}))_All))
        ∧(  Sys.IniTrans = {}
          ⇒ (Change.{Sys.Ini.Selected})_All))]_Sys.Ini.Selected
```

und

```
AusführungsmodellOhneLiveness ≜
    □LocAusführungsmodell(Sys)
  ∧ TimerDefs
  ∧ IniDefs
  ∧ □[(∃tr: tr.Id ∈ Sys.Trans ∪ {Sys.Ini.Id}:
        tr.Selected ≠ tr.Selected′)]_All
```

Und darauf aufbauend die eigentliche Spezifikation des Ausführungsmodells:

```
Ausführungsmodell ≜
    AusführungsmodellOhneLiveness
  ∧ (∀tr: tr.Id ∈ Sys.Trans:
        WF_tr.Selected(  ¬tr.Selected ∧ tr.Selected′
                       ∧ AusführungsmodellOhneLiveness
                       ∧ Timer.Running[tr.Id] ∧ Timer.2[tr.Id] ≤ Clock)
      ∧ WF_Timer.Running[tr.Id](  ¬Timer.Running[tr.Id] ∧ Timer.Running[tr.Id]′
                                ∧ AusführungsmodellOhneLiveness)
      ∧ tr.Selected ⇝ ¬tr.Selected)
```

Eine Bewertung des hiermit Erreichten werden wir in Kapitel 8 vornehmen.

8. Ergebnisse

In diesem Kapitel soll zusammengefaßt werden, was in dieser Arbeit durchgeführt wurde und was damit erreicht wurde. Im nächsten Kapitel folgt dann noch ein Ausblick, in welche Richtungen fortsetzende Arbeit sinnvoll erscheint.

Ursprünglich wurde diese Arbeit begonnen mit der Aufgabenstellung, für Estelle ein Ausführungsmodell formal zu definieren. Denn der ISO-Standard [ISO89a], der Estelle definiert, tut dies nicht formal und nicht verständlich genug, um das zu ermöglichen, was wir als einen der Hauptnutzen einer Formalisierung von Estelle ansehen: Nämlich die Eigenschaften einer Estelle-Spezifikation analysieren zu können und sie gegen eine (noch) abstraktere formale Beschreibung verifizieren zu können. (Der zweite wichtige Nutzen ist eine Präzisierung der nach wie vor vorhandenen natürlichsprachlichen Beschreibung, damit ein Anwender Zweifelsfälle entscheiden kann. Siehe Kapitel 1.)

Weiterhin war Teil der Aufgabenstellung, das Estelle-Ausführungsmodell auf der Grundlage prädikatenlogischer Zusicherungen für die Estelle-Transitionen zu definieren. Leider stellte es sich heraus, daß dies nicht möglich ist. Deshalb haben wir zunächst grundsätzlich untersucht und herausgefunden, auf welche Weise man die Semantik von Estelle überhaupt für die genannten Zwecke am geeignetsten beschreiben kann.

Die Semantikdefinitionsmethoden (für sequentielle Programme) lassen sich einteilen in die Übersetzersemantik, die operationale Semantik, die denotationale Semantik und die axiomatische Semantik. Die Reihenfolge der Aufzählung gibt gleichzeitig auch eine Zunahme der Eignung für die Verifikation an. (Siehe Kapitel 3.)

Eine denotationale und eine axiomatische Semantik als Grundlage für eine Definition der Semantik von Estelle schieden aber aus, da mit Estelle keine Systeme spezifiziert werden, die genau eine Eingabe bekommen und (höchstens) eine Ausgabe liefern. Stattdessen werden mit Estelle mögliche Abfolgen von Systemzuständen spezifiziert - oder bei anderer Betrachtung Folgen von Zustandsübergängen, inklusive Kommunikation. Auf jeden Fall aber *Folgen* von Schritten. Nur eine operationale Semantik kann dies gut darstellen.

Estelle dient insbesondere dazu, verteilte Systeme zu spezifizieren. Daher mußte entschieden werden, auf welche Weise Nebenläufigkeit dargestellt werden soll. Es wurde festgestellt, daß mit Estelle das zulässige Verhalten eines Systems spezifiziert wird, also das, was beobachtbar ist, aber nicht die Gründe für dieses Verhalten. Daher wurde das Konzept der Kausalität in der Beschreibung nicht benötigt. Und so konnte „Nebenläufigkeit“ gleichgesetzt werden mit „Nichtdeterminismus bei der Wahl der Reihenfolge“. Als Konsequenz hiervon reichten gewöhnliche total geordnete Folgen („Ausführungspfade“) für die Beschreibung der Semantik von Estelle aus, Teilordnungen waren nicht notwendig.

Es wurden die Methoden untersucht, mit denen ein Transitionssystem verifiziert werden kann, also ein operational definiertes System. Es ergab sich (unter anderem), daß es ausreichend und angemessen ist, Estelle-Transitionen mit prädikatenlogischen Mitteln darzustellen. Weiterhin ergab sich, daß Lamports temporale Logik der Aktionen (TLA, [Lam91]) alle wichtigen Methoden unterstützt.

Ursprünglich hatten wir beabsichtigt, einen Ansatz zur Semantikdefinition zu verfolgen, bei dem die operationale Semantik durch einen expliziten abstrakten Automaten angegeben wird, so daß dessen aktueller Zustand durch eine komplizierte Struktur aus Tupeln und Mengen definiert würde. Hierauf würde dann eine entsprechende Zustandsübergangsrelation definiert werden. Der ISO-Standard beruht

auf einem solchen Ansatz. Aber es zeigte sich, daß die Tupel viel zu umfangreich und dadurch zu unübersichtlich wurden, um mit ihnen noch irgendwie praktisch arbeiten zu können. Daher entschlossen wir uns, den Automaten nicht explizit anzugeben und stattdessen seine Eigenschaften auf der Basis der Prädikatenlogik zu spezifizieren. Alle Komponenten des Gesamtzustandes werden durch einzelne Variablen beschrieben, und man braucht sich nur auf die jeweils relevanten Teile zu beziehen, sowohl bei der Systembeschreibung, wie auch bei der Argumentation über einzelne Eigenschaften des Systems. Im Rahmen dieser Entscheidung stießen wir auf die TLA, und sie bietet genau die gewünschte Art der Darstellung des Zustandes. Auch ihre „Aktionen", die Zustandsübergänge, ermöglichen es, auf die gleiche Weise jeweils nur über Teile des Systems zu reden.

Da Estelle auf erweiterten endlichen Automaten aufbaut, spielen die Estelle-Transitionen eine sehr wichtige Rolle. Aus diesem Grunde wurde untersucht, wie die Semantik der Pascal-ähnlichen Transitionsrümpfe am geeignetsten auf prädikatenlogische Weise dargestellt werden kann. Dies geschah im Hinblick darauf, wie gut jeweils die oben erwähnten Beweisverfahren für Transitionssysteme unterstützt werden. Denn die Voraussetzungen für formales Schließen über Estelle-Spezifikationen sind eine geeignete formale Beschreibung der nachzuweisenden Eigenschaften, eine geeignete Formalisierung der Relation von aufeinanderfolgenden Zuständen sowie ein geeignetes formales Schlußsystem.

Eine besondere Schwierigkeit bei der Darstellung der Estelle-Transitionen ergab sich dadurch, daß der Transitionsrumpf in einer Pascal-ähnlichen Art ausgedrückt wird. Dies bedeutet, daß die sequentielle Komposition ein gewichtiges Ausdrucksmittel ist, und dies, obwohl eine Estelle-Transition konzeptuell eine atomare Zustandsänderung beschreibt, also eine *ohne* Zwischenschritte.

Es wurden anhand eines Beispiels sehr ausführlich die verschiedenen prädikatenlogischen Darstellungsformen für Estelle-Transitionen verglichen:

Zur Darstellung von Transitionen als jeweils einzelnes Prädikat existierte bereits ein fertiger, guter Formalismus, die TLA. Mit ihr ist formales Schließen über temporale Eigenschaften eines Transitionssystems sehr schön möglich. Aber leider ist dieser Ansatz nur zur Darstellung abstrakter Algorithmen und zu ihrer Verfeinerung geeignet, nicht zur Analyse sequentiell ausgedrückter Rümpfe von Estelle-Transitionen.

Weiterhin wurde die Darstellung durch eine Zusicherung nach Hoare aus Vorbedingung, Transition und Nachbedingung untersucht und begründet, warum dieser ursprünglich geplante Ansatz nicht geeignet ist.

Dann wurden Prädikatentransformatoren nach Dijkstra ([DiSc90]) untersucht, sowohl die „stärkste Nachbedingung", als auch die „schwächste Vorbedingung". Beide sind gut geeignet, die Wirkung von Estelle-Transitionsrümpfen zu definieren. Die „schwächste Vorbedingung" erweist sich als die handlichere und ausdrucksstärkere. Allerdings ist dieser Ansatz, wie oben erwähnt, nicht geeignet, um Transitionssysteme und ihre temporalen Eigenschaften zu beschreiben.

Schließlich wurde die Möglichkeit einer Mischung der Darstellungsformen erwogen. Sie bringt keinen weiteren Nutzen für die Definition der Bedeutung von Estelle-Transitionsrümpfen.

Aber mit einer Mischform erreichen wir alles, was wir für die Gesamt-Definition benötigen. Denn mit ihr können jeweils die Eigenschaften der einen Beschreibungsform die fehlenden der anderen ergänzen:

Die Vorteile der TLA und der Prädikatentransformatoren lassen sich kombinieren, und zwar einerseits die Eignung für die Verifikation von Transitionssystemen, die Beschreibbarkeit von Eigenschaften auf einer höheren Abstraktionsebene und das

fertige Schlußsystem dazu und andererseits die Darstellbarkeit der Estelle-Transitionsrümpfe und die Fähigkeit, deren Eigenschaften zu verifizieren. Es stellte sich als am günstigsten heraus, Prädikatentransformatoren direkt in die TLA zu integrieren, da sich so Korrektheitsbeweise am leichtesten führen lassen.

Auf diese Weise werden alle drei oben genannten Bedingungen für formales Schließen über Estelle-Spezifikationen durch einen einzigen Formalismus erfüllt.

So wurde eine „temporale Logik der Aktionen, mit Prädikatentransformatoren" (TLA/PT) entworfen. Hierzu wurden die Syntax und die Semantik der TLA/PT ausgearbeitet, die Schlußregeln dagegen wurden nur noch skizziert. Eine vollständige Ausarbeitung, einschließlich eines Beweises der Korrektheit und der Konsistenz, war nicht mehr Gegenstand dieser Arbeit. Denn wir wollten lediglich einen Ansatz erarbeiten, wie die Semantik von Estelle definiert werden kann, um das Ausführungsmodell definieren zu können. Erst für die eigentliche Definition der Semantik von Estelle wäre auch eine vollständige TLA/PT notwendig.

Die Fähigkeiten der TLA/PT umfassen zunächst einmal alle der TLA. Die TLA ist insbesondere gedacht für das Gebiet der verteilten Systeme. Sie vereinigt drei Konzepte („Programm", „Eigenschaft", „erfüllt") in nur einem Konzept („logische Formel"). Hierdurch ist sie einfach, und ihre Anwendung auf praktische Probleme ist möglich. Die TLA ist für folgenden Einsatz vorgesehen: Zuerst werden die Eigenschaften eines Algorithmus′ mit TLA spezifiziert, anschließend wird er in Verfeinerungsschritten entwickelt, und schließlich wird mit TLA gezeigt, daß das resultierende TLA-Programm die Eigenschaften erfüllt. Die TLA erlaubt es, dies alles mit nur einem einzigen Formalismus durchzuführen. Bisher ist die TLA nur zur Beschreibung abgeschlossener Systeme geeignet.

Aufgebaut ist die TLA aus einer Logik von sogenannten Aktionen einerseits und aus einer einfachen temporalen Logik andererseits. Mit TLA können auch Liveness-Eigenschaften ausgedrückt werden. Der größte Teil der Arbeit einer Verifikation in TLA spielt sich im Gebiet der Aktionen ab, also in einfacher, gewöhnlicher Prädikatenlogik. Temporale, aufwendige Argumente sind nur in geringem Umfange erforderlich. Da TLA-Formeln invariant gegen das sogenannte „Stottern" sind, unterstützt die TLA eine sehr große Spannweite im Grad der Abstraktion bei der Verfeinerung von Eigenschaften zu einem Programm. Die TLA befindet sich noch in der Weiterentwicklung, eine TLA^+ soll mehr syntaktische Hilfen für den Umgang mit großen und sehr großen Formeln geben. Eine Erweiterung hin zu offenen Systemen wird überlegt. Weiterhin werden derzeit Ansätze zu einer Mechanisierung des Schlußfolgerns in TLA entwickelt.

Die TLA/PT fügt zur TLA die Prädikatentransformatoren nach Dijkstra hinzu. In dieser axiomatischen Semantikdefinitionstechnik wird nicht mit Zuständen, sondern nur noch mit deren Eigenschaften gearbeitet, beschrieben durch Prädikate. Die Punkte des Zustandsraumes sind anonym, erst durch Prädikate werden sie unterscheidbar, indem die Prädikate als Koordinatenachsen fungieren. Sie teilen die Zustände in Klassen ein. Entsprechend beschreibt ein Prädikatentransformator eine Koordinatentransformation im Zustandsraum.

Eine terminierende Operation wird aufgefaßt als eine Relation zwischen Anfangs- und Endzuständen. Da aber Relationen schlechter handzuhaben sind als Funktionen, werden Funktionen über Zustandsklassen definiert, eben die Prädikatentransformatoren. Um die Semantik der Zeichenketten einer Sprache zu beschreiben, werden Funktionen von diesen nach Prädikatentransformatoren definiert. Auch Nichttermination läßt sich gut beschreiben, ebenso Nichtdeterminismus.

Da Prädikatentransformatoren Operationen auf einer sehr hohen Abstraktionsebene beschreiben, sind sie ohnehin für Zwecke der Verifikation sehr gut geeignet. Das

Kalkül der Prädikatentransformatoren selbst kommt völlig ohne den Begriff des Zustandes aus. Erst sobald man damit die Semantik von Operationen definiert, benötigt man ihn für die Anfangs- und Endzustände. TLA-Aktionen betrachten ebenfalls nur je einen Vor- und Nachzustand, aber nur mit Prädikatentransformatoren läßt sich sequentielle Komposition größeren Ausmaßes gut beschreiben. Diese kommt in den Pascal-artigen Rümpfen von Estelle-Transitionen vor. Andererseits erlaubte es uns die genannte Ähnlichkeit, einen Prädikatentransformator in der TLA/PT als eine besondere Art von Aktion zu definieren. Damit hat man einerseits die herkömmlichen TLA-Aktionen für die bisherigen Zwecke, und außerdem die Prädikatentransformatoren als Aktionen, mit denen man die Pascal-artig ausgedrückten, atomaren Operationen der Estelle-Transitionsrümpfe beschreiben kann. Weiterhin, um die Verbindung der beiden Formalismen orthogonal zu machen, können Prädikatentransformatoren in der TLA/PT auch auf gewöhnliche Prädikate der TLA angewandt werden.

Bei der Verbindung der Formalismen ist es uns gelungen, die Einfachheit der Logik, die die Stärke der TLA (und auch von Dijkstras Formalismus) ist, in der TLA/PT zu erhalten. (Dies betrifft zum Beispiel den Punkt, welche Aussagen über Variablen gemacht werden, die in einer Formel *nicht* explizit erwähnt werden.)

Nachdem die TLA (nur) die Verfeinerung von einer Eigenschafts-Spezifikation zu einem Programm ermöglicht, unterstützt die TLA/PT nun auch die entgegengesetzte Richtung. Man kann mit ihr die Eigenschaften einer vorgegebenen Estelle-Spezifikation herausfinden, beziehungsweise ihre Semantik definieren. Dabei bleiben die Möglichkeiten der TLA wie gesagt voll erhalten.

Nachdem wir nun grundsätzlich untersucht und herausgefunden hatten, auf welche Weise man die Semantik von Estelle überhaupt am besten beschreiben kann, und nachdem mit der TLA/PT ein dafür geeigneter Formalismus entworfen worden war, konnten wir daran gehen, einen Ansatz zu entwickeln, um die Semantik von Estelle zu beschreiben. Dafür wurde die Skizze einer Semantikdefinition ausgeführt.

Natürlich bedeutet eine solche Definition notwendigerweise, daß mannicht einfach die Definition aus dem vorhandenen ISO-Standard in eine „bessere“ Form bringt. Denn da mit unserem Ansatz weitergehende Ziele wie etwa die formale Verifikation unterstützt werden sollen, muß es sich um eine Neudefinition handeln. Und dies hat zur Konsequenz, daß man ein Estelle definiert, das in Details gelegentlich nicht mit dem ISO-Standard übereinstimmt. Entweder läßt der ISO-Standard aufgrund von Unschärfe in seinem umgangssprachlichen Teil mehr Interpretationsmöglichkeiten offen als die neue, formale Definition, oder es kommt gegenüber seinem formaleren Teil aufgrund der ganz anderen Darstellungsweise zu kleinen Differenzen. Unvermeidlicherweise definiert man daher letztendlich immer ein neues Estelle, nicht das bisherige ISO-Estelle. Eine formale Verifikation beider Definitionen gegeneinander scheidet allein schon daher aus, daß der ISO-Standard hierfür einen zu geringen Formalisierungsgrad besitzt. Man muß sich entscheiden, auf welche Version man für seine Zwecke am Ende zurückgreift. Und sei es, daß man das Etikett „offizielle Definition“ einer anderen Definition als jetzt verleiht, weil sie sich als leistungsfähiger erweist.

Eine Bedeutung definiert man nur für diejenigen Zeichenketten, die die Syntax einer Sprache erfüllen. Für den kontextfreien Anteil der Syntax von Estelle wurde in dieser Arbeit auf die Definition in erweiterter Backus-Naur-Form (EBNF) zurückgegriffen, so wie sie im ISO-Standard zu finden ist. Diese löste dieses Problem. Der kontextsensitive Anteil der Syntax wird im ISO-Standard nur umgangssprachlich definiert, da es sich offensichtlich um ein sehr anspruchsvolles Problem handelt. Es war nicht Gegenstand dieser Arbeit, dieses weiter zu bearbeiten.

Um die Bedeutungsfunktion zu definieren, die die Estelle-Zeichenketten auf TLA/PT-Formeln abbildet, wurde ein analoges Vorgehen vorgeschlagen zum Vorgehen bei der Wiener Beschreibungssprache VDL. Die Bedeutungsfunktion wird ausgedrückt als eine Verkettung von zwei Funktionen. Die erste löst das Zerteilungsproblem (englisch: „parsing problem“) auf der Basis der bereits vorhandenen EBNF-Grammatik, die zweite, kontextsensitive Funktion setzt den resultierenden Ableitungsbaum in TLA/PT-Formeln um, wobei sie etliche Prüfungen und Transformationen ausführt.

Bei der Darstellung von „parallel“ schaltenden Estelle-Transitionen wurde - wie auch vom ISO-Standard - aufgrund der dann (für die Verifikation) deutlich einfacheren Semantikdefinition eine „Interleaving“-Semantik gewählt, das heißt, daß in einem Zustandsübergang höchstens eine Estelle-Transition schaltet. Trotzdem bedeutet die Wahl einer derartigen Semantikdefinition nicht, daß sich praktische Einschränkungen für die mögliche Parallelität einer Implementation ergeben würden. Die Entscheidung für eine Interleaving-Semantik baute auf dem oben beschriebenen Verzicht auf eine explizite Ausdrucksmöglichkeit für Nebenläufigkeit auf, da erst das Fehlen einer besonderen Kausalitätsrelation eine einfache totale Ordnung der Aktionen ermöglichte.

Mit Estelle können auch quantitative Zeitaspekte spezifiziert werden, so daß es notwendig war, das Konzept der Zeit angemessen darzustellen. Ebenso wie im ISO-Standard wird auch hier ein globaler, unabhängiger „Zeit-Prozeß“ angenommen. „Global“ ist dieser allerdings nur insofern, als daß er die globale Zeit der Newtonschen Physik modelliert. Zugriffe sind lediglich erlaubt auf Differenzen von Zeiten, so daß diese Modellierung ebenfalls zu keinerlei Einschränkungen führt; verteilte Implementationen sind leicht möglich.

Es gibt eine von der Übersetzungsfunktion gelieferte Teilformel, die für jede Spezifikation gleich ist, sie beschreibt den „Kern von Estelle“: Das Ausführungsmodell. Um dieses auf das jeweilige Übersetzungsergebnis für eine Estelle-Spezifikation beziehen zu können, wurde folglich auch grob skizziert, wie dieses Ergebnis aufgebaut sein soll. Der Schwerpunkt der Betrachtung lag dabei auf der Beschreibung der Modulstruktur und der Estelle-Transitionen. Auch die Kommunikation wurde beleuchtet, da sie besonders Estelle-spezifisch ist.

Die Definition des Ausführungsmodells von Estelle in TLA/PT wurde vollständig ausgeführt. Hiermit wurde das ursprünglich gesetzte Ziel der Arbeit erreicht.

Die resultierende TLA/PT-Formel nimmt immerhin knapp zwei Seiten ein. Dies ist zwar wesentlich kompakter als die Beschreibung im ISO-Standard, aber jedenfalls für Zwecke der *manuellen* Verifikation zu lang. Die Ursache hierfür liegt darin, daß die Semantik von Estelle schlicht kompliziert ist. Insbesondere dadurch, daß in Estelle quantitative Zeitaspekte ausdrückbar sind, ergab sich viel zusätzlicher Aufwand. Dies ist auch nicht weiter erstaunlich, denn ganz allgemein ist das Problem der Verifikation von Spezifikationen beziehungsweise Programmen mit (Echt-)Zeitangaben noch weitgehend ungelöst. Aber auch die hierarchische Modulstruktur, und dabei besonders die Vater-Sohn-Vorrangregeln und die Parallelitätsregeln, erfordern viel Aufwand in der Beschreibung.

Estelle besitzt ein einfaches Grundkonzept, zu dem „zu“ viele weitere Konzepte hinzugefügt wurden. Um das Wörtchen „zu“ haben wir Anführungsstriche gesetzt, denn es kommt immer auf das Ziel an, das erreicht werden soll. Für die Verifikation benötigt man ein möglichst einfaches Sprachsystem. Dagegen wünscht man sich beim Spezifizieren (und erst recht beim Implementieren) eine mächtige und effiziente Sprache. Es kommt bei dem Entwurf beziehungsweise bei der Wahl einer Sprache also immer darauf an, das für die jeweilige Aufgabe richtige Maß an Ausdrucksfähigkeit zu finden. Wie allein schon der Aufwand für die formale Beschreibung der

Estelle-Transitionen zeigt, ist Estelle offensichtlich nicht für die Zwecke der formalen Verifikation entworfen worden. Um so erfreulicher ist es, daß es gelungen ist, den Definitionsaufwand hierfür immerhin in solchen Grenzen zu halten, daß eine formale Verifikation von Estelle-Spezifikationen nun grundsätzlich möglich erscheint, sobald eine mechanische Unterstützung die Handhabung der Formeln übernimmt und damit erleichtert.

Und damit kommen wir zu dem Ausblick, was über das Ziel dieser Arbeit hinaus noch erreicht werden kann, und was dafür zu leisten wäre.

9. Ausblick

Das gesteckte Ziel dieser Arbeit, die Definition des Estelle-Ausführungsmodells, wurde erreicht, und noch vieles mehr, wie wir gesehen haben. Aber in dem großen Gebiet der formalen Definition der Semantik von Estelle und dann auch des Einsatzes dieser formalen Semantik bleibt noch vieles zu erforschen und zu entwickeln.

Um den in dieser Arbeit entworfenen Ansatz für eine formale Definition der Semantik von Estelle ausarbeiten zu können, muß zuerst einmal das Kalkül der TLA/PT vollständig ausformuliert werden. Dazu muß das System der Schlußregeln vervollständigt werden, einschließlich eines Beweises der Vollständigkeit und Korrektheit. Auch die Definition einer Logik, um über Aktionen zu schlußfolgern, muß noch durchgeführt werden, da diese in den TLA-Papieren bisher ausgelassen ist. (Sie ergibt sich aber leicht aus der gewöhnlichen Prädikatenlogik, nur müssen zum Beispiel x und x' als verschiedene Variablen behandelt werden.)

Weiterhin dürfte es sehr sinnvoll sein, Lamports Schritt zur TLA^+ zu folgen und eine „TLA^+/PT“ zu konstruieren, sobald die TLA^+ fertiggestellt ist. Damit erhält man wichtige syntaktische Unterstützung, um mit großen Formeln umgehen zu können (siehe dazu Kapitel 5.4).

Eine weitere Arbeit, die vor der eigentlichen formalen Definition der Semantik von Estelle geleistet werden muß, ist eine formale Definition der vollständigen Syntax, also auch einschließlich der kontextsensitiven Syntax. Diese ist bisher auch im ISO-Standard nur umgangssprachlich beschrieben.

Dann kann die Ausformulierung der Übersetzungsfunktion selbst, welche Zeichenketten in TLA/PT-Formeln übersetzt, in Angriff genommen werden. Dies sollte auf der Grundlage des Konzeptes geschehen, das in Kapitel 6 ausgeführt wurde. Die dort eingeführte Übersetzungsfunktion muß formal definiert werden, und zwar für alle syntaktischen Einheiten, die Estelle zu bieten hat. (In Kapitel 6 wurden nur die wichtigsten erwähnt.) Ein Teil dieser Aufgabe ist zum Beispiel, daß Prädikatentransformatoren zu allen Pascal-artigen Anweisungen, welche in den Transitionsrümpfen möglich sind, definiert werden müssen. Wie man sieht, ist hier noch viel und teilweise auch recht schwierige Arbeit zu leisten. Denn bisher hat noch kein Autor für Pascal eine vollständige Semantik auf der Basis von Prädikatentransformatoren angegeben, es existiert lediglich eine auf der Basis von Hoareschen Tripeln. (Man kann sich dabei allerdings wenigstens zunutze machen, daß Prädikatentransformatoren eine Weiterentwicklung der Hoareschen Tripel sind.)

Sobald man schließlich eine formale Definition der Semantik von Estelle nach dem Ansatz dieser Arbeit besitzt, kann man darangehen, sie für die formale Verifikation von Estelle-Spezifikationen zu nutzen und deren Eigenschaften formal zu analysieren. Man kann dann formal nachweisen, daß die Estelle-Spezifikation eines Systems eine Spezifikation erfüllt, die auf einer noch höheren Abstraktionsebene formuliert ist, zum Beispiel in TLA/PT. (Und es steht natürlich grundsätzlich auch die Möglichkeit offen nachzuweisen, daß umgekehrt die Estelle-Spezifikation von einem Programm erfüllt wird.)

Damit eine Verifikation von Estelle-Spezifikationen praktisch durchführbar ist, ist eine mechanische Unterstützung notwendig, wie wir bereits gegen Ende des vorigen Kapitels gesehen haben. Aber auch unabhängig von Estelle ist eine rein manuelle Programmverifikation immer problematisch. Bei echten Problemen sind die Formeln recht groß, es ist viel eintönige Fleißarbeit zu leisten, und mit beidem verbunden besteht die Gefahr von Flüchtigkeits- und Umformungsfehlern. Gerade der eintönige Anteil der Arbeit ist wie geschaffen zur Mechanisierung. Sehr leicht maschinell zu

unterstützen sind zum Beispiel die Handhabung der Formeln und die Buchführung über die bisherigen Schritte.

Es ist zwar oft schwierig, einen Beweis und die dafür notwendigen Zwischenschritte zu finden, aber es ist meist leicht, einen Beweis nachzuprüfen. Diese Aufgabe kann man auf jeden Fall mechanisieren.

Zur Zeit strebt offenbar auch Lamport mit seiner TLA beziehungsweise TLA$^+$ in die Richtung einer derartigen mechanischen Unterstützung von Beweisen. In Kapitel 10.1 von [Lam91], wiedergegeben gegen Ende von Kapitel 3.3 dieser Arbeit, wird von ersten Experimenten mit mechanischen Beweisern für die TLA berichtet. Der dort verwendete „Larch Prover“ leistet im wesentlichen genau die eben beschriebene Unterstützung bei der Handhabung und Nachprüfung, sehr kleine Schritte kann er auch selbst beweisen.

Bei einem Fortschritt in den heuristischen Fähigkeiten der Werkzeuge ist damit zu rechnen, daß schließlich auch größere Schritte selbsttätig gefunden werden, so daß am Ende nur das Finden der strategischen Beweisideen bei dem Benutzer verbleibt.

Und spätestens damit dürfte der zeitliche Aufwand für eine formale Verifikation in eine Größenordnung absinken, die einen praktischen Einsatz für etliche reale Probleme sinnvoll erscheinen läßt.

Die praktische Nutzbarkeit der formalen Verifikation wiederum ist das erhoffte Ziel der formalen Definition der Semantik, die mit unserer Arbeit begonnen wurde. Die formale Verifikation steigert das Vertrauen dahinein entscheidend, daß das realisierte System den Anforderungen der Nutzer entspricht. Denn wenn überhaupt, dann lassen sich diese Anforderungen am ehesten mit einer eigenschaftsorientierten, sehr abstrakten Spezifikation auf hoher Ebene formal einfangen. Da für eine Realisierung aber eine Darstellung notwendig ist, die nicht nur Aussagen über das „Was“, sondern auch über das „Wie“ macht, ist eine Verbindung zwischen beiden notwendig, und diese kann nur von einer Verifikation sicher geleistet werden.

Selbstverständlich kann auch ein mathematischer Beweis der Korrektheit Fehler enthalten. Indem seine Führung aber formalisiert wird, sinkt die Anzahl der möglichen Fehlerquellen sehr stark, und so kann man auch in einen langen Beweis noch gutes Vertrauen haben.

Eine Verifikation bringt auch noch weiteren Nutzen. So ergibt sich durch sie ein viel tieferes Verständnis für die Eigenschaften und Probleme des untersuchten Systems. In [Bre90] zeigte beispielsweise die Verifikation des „InRes-Protokolls“ gegen dessen Dienstspezifikation, daß sich einige wichtige Eigenschaften nicht wie bis dahin angenommen aus der Dienstspezifikation herleiten ließen. Und ein in der Verifikation erfahrener Systementwickler wird, um die Qualität seiner Arbeit zu verbessern, auch dann möglichst verständliche, lesbare und damit leichter verifizierbare Spezifikationen schreiben, wenn er deren Verifikation anschließend nicht oder nur teilweise durchführen wird. Denn Schwierigkeiten bei der Verifikation sind häufig ein Zeichen dafür, daß man das Untersuchte nicht wirklich verstanden hat. Gerade an diesen Punkten aber sind oft Fehler verborgen.

Ein vollständiges System besteht nicht nur aus einem verifizierten Programm, sondern es ist zum Beispiel auch ein Übersetzer, ein Betriebssystem und - sehr wichtig - Hardware erforderlich. Es sollte unmittelbar einleuchten, daß dieses alles fehlerhaft sein oder werden kann. Auch hier sind Maßnahmen zur Erhöhung der Zuverlässigkeit notwendig; und auch an diesen Problemen wird intensiv gearbeitet. Trotzdem ist die Anwender-Software zur Zeit eine sehr bedeutende Quelle von Fehlern, und eine Verbesserung an dieser Stelle bringt eine erhebliche Steigerung der Zuverlässigkeit.

Unabhängig von allen - immer endlichen - Steigerungen der Zuverlässigkeit wird der Entwickler allerdings nie der Verantwortung enthoben zu prüfen, welche Konse-

quenzen ein Versagen des Systems haben würde, ob also ein Einsatz verantwortbar ist.

10. Literaturverzeichnis

[AbLa91] Abadi, M., Lamport, L.: *The Existence of Refinement Mappings*; in: Theoretical Computer Science, Vol. 82(2), Mai 1991, S. 253-284

[AmPrSc88] Amer, P., Pridor, A., Schmidt, J.: *Expansion of Transitions in Estelle Formal Specifications*; in: Protocol Specification, Testing and Verification VIII, 1988, S. 159-170

[AnBrHi91] Andrae, C., Bredereke, J., Hille, C., Peter, D., Reimer, T., Schüler, U., Gotzhein, R., Vogt, F. H.: *Praktischer Einsatz und Weiterentwicklung von Estelle*; Universität Hamburg, Fachbereich Informatik, Bericht Nr. 150, 1991, 22 S. /
Auch in: Encarnacao, J. (Hrsg.): Telekommunikation und multimediale Anwendungen der Informatik; 21. Jahrestagung der Gesellschaft für Informatik, Darmstadt, 14.-18.10.1991, Springer Verlag, 1991 /
Auch in: DIN-Mitteilungen 70, Mai 1991, S. 288-295

[Apt81] Apt, K. R.: *Ten Years of Hoare's Logic - A Survey*; in: ACM Trans. on Programming Languages and Systems, Vol. 3, No. 4, 1981, S. 431-483

[BaBe89] Baeten, J. C. M., Bergstra, J. A.: *Design of a Specification Language by Abstract Syntax Engineering*; Centre for Math. and Comp. Sc., Amsterdam, Report CS-R8934, September 1989, 28 S.

[Bak77] de Bakker, J. W.: *Semantics of Infinite Processes Using Generalized Trees*; Mathematical Centre, Amsterdam, Report IW 82/77 MEI, 1977, 7 S.

[Bak89] de Bakker, J. W.: *Designing concurrency semantics*; Free University of Amsterdam, Centre for Mathematics and Computer Science, Report CS-R8902, März 1989, 15 S.

[Bar85] Barringer, H.: *A Survey of Verification Techniques for Parallel Programs*; Lecture Notes in Comp. Sc. 191, Springer-Verlag, Berlin Heidelberg, 1985, 115 S.

[BaRu89] de Bakker, J. W., Rutten, J. J. M. M.: *Concurrency semantics based on metric domain equations*; Free University of Amsterdam, Centre for Mathematics and Computer Science, Report CS-R8954, Dezember 1989, 26 S.

[BoCa86] Boudol, G., Castellani, I.: *On the Semantics of Concurrency: Partial Orders and Transition Systems*; INRIA Sophia-Antipolis, Valbonne, Juli 1986, 16 S.

[BoCa88] Boudol, G., Castellani, I.: *A Non-Interleaving Semantics for CCS Based on Proved Transitions*; INRIA Sophia-Antipolis, Valbonne, Oktober 1988, 13 S.

[Bra91] Brauer, W.: *Vortrag während des Festkolloquiums zum 20. Geburtstag der Informatik an der Universität Hamburg und zum 60. Geburtstag von Professor Flessner*; private Mitschrift, 11. Januar 1991

[Bre90] Bredereke, J.: *Spezifikation und Implementation des InRes-Protokolls unter Verwendung von Estelle und temporaler Logik*; Studienarbeit Nr. 700, Universität Hamburg, Fachbereich Informatik, Dezember 1990, 153 S.

[BrNe89] Broy, M., Nelson, G.: *Can Fair Choice Be Added to Dijkstra's calculus?*; Universität Passau, Fak. f. Math. u. Inform., Report MIP-8902, 1989, S. 1-7

[Bro87] Brown, F. M. (Hrsg.): *The Frame Problem In Artificial Intelligence*; Proc. of the 1987 Workshop, Morgan Kaufmann Pub. Inc., Lawrence, Kansas, 12.-15. April 1987, 357 S.

[BuDe87] Budkowski, S., Dembinski, P.: *An Introduction to Estelle: A Specification Language for Distributed Systems*; in: Computer Networks and ISDN Systems 14, 1987, S. 3-23

[CaFrMo82] Castellani, I., Franceschi, P., Montanari, U.: *Labeled Event Structures: a Model for Observable Concurrency*; in: Bjørner, D. (Hrsg.): Formal Description of Programming Concepts - II, North-Holland, Amsterdam, 1.-4. Juni 1982, S. 383-399

[DaDe90] Darondeau, Ph., Degano, P.: *Causal Trees: Interleaving + Causality*; in: Guessarian, I. (Hrsg.): Semantics of Systems of Concurrent Processes, Lecture Notes in Comp. Sc. 469, Springer Verlag, Berlin Heidelberg, April 1990, S. 239-255

[Dij76] Dijkstra, E. W.: *A Discipline of Programming*; Prentice-Hall, Englewood Cliffs, 1976, 217 S.

[DiSc90] Dijkstra, E. W., Scholten, C. S.: *Predicate Calculus and Program Semantics*; Springer Verlag, New York, 1990, 220 S.

[Don76] Donahue, J. E.: *Complementary Definitions of Programming Language Semantics*; Springer Verlag, Berlin, Heidelberg, 1976, 172 S.

[Eng88] Engesser, H. (Hrsg.): *Duden „Informatik“*; Dudenverlag, Mannheim, 1988, 671 S.

[Fai72] Fairley, R.: *The Formal Definition of a Parameter Passing Language*; University of Colorado, Dept. of Comp. Sc., Boulder, Report #CU-CS-010-72, Dezember 1972, 48 S.

[GaGu89] Garland, S. J., Guttag, J. V.: *An Overview of LP, The Larch Prover*; in: Dershowitz, N. (Hrsg.): Proc. 3rd Intl. Conf. Rewriting Techniques and Applications; Chapel Hill, Lecture Notes on Comp. Sce., vol. 355, Springer, April 1989, S. 137-151

[Gai88] Gaifman, H.: *Modeling Concurrency by Partial Orders And Nonlinear Transition Systems*; in: de Bakker, J. W., de Roever, W.-P., Rozenberg, G. (Hrsg.): Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Lecture Notes in Comp. Sc. 354, Springer-Verlag, Berlin Heidelberg, 30. Mai - 3. Juni 1988, S. 467-488

[Gau91] Gaudel, M.-C.: *Advantages and Limits of Formal Approaches for Ultra-High Dependability*; in: Proc. of the 6th Int. Workshop on Software Specification and Design, Como, Italien, IEEE Computer Society Press, Los Alamitos, 25.-26. Oktober 1991, S. 237-241

[Ger80] German, S. M.: *An Extended Semantic Definition of Pascal For Proving the Absence of Common Runtime Errors*; Standford University, Comp. Sc. Dept.,

Report No. STAN-CS-80-811, Juni 1980, 58 S.

[GhJa89] Ghezzi, C., Jazayeri, M.: *Konzepte der Programmiersprachen*; Oldenbourg-Verlag, München, 1989, 602 S.

[Got85] Gotzhein, R.: *Modellierung und Spezifikation von Diensten und Verhalten in verteilten Systemen*; Dissertation, Universität Erlangen, Technische Fakultät, 1985, 280 S.

[Got92] Gotzhein, R.: *Temporal Logic and Applications - A Tutorial*; in: Computer Networks and ISDN Systems 24, 1992 (in Druck)

[Got92a] Gotzhein, R.: *Formal Definition and Representation of Interaction Points*; in: Computer Networks and ISDN Systems 25, 1992 (in Druck)

[GuHo83] Guttag, J. V., Horning J. J.: *Preliminary Report On The Larch Shared Language*; Lab. for Comp. Sce., Mass. Inst. of Technol., Cambridge (Mass.), Report MIT/ICS/TR-307, Oktober 1983, 55 S.

[Har88] Harel, D.: *On Visual Formalisms*; in: Communications of the ACM, Volume 31, Number 5, Mai 1988, S. 514-530

[Har89] Harter, P. K. Jr.: *Writing Formal Specifications Using Transition Axioms*; interner DEC-Report, 12. Oktober 1989, 65 S.

[Hoa69] Hoare, C. A. R.: *An Axiomatic Basis for Computer Programming*; in: Communications of the ACM, Vol. 12, Oktober 1969, S. 576-583

[Hoa73] Hoare, C. A. R.: *Parallel Programming: An Axiomatic Approach*; Stanford University, Comp. Sc. Dept., Report STAN-CS-73-394, Oktober 1973, 33 S.

[Hoa85] Hoare, C. A. R.: *Programs are Predicates*; in: Mathematical Logic and Programming Languages; Hoare, C. A. R., Shepherdson, J. C. (Hrsg.), Prentice-Hall, Englewood Cliffs, New Jersey, 1985, S. 141-155

[HoHe85] Hoare, C. A. R., He, Jifeng: *The Weakest Prespecification*; Oxford University Computing Lab., Technical Monograph PRG-44, Juni 1985, 60 S.

[HoRaRo90] Hooman, J. J. M., Ramesh, S., de Roever, W. P.: *A Compositional Axiomatisation of Safety and Liveness Properties for Statecharts*; in: Semantics for Concurrency, Leicester 1990, proceedings; Kwiatkowska, M. Z., Shields, M. W., Thomas, R. M. (Hrsg.), Springer Verlag, London, 23.-25. Juli 1990, S. 242-261

[HoWi73] Hoare, C. A. R., Wirth, N.: *An Axiomatic Definition of the Programming Language PASCAL*; in: Acta Informatica 2, Springer Verlag, 1973, S. 335-355

[HuCr78] Hughes, G. F., Cresswell, M. J.: *Einführung in die Modallogik*; deGruyter, Berlin, 1978, 340 S., Kapitel 1-6

[ISO81] ISO/TC 97/SC 16: *Data Processing - Open Systems Interconnection - Basic Reference Model*; Computer Networks 5, 1981, S. 81-118

[ISO86] ISO/TC 97/SC 21: *Estelle - A Formal Description Technique Based on an Extended State Transition Model*; ISO/TC 97/SC 21 N xxxx DP 9074, 4.9.1986

[ISO89] ISO/IEC JTC 1/SC 21: *Estelle - A Formal Description Technique Based on an Extended State Transition Model*; ISO/IEC JTC 1/SC 21 N 3611 DIS 9074, Mai 1989

[ISO89a] ISO IS 9074, *Estelle - A Formal Description Technique Based on an Extended State Transition Model*; 1989, 179 S.

[JaEm77] Janssen, T. M. V., van Emde Boas, P.: *On the Proper Treatment of Referencing, Dereferencing and Assignment*; in: Goos, G., Hartmanis, J. (Hrsg.): Automata, Languages and Programming, 4th Colloq., Univ. of Turku, Lecture Notes in Comp. Sce., vol. 52, Springer, Berlin, Juli 1977, S. 282-300

[Jür91] Jürgensen, W.: *Fehlerursachenlokalisierung in Kommunikationsprotokollen*; Dissertation, Universität Karlsruhe, Dezember 1991, 177 S.

[KaPe88] Katz, S., Peled, D.: *An Efficient Verification Method For Parallel and Distributed Programs*; in: de Bakker, J. W., de Roever, W.-P., Rozenberg, G. (Hrsg.): Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Lecture Notes in Comp. Sc. 354, Springer-Verlag, Berlin Heidelberg, 30. Mai - 3. Juni 1988, S. 489-507

[KiSøWo88] King, S., Sørensen, I. H., Woodcock, J.: *Z: Grammar and Concrete and Abstract Syntaxes*; Version 2.0, Technical Monograph PRG-68, Oxford Univ. Comp. Lab., Prog. Research Group, Oxford, Juli 1988, 48 S.

[LaCa75] Lauer, P. E., Campbell, R. H.: *Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes*; University of Newcastle Upon Tyne, Comp. Lab., Tech. Report Series, no. 74, April 1975, 43 S.

[Lam80] Lamersdorf, W.: *Semantikspezifikationstechniken und eine Anwendung auf die semantische Definition von Pascal/R*; Diplomarbeit, Uni Hamburg, FB Informatik, Juni 1980, 187 S.

[Lam90] Lamport, L.: *win and sin - Predicate Transformers for Concurrency*; in: ACM Trans. on Programming Languages and Systems, Vol. 12, No. 3, Juli 1990, S. 396-428

[Lam91] Lamport, L.: *The Temporal Logic of Actions*; Report No. 79, Digital Equipment Corp., Palo Alto, California, USA, Dezember 1991, 73 S.

[Lam91a] Lamport, L.: diverse private Korrespondenz

[Lam91b] Lamport, L.: *Introducing TLA*$^+$; Digital Equipment Corp., Preliminary Draft, (in Vorbereitung), rev. 2. September 1991, 41 S.

[LaShBe79] Lauer, P. E., Shields, M. W., Best, E.: *Formal Theory of the Basic COSY Notation*; University of Newcastle Upon Tyne, Comp. Lab., Tech. Report Series, no. 143, November 1979, 130 S.

[LaTo78] Lauer, P. E., Torrigiani, P. R.: *Toward a System Specification Language Based on Paths and Processes*; University of Newcastle Upon Tyne, Comp. Lab., Tech. Report Series, no. 120, Februar 1978, 57 S.

[LaToSh79] Lauer, P. E., Torrigiani, P. R., Shields, M. W.: *COSY: A System Specification Language Based on Paths and Processes*; University of Newcastle Upon Tyne, Comp. Lab., Tech. Report Series, no. 136, April 1979, 73 S.

[Lau68] Lauer, P. E.: *Formal Definition of Algol 60*; IBM Lab. Vienna, Tech. Report TR 25.088, 13. Dezember 1968, 62 S.

[Lau76] Lauer, P. E.: *Abstract Tree Processors with Networks of State Machines as Control: Their use in Programming Language Definition*; University of Newcastle Upon Tyne, Comp. Lab., Tech. Report Series, no. 87, März 1976, 35 S.

[Lau81] Laut, A.: *Von abstrakter Syntax zu verketteten Bäumen - Entwicklung einer Datenstruktur für die Programm-Manipulation*; Techn. Univ. München, Inst. f. Inf., Bericht TUM-I8114, München, November 1981, 108 S.

[LuAlBa68] Lucas, P., Alber, K., Bandat, K., Bekic, H., Oliva, P., Walk, K., Zeisel, G.: *Informal Introduction to the Abstract Syntax and Interpretation of PL/I*; IBM Lab. Vienna, Technical Report TR 25.083, 28. Juni 1968, 192 S.

[Luc87] Lucas, P.: *VDM: Origins, Hopes, and Achievements*; IBM Watson Research Center, Dept. Comp. Sce., Yorktown Heights, Report RJ 5470 (55869), 19. Januar 1987, 19 S.

[LuLaSt70] Lucas, P., Lauer, P., Stigleitner, H.: *Method and Notation for the Formal Definition of Programming Languages*; IBM Lab. Vienna, Tech. Report TR 25.087, 28 June 1868, revised: 1. Juli 1970, 108 S.

[LuWa69] Lucas, P., Walk, K.: *On the Formal Description of PL/I*; in: Ann. Review in Automatic Programming; Vol. 6, Pergamon Press, Oxford, 1969, S. 105-182

[MaNe87] Mackert, L. F., Neumeier-Mackert, I. B.: *Communicating Rule Systems*; Technical Report No. 43.8705, IBM European Networking Center, 1987

[Mey90] Meyers Lexikonred. (Chefred.: Digel, W., Kwiatkowski, G.): *Meyers Großes Taschenlexikon*; 24 Bd., BI-Taschenbuch-Verlag, Mannheim, 1990

[Mon73] Montague, R.: *The Proper Treatment of Quantification in Ordinary English*; in: Hintikka, J., Moravcsik, J., Suppes, P. (Hrsg.): Approaches to Natural Language, Reidel, Dordrecht, 1973, nachgedruckt in: Thomason, R. H.: Formal Philosophy, Selected Papers of Richard Montague, Yale University Press, New Haven and London, 1974, S. 247-270

[NaKa90] Naik, K., Sarikaya, B.: *Chart Semantics for Estelle*; Concordia University Montreal, Dpt. of Electrical Engineering, unveröffentlichter technischer Report, 1990, 15 S.

[NBS87] NBS Institute for Computer Science and Technology: *User Guide for the NBS Prototype Compiler for Estelle*; Rep. No. ICST/SNA 87-3; Oktober 1987

[NBS87a] NBS Institute for Computer Science and Technology: *User Guide for the NBS Prototype Compiler for Estelle*; Rep. No. ICST/SNA 87-4; September 1987

[NiRoTh90] Nielsen, M., Rozenberg, G., Thiagarajan, P. S.: *Elementary Transition Systems*; Aarhus University, Comp. Sce. Dept., Report DAIMI PB-310, April 1990, 41 S.

[Old87] Olderog, E.-R.: *Operational Petri Net Semantics for CCSP*; Christian-Albrechts-Universität Kiel, Institut für Informatik und praktische Mathematik, Bericht Nr. 8704, April 1987, 30 S.

[Par83] Parnas, D. L.: *A Generalized Control Structure and Its Formal Definition*; in: Communications of the ACM, Vol. 26, No. 8, August 1983, S. 572-581

[Par92] Parnas, D. L.: *Predicate Logic for Software Engineering*; (in Vorbereitung), Februar 1992, 8 S.

[Pet89] Peter, D.: *Konzeption eines Environments zur Generierung und verteilten Ausführung von Estelle-Code*; Studienarbeit Nr. 379, Universität Hamburg, Fachbereich Informatik, 1989

[Pet91] Peter, D.: *Entwurf, Realisierung und Integration eines Protokolls zur verteilten Ausführung von Estelle-Spezifikationen*; Diplomarbeit Nr. 791, Universität Hamburg, Fachbereich Informatik, 1991

[Plo82] Plotkin, G. D.: *An Operational Semantics for CSP*; University of Edinburgh, Dpt. of Computer Science, Internal Report CSR-114-82, Mai 1982

[Pnu86] Pnueli, A.: *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends*; in: Goos, G., Hartmanis, J. (Hrsg.), Current Trends in Concurrency, Lecture Notes in Comp. Sc. 224, Springer-Verlag, Berlin Heidelberg, 1986, S. 510-584

[Pra73] Pratt, T. W.: *A Formal Definition of Algol 60 Using Hierarchical Graphs and Pair Grammars*; University of Texas at Austin, Comp. Center, Dept. of Comp. Sc., Report UTEX-CC-TSN-33, Januar 1973, 82 S.

[Pri91] Prinoth, R.: *Beschreibungsmittel und Konzepte zur Realisierung verteilter Systeme - Überlegungen anhand von Produktnetzen*; GMD-Studien Nr. 192, Sankt Augustin, Mai 1991, 77 S.

[Sch86] Schneider, H. J.: *Programmverifikation*; Computer Magazin 4'86, 1986, S. 50-57

[ScMeVo83] Schwartz, R. L., Melliar-Smith, P. M., Vogt, F. H.: *Interval Logic: A Higher-Level Temporal Logic for Protocol Specification*; in: H. Rudin, C. H. West (Hrsg.), Protocol Specification, Testing, and Verification, III, North-Holland, Amsterdam, 1983, S. 3-18

[Sta75] Staunstrup, J.: *Platon. Axiomatic Definition*; University of Aarhus, RECAU, Report RECAU 75-61, August 1975, 24 S.

[TeCa85] Terwilliger, R. B., Campbell, R. H.: *PLEASE: Predicate Logic based Executable Specifications*; University of Illinois at Urbana-Champaign, Dpt. of Comp. Sce., Report UIUCDCS-R-85-1231, Oktober 1985, 22 S.

[Tof90] Tofts, C.: *Timed Concurrent Processes*; in: Semantics for Concurrency, Leicester 1990, proceedings; Kwiatkowska, M. Z., Shields, M. W., Thomas, R. M. (Hrsg.), Springer Verlag, London, 23.-25. Juli 1990, S. 281-294

[Vog87] Vogt, F.: *Entwicklung verteilter Systeme II: Estelle*; Uni Hamburg, Vorlesungsskript, Sommersemester 1987

[Wag86] Wagner, E. G.: *A Categorical Treatment of Pre- and Post-Conditions*; IBM Watson Research Center, Math. Sc. Dpt., Yorktown Heights, Report RC 12181 (#54757), 24. September 1986, 26 S.

[Win88] Winskel, G.: *An Introduction to Event Structures*; in: de Bakker, J. W., de Roever, W.-P., Rozenberg, G. (Hrsg.): Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Lecture Notes in Comp. Sc. 354, Springer-Verlag, Berlin Heidelberg, 30. Mai - 3. Juni 1988, S. 364-397

[ZöHo88] Zöbel, D., Hogenkamp, H.: *Konzepte der parallelen Programmierung*; Teubner, Stuttgart, 1988, 235 S.

[Zwi89] Zwiers, J.: *Compositionality, Concurrency and Partial Correctness*; Springer Verlag, Berlin, Heidelberg, 1989, 272 S.

[ZwRo89] Zwiers, J., de Roever, W.-P.: *Predicates are Predicate Transformers: A Unified Compositional Theory for Concurrency*; in: Proc. of the 8th ACM Symp. on Princ. of Distributed Computing, Edmonton, 14.-16. August 1989, S. 265-279

11. Abbildungsverzeichnis

Abbildung 2-1: Ein einfaches Schreiber-Leser-System 10–11

Abbildung 3-1: Syntax und Semantik der einfachen TLA 21

Abbildung 3-2: Axiome und Schlußregeln der einfachen TLA 22

Abbildung 3-3: Quantifikation in der TLA 23

Abbildung 3-4: Veranschaulichung von `wp` und `wlp` 31

Abbildung 3-5: Veranschaulichung von `sp` 32

Abbildung 4-1: Die allgemeine **`do..od`**-Schleife der sequentiellen Programmierung 42

Abbildung 5-1: Vollständige Syntax der TLA/PT 71

Abbildung 5-2: Die abgeleitete Schlußregel `INV3` 82

Abbildung 6-1: Ein Demonstrationsbeispiel in Estelle für die Definition der Semantik 89–90

Anhang

Syntax der TLA/PT

<allgemeine Formel> $\triangleq$
<Formel> | ($\exists$*<Variable>*:: *<allgemeine Formel>*)
| ($\exists$*<starre Variable>*:: *<allgemeine Formel>*)
| *<allgemeine Formel>* $\wedge$ *<allgemeine Formel>*
| $\neg$*<allgemeine Formel>*

<Formel> $\triangleq$ *<Prädikat>* | $\Box$[*<Aktion>*]$_{\texttt{<Zustandsfunktion>}}$ | $\neg$*<Formel>*
| *<Formel>* $\wedge$ *<Formel>* | $\Box$ *<Formel>*

<Aktion> $\triangleq$ boolwertiger Ausdruck, der enthalten kann:
Konstanten, Variablen, gestrichene Variablen,
„*<Prädikatentransformator>*.*<einfaches Prädikat>*“
und „($\exists$*<Variable>*:: *<Aktion>*)“

<Prädikat> $\triangleq$ *<einfaches Prädikat>* | *Enabled <Aktion>*

<einfaches Prädikat> $\triangleq$ bool-wertige *<Zustandsfunktion>*

<Zustandsfunktion> $\triangleq$
Ausdruck, der enthalten kann:
Konstanten, Variablen,
„*<Prädikatentransformator>*.*<einfaches Prädikat>*“
und „($\exists$*<Variable>*:: *<Zustandsfunktion>*)“

<Prädikatentransformator> $\triangleq$
ein Prädikatentransformator wie bei Dijkstra

Semantik der TLA/PT

$\mathtt{s}[\![\mathtt{f}]\!] \triangleq (\mathtt{pt_eval(f)})(\forall\text{"v"}\colon \mathtt{s}[\![\mathtt{v}]\!]/\mathtt{v})$

$\mathtt{s}[\![A]\!]\,\mathtt{t} \triangleq (\mathtt{pt_eval}(A))(\forall\text{"v"}\colon \mathtt{s}[\![\mathtt{v}]\!]/\mathtt{v},\ \mathtt{t}[\![\mathtt{v}]\!]/\mathtt{v}')$

$\models A \triangleq \forall \mathtt{s},\mathtt{t} \in \mathbf{St}\colon \models \mathtt{s}[\![A]\!]\,\mathtt{t}$

$\sigma[\![\mathtt{F} \wedge \mathtt{G}]\!] \triangleq \sigma[\![\mathtt{F}]\!] \wedge \sigma[\![\mathtt{G}]\!]$

$\sigma[\![\neg\mathtt{F}]\!] \triangleq \neg\sigma[\![\mathtt{F}]\!]$

$\models \mathtt{F} \triangleq \forall\sigma \in \mathbf{St}^{\infty}\colon \models \sigma[\![\mathtt{F}]\!]$

$\mathtt{s}[\![\mathit{Enabled}\ A]\!] \triangleq \exists \mathtt{t} \in \mathbf{St}\colon \mathtt{s}[\![A]\!]\,\mathtt{t}$

$\ll \mathtt{s}_0,\mathtt{s}_1,\ldots \gg [\![\Box\mathtt{F}]\!] \triangleq \forall \mathtt{n} \in \mathbf{Nat}\colon \ll \mathtt{s}_n,\mathtt{s}_{n+1},\ldots \gg [\![\mathtt{F}]\!]$

$\ll \mathtt{s}_0,\mathtt{s}_1,\ldots \gg [\![A]\!] \triangleq \mathtt{s}_0[\![A]\!]\,\mathtt{s}_1$

$\ll \mathtt{s}_0,\mathtt{s}_1,\ldots \gg =_{\mathtt{x}} \ll \mathtt{t}_0,\mathtt{t}_1,\ldots \gg \triangleq \forall \mathtt{n} \in \mathbf{Nat}\colon \forall\text{"v"} \neq \text{"x"}\colon \mathtt{s}_n[\![\mathtt{v}]\!] = \mathtt{t}_n[\![\mathtt{v}]\!]$

$\natural \ll \mathtt{s}_0,\mathtt{s}_1,\mathtt{s}_2,\ldots \gg \triangleq$ **if** $\forall \mathtt{n} \in \mathbf{Nat}\colon \mathtt{s}_n = \mathtt{s}_0$
then $\ll \mathtt{s}_0,\mathtt{s}_0,\mathtt{s}_0,\ldots \gg$
else if $\mathtt{s}_1 = \mathtt{s}_0$ **then** $\natural \ll \mathtt{s}_1,\mathtt{s}_2,\mathtt{s}_3,\ldots \gg$
else $\ll \mathtt{s}_0 \gg \circ \natural \ll \mathtt{s}_1,\mathtt{s}_2,\ldots \gg$

$\sigma[\![\exists \mathtt{x}\colon \mathtt{F}]\!] \triangleq \exists \rho,\tau \in \mathbf{St}^{\infty}\colon (\natural\sigma = \natural\rho) \wedge (\rho =_{\mathtt{x}} \tau) \wedge \tau[\![\mathtt{F}]\!]$

$\sigma[\![\exists \mathtt{c}\colon \mathtt{F}]\!] \triangleq \exists \mathtt{c} \in \mathbf{Val}\colon \sigma[\![\mathtt{F}]\!]$

$\mathtt{s}[\![(\exists \mathtt{x}\colon\colon \mathtt{f})]\!] \triangleq (\exists\alpha\colon \alpha \in \mathbf{St} \wedge \alpha =_{\mathtt{x}} \mathtt{s} \wedge \alpha[\![\mathtt{f}]\!])$

$\mathtt{s}[\![(\exists \mathtt{x}\colon\colon A)]\!]\,\mathtt{t} \triangleq (\exists\alpha,\beta\colon \alpha,\beta \in \mathbf{St} \wedge \alpha =_{\mathtt{x}} \mathtt{s} \wedge \beta =_{\mathtt{x}} \mathtt{t} \wedge \alpha[\![A]\!]\,\beta)$

Fortsetzung nächste Seite

zusätzliche Notation

$f' \triangleq f(\forall "v": v'/v)$

$[A]_f \triangleq A \vee (f' = f)$

$<A>_f \triangleq A \wedge (f' \neq f)$

$Unchanged\ f \triangleq f' = f$

$\Diamond F \triangleq \neg\Box\neg F$

$F \rightsquigarrow G \triangleq \Box(F \Rightarrow \Diamond G)$

$WF_f(A) \triangleq \Box\Diamond<A>_f \vee \Box\Diamond\neg Enabled\ <A>_f$

$SF_f(A) \triangleq \Box\Diamond<A>_f \vee \Diamond\Box\neg Enabled\ <A>_f$

$PT_f \triangleq (\exists hvar:: \underline{f}'=hvar \wedge \neg PT.(\underline{f}\neq hvar))$

wobei

$[\![\ ]\!]$ die zu definierende Bedeutungsfunktion ist,
f eine <*Zustandsfunktion*> ist,
A eine <*Aktion*> ist,
F,G jeweils <*Formel*>n sind,
PT ein <*Prädikatentransformator*> ist,
$s,t,s_0,s_1,\ldots$ Zustände sind,
σ eine Zustandsfolge ist,
St die Menge aller Zustände ist,
$x,hvar$ <*Variable*>n sind,
$(\forall "v": .../v, .../v')$ die Ersetzung für alle Variablen v bezeichnet
und
`pt_eval` eine Funktion ist, die in Kapitel 5.1 erläutert wird.

$(\forall v:: F) \triangleq \neg(\exists v:: \neg F)$

$(\forall c:: F) \triangleq \neg(\exists c:: \neg F)$

$(\forall v: F: G) \triangleq (\forall v:: F \Rightarrow G)$

$(\forall c: F: G) \triangleq (\forall c:: F \Rightarrow G)$

$(\exists v: F: G) \triangleq (\exists v:: F \wedge G)$

$(\exists c: F: G) \triangleq (\exists c:: F \wedge G)$

wobei

v eine <*Variable*> ist,
c eine <*starre Variable*> ist und
F,G beide entweder <*allgemeine Formel*>n, <*Aktion*>en oder <*Zustandsfunktion*>en sind.

Die Schlußregeln der einfachen temporalen Logik

STL1. $\dfrac{\text{F ist beweisbar durch Aussagenlogik}}{\mathtt{F}}$

STL2. $\vdash \Box\mathtt{F} \Rightarrow \mathtt{F}$

STL3. $\vdash \Box\Box\mathtt{F} \equiv \Box\mathtt{F}$

STL4. $\dfrac{\mathtt{F} \Rightarrow \mathtt{G}}{\Box\mathtt{F} \Rightarrow \Box\mathtt{G}}$

STL5. $\vdash \Box(\mathtt{F} \wedge \mathtt{G}) \equiv (\Box\mathtt{F}) \wedge (\Box\mathtt{G})$

STL6. $\vdash (\Diamond\Box\mathtt{F}) \wedge (\Diamond\Box\mathtt{G}) \equiv \Diamond\Box(\mathtt{F} \wedge \mathtt{G})$

LATTICE.
$\succ$ ist eine Wohlordnung auf einer nichtleeren Menge S

$$\frac{\mathtt{F} \wedge (\mathtt{c} \in \mathtt{S}) \Rightarrow (\mathtt{H}_\mathtt{c} \rightsquigarrow (\mathtt{G} \vee (\exists \mathtt{d} \in \mathtt{S}: (\mathtt{c} \succ \mathtt{d}) \wedge \mathtt{H}_\mathtt{d})))}{\mathtt{F} \Rightarrow ((\exists \mathtt{c} \in \mathtt{S}: \mathtt{H}_\mathtt{c}) \rightsquigarrow \mathtt{G})}$$

Die grundlegenden Schlußregeln der TLA/PT

TLA1. $\vdash \Box\mathtt{P} \equiv \mathtt{P} \wedge \Box[\mathtt{P} \Rightarrow \mathtt{P}']_\mathtt{P}$

TLA2. $\dfrac{\mathtt{P} \wedge [A]_\mathtt{f} \Rightarrow \mathtt{Q} \wedge [B]_\mathtt{g}}{\Box\mathtt{P} \wedge \Box[A]_\mathtt{f} \Rightarrow \Box\mathtt{Q} \wedge \Box[B]_\mathtt{g}}$

Weitere Schlußregeln

INV1. $\dfrac{\mathtt{I} \wedge [N]_\mathtt{f} \Rightarrow \mathtt{I}'}{\mathtt{I} \wedge \Box[N]_\mathtt{f} \Rightarrow \Box\mathtt{I}}$

INV2. $\vdash \Box\mathtt{I} \Rightarrow (\Box[N]_\mathtt{f} \equiv \Box[N \wedge \mathtt{I} \wedge \mathtt{I}']_\mathtt{f})$

WF1.
$$\frac{\begin{array}{l}\mathtt{P} \wedge [N]_\mathtt{f} \Rightarrow (\mathtt{P}' \vee \mathtt{Q}') \\ \mathtt{P} \wedge \langle N \wedge A\rangle_\mathtt{f} \Rightarrow \mathtt{Q}' \\ \mathtt{P} \Rightarrow \mathit{Enabled}\ \langle A\rangle_\mathtt{f}\end{array}}{\Box[N]_\mathtt{f} \wedge \mathtt{WF}_\mathtt{f}(A) \Rightarrow (\mathtt{P} \rightsquigarrow \mathtt{Q})}$$

WF2.
$$\frac{\begin{array}{l}\langle N \wedge B\rangle_\mathtt{f} \Rightarrow \langle(:M:)\rangle_{(:\mathtt{g}:)} \\ \mathtt{P} \wedge \mathtt{P}' \wedge \langle N \wedge A\rangle_\mathtt{f} \Rightarrow B \\ \mathtt{P} \wedge (:\mathit{Enabled}\ \langle M\rangle_\mathtt{g}:) \Rightarrow \mathit{Enabled}\ \langle A\rangle_\mathtt{f} \\ \Box[N \wedge \neg B]_\mathtt{f} \wedge \mathtt{WF}_\mathtt{f}(A) \wedge \Box\mathtt{F} \Rightarrow \Diamond\Box\mathtt{P}\end{array}}{\Box[N]_\mathtt{f} \wedge \mathtt{WF}_\mathtt{f}(A) \wedge \Box\mathtt{F} \Rightarrow (:\mathtt{WF}_\mathtt{g}(M):)}$$

SF1.
$$\frac{\begin{array}{l}\mathtt{P} \wedge [N]_\mathtt{f} \Rightarrow (\mathtt{P}' \vee \mathtt{Q}') \\ \mathtt{P} \wedge \langle N \wedge A\rangle_\mathtt{f} \Rightarrow \mathtt{Q}' \\ \Box\mathtt{P} \wedge \Box[N]_\mathtt{f} \wedge \Box\mathtt{F} \Rightarrow \Diamond\mathit{Enabled}\ \langle A\rangle_\mathtt{f}\end{array}}{\Box[N]_\mathtt{f} \wedge \mathtt{SF}_\mathtt{f}(A) \wedge \Box\mathtt{F} \Rightarrow (\mathtt{P} \rightsquigarrow \mathtt{Q})}$$

SF2.
$$\frac{\begin{array}{l}\langle N \wedge B\rangle_\mathtt{f} \Rightarrow \langle(:M:)\rangle_{(:\mathtt{g}:)} \\ \mathtt{P} \wedge \mathtt{P}' \wedge \langle N \wedge A\rangle_\mathtt{f} \Rightarrow B \\ \mathtt{P} \wedge (:\mathit{Enabled}\ \langle M\rangle_\mathtt{g}:) \Rightarrow \mathit{Enabled}\ \langle A\rangle_\mathtt{f} \\ \Box[N \wedge \neg B]_\mathtt{f} \wedge \mathtt{SF}_\mathtt{f}(A) \wedge \Box\mathtt{F} \Rightarrow \Diamond\Box\mathtt{P}\end{array}}{\Box[N]_\mathtt{f} \wedge \mathtt{SF}_\mathtt{f}(A) \wedge \Box\mathtt{F} \Rightarrow (:\mathtt{SF}_\mathtt{g}(M):)}$$

wobei

$\mathtt{F},\mathtt{G},\mathtt{H}_\mathtt{c}$	TLA-Formeln sind,
A, B, N, M	Aktionen sind,
$\mathtt{P},\mathtt{Q},\mathtt{I}$	Prädikate sind und
$\mathtt{f},\mathtt{g}$	Zustandsfunktionen sind.

E1. $\vdash$ F(f/x) $\Rightarrow$ $\exists$x: F

E2.
$$\frac{\begin{array}{l}\mathtt{F} \Rightarrow \mathtt{G} \\ \text{x kommt in G nicht frei vor}\end{array}}{(\exists \mathtt{x}\colon \mathtt{F}) \Rightarrow \mathtt{G}}$$

F1. $\vdash$ F(e/c) $\Rightarrow$ $\exists$c: F

F2.
$$\frac{\begin{array}{l}\mathtt{F} \Rightarrow \mathtt{G} \\ \text{c kommt in G nicht frei vor}\end{array}}{(\exists \mathtt{c}\colon \mathtt{F}) \Rightarrow \mathtt{G}}$$

wobei

x	eine <*Variable*> ist,
f	eine <*Zustandsfunktion*> ist,
c	eine <*starre Variable*> ist,
e	ein Konstanten-Ausdruck ist,
F,G	jeweils <*allgemeine Formel*>n sind und
σ	eine Zustandsfolge ist.

INV3.

$$\frac{\begin{array}{l}(\mathtt{I} \wedge [N]_{\mathtt{f}}) \Rightarrow \mathtt{I}' \\ (\mathtt{I} \wedge \mathtt{tr1.Selected}) \Rightarrow \mathtt{PT1.((Change.\{tr1.Selected).I)} \\ \ldots \\ (\mathtt{I} \wedge \mathtt{trn.Selected}) \Rightarrow \mathtt{PTn.((Change.\{trn.Selected).I)}\end{array}}{(\mathtt{I} \wedge \varphi) \Rightarrow \Box\mathtt{I}}$$

wobei

PT1,...,PTn	Prädikatentransformatoren sind,
N	eine Aktion ist,
I	ein Prädikat ist,
tr1,...trn	Verbundvariablen (unter anderem) mit den Komponenten Id und Selected sind,
f	eine Zustandsfunktion ist,

$\mathtt{Trans} \triangleq \{\mathtt{tr1.Id},\ldots,\mathtt{trn.Id}\}$ und

$$\begin{array}{l}
\varphi \triangleq \\
\quad \Box[N]_{\mathtt{f}} \\
\quad \wedge\ \Box[\ (\mathtt{tr1.Selected} \wedge \neg\mathtt{tr1.Selected}') \\
\qquad \Rightarrow (\mathtt{PT1.(Change.\{tr1.Selected\}))}_{\mathtt{f}}]_{\mathtt{tr.Selected}} \\
\quad \wedge\ \ldots \\
\quad \Box[\ (\mathtt{trn.Selected} \wedge \neg\mathtt{trn.Selected}') \\
\qquad \Rightarrow (\mathtt{PTn.(Change.\{trn.Selected\}))}_{\mathtt{f}}]_{\mathtt{tr.Selected}} \\
\quad \wedge\ \Box[(\exists \mathtt{tr}\colon \mathtt{tr.Id} \in \mathtt{Trans}: \\
\qquad \mathtt{tr.Selected} \neq \mathtt{tr.Selected}')]_{\mathtt{f}}
\end{array}$$