Entwurf und Implementierung eines Werkzeugs zur Analyse von Feature-Interaktionen in Estelle-Spezifikationen

Joachim Thees

April 1995

Entwurf und Implementierung eines Werkzeugs zur Analyse von Feature-Interaktionen in Estelle-Spezifikationen

Joachim Thees

25. April 1995

Zusammenfassung

In dieser Arbeit wird die Entwicklung eines Werkzeugs dargestellt, mit dessen Hilfe die Analyse von Feature-Interaktionen in Intelligenten Netzwerken unterstützt wird. Es basiert auf der formalen Beschreibungstechnik Estelle, wobei durch einen speziellen Spezifikationsstil Feature-Interaktionen anhand von bestimmten Wechselwirkungen zwischen Transitionen verschiedener Features (u.a. Indeterminismus) erkannt werden können. Das Ziel ist dabei die statische Erkennung und Protokollierung dieser Wechselwirkungen sowie die Entfernung von nicht ausführbaren Transitionen zur Laufzeitoptimierung.

Dazu werden zunächst die theoretischen Möglichkeiten zur Erkennung dieser Wechselwirkungen untersucht. Danach werden anhand der Implementierung des Analysewerkzeugs die eingesetzten Methoden und Algorithmen dargestellt und schließlich der Einsatz des Werkzeugs erläutert, das auf dem Estelle-Compiler PET basiert.

Inhaltsverzeichnis

1 Einleitung **3**

1.1 Motivation 3

1.2 Konkretisierung der Aufgabenstellung 5

1.3 Technisches Umfeld 8

1.4 Gliederung 9

2 Die Theorie des Analysevorgangs **10**

2.1 Globale Analyse 11

2.2 Modullokale Analyse 12

2.3 Vergleich von Transitionsklauseln 15

2.3.1 Prioritäten von Transitionen: Die `PRIORITY`-Klausel . . . 16

2.3.2 Hauptzustandsentwicklung: Die `FROM`- und `TO`-Klauseln . . 19

2.3.3 Nachrichtenbehandlung: Die `WHEN`-Klausel 20

2.3.4 Verzögerte Schaltbarkeit: Die `DELAY`-Klausel 24

2.3.5 Boolsche Ausdrücke als Schaltbedingung: Die `PROVIDED`-Klausel 27

2.3.6 Quantifizierung von Transitionen: Die `ANY`-Klausel 29

2.3.7 Formulierung von Zusicherungen 32

2.4 Aufbereitung der Vergleichsergebnisse 36

2.4.1 Die erweiterte effektive PROVIDED-Bedingung 37

2.4.2 Aussagenlogische Interpretation von Ausdrücken: Disjunktive Normalformen 38

2.4.3 Ausdruckstransformation und Gleichheit von Ausdrücken 40

2.4.4 Expansion von Funktionsaufrufen 49

2.4.5 Widerlegbarkeit von Konjunktionen atomarer Aussagen . 53

2.5 Entfernung von „dead code“ 57

3 Implementation des Analysewerkzeugs 61

3.1 Allgemeine Konventionen 61

3.2 Modulstruktur 63

3.2.1 Das Modul *atools* 64

3.2.2 Das Modul *adnf* 65

3.2.3 Das Modul *aexpr* 67

3.2.4 Das Modul *analyse* 73

3.3 Erweiterungsmöglicheiten 75

3.4 Einsatz des Analysewerkzeugs 76

3.4.1 Übersetzung 76

3.4.2 Aufruf 77

3.4.3 Meldungen und Warnungen 80

3.4.4 Struktur des Analyseprotokolls 82

4 Zusammenfassung und Ausblick 86

A Reduktionsordnungen 88

Abbildungsverzeichnis 91

Literaturverzeichnis 93

Kapitel 1

Einleitung

1.1 Motivation

Die Entwicklung zunehmend komplexer werdender Kommunikationsprotokolle macht in steigendem Maße den Einsatz von Modularisierungsansätzen notwendig. Einer dieser Ansätze beruht auf der Idee, verschiedene Protokollelemente zunächst separat zu entwickeln und zu testen, um diese erst in einer späteren Entwicklungsphase zu einem Protokoll zu verschmelzen. Besonders interessant ist diese Technik dann, wenn Protokollelemente im Sinne von parametrierbaren Protokollen nur optional und in verschiedenen Kombinationen zum Einsatz kommen.

Beim Zusammenfügen der verschiedenen Protokollelemente kann es jedoch neben den beabsichtigten auch zu unerwarteten Interaktionen zwischen den zusammengefügten Protokollteilen kommen. Die Komplexität der Analyse und eventuellen Behandlung solcher Effekte steigt mit der Anzahl der optionalen Protokollanteile rapide an, da unter Umständen jede mögliche Kombination zu neuen (erwünschten oder unerwünschten) Interaktionen führt.

Ein Beispiel für derartige optionale Protokollelemente sind Features in Intelligenten Netzwerken ([ITU93]). Hier soll aufbauend auf einem Basisdienst für jeden Teilnehmer eine individuelle Auswahl aus einer großen Menge verschiedener Kommunikationsdienstleistungen (**Features**) angeboten werden. Viele dieser Features können nicht ohne weiteres nebeneinander bestehen, wenn sie z.B. widersprüchliche Zielsetzungen verfolgen. Derartige Interaktionen (**Feature Interactions**) ([Bow+89, Cam+94]) müssen aufgelöst werden, um ein (gemäß des intendierten Verhaltens) korrektes und zuverlässiges Gesamtprotokoll zu erhalten. Es bleiben dabei zwei Fragen offen: *Wie erkennt man Feature Interaktionen* und *wie beseitigt man sie*?

Einer der Ansätze zur Lösung dieses Problems (siehe [BrGo94a, BrGo94b]) basiert auf der formalen Beschreibungstechnik **Estelle** ([ISO89]). Hierbei werden die einzelnen Features[1] jeweils innerhalb eines Moduls als Menge von Transitionen realisiert, wobei die Aktivierung eines Features zunächst einzig durch

[1] Der Basisdienst wird in diesem Fall ebenfalls als Feature betrachtet.

Aktivierung der zugehörigen Transitionen erfolgt.[2] Durch die Einhaltung einiger weiterer Stilvorschriften wird dann sichergestellt, daß sich sämtliche möglichen Feature Interaktionen in Form von **Indeterminismus** zwischen Transitionen verschiedener Features manifestieren.[3]

Es genügt in diesem Fall also, alle Situationen zu bestimmen, in denen es zu Indeterminismus zwischen Transitionen verschiedener Features kommen kann, um gegebenenfalls durch die Vergabe von entsprechenden Prioritäten an die beteiligten Features (d.h. an alle Transitionen eines Features) diesen Indeterminismus zu beseitigen, und so eine Vorrangbeziehung (**Präzedenz**) zwischen den Features zu realisieren. In Fällen, in denen es nicht genügt, nur eines der Features zu bevorzugen, sondern ein völlig neues Verhalten zur Beseitigung der Feature-Interaktion nötig ist, kann durch das Hinzufügen eines zusätzlichen Features (zusätzliche Transitionen) mit noch höherer Priorität dieses neue Verhalten realisiert und die Feature-Interaktion behandelt werden.

In beiden Fällen kann es dabei dazu kommen, daß Transitionen aus Features mit geringerer Priorität in ihrer Ausführbarkeit eingeschränkt werden, da in einigen (bzw. allen) Zuständen, in denen die Transition schaltbar werden kann, ebenfalls eine oder mehrere andere Transitionen höherer Priorität schaltbar werden. In diesen Situationen kann immer nur eine der anderen Transition ausgewählt werden. Diese Konstellation nennen wir **partielle** (bzw. **vollständige**) **Überlappung**. Sie wird unter dem o.g. Spezifikationsstil gezielt eingesetzt, um bei der Aktivierung eines Features allein durch Einfügen von Transitionen andere Transitionen aus Features niederer Priorität teilweise oder ganz zu ersetzen.

Es wird sich zeigen, daß vollständig überlappte und damit nicht ausführbar gewordene Transitionen ohne Semantikänderung aus der Spezifikation entfernt werden können. Dies verbessert i.a. die Ausführungsgeschwindigkeit einer Implementation der Spezifikation, da dadurch die Auswahl der zu schaltenden Transitionen vereinfacht wird.

Zur Durchführung des geschilderten Verfahrens ist es also notwendig, für jede neu gebildete Kombination von Features folgende Punkte zu überprüfen:

- *Ist **Indeterminismus** zwischen Transitionen verschiedener Features möglich?*

  Dies kann auf eine nicht behandelte Feature-Interaktion hindeuten.

- *In welchem Maße wird die **Ausführbarkeit** von Transitionen durch andere Features höherer Priorität **eingeschränkt** (partielle Überlappung)?*

  Unter Umständen kann dies dazu führen, daß die eingeschränkten Features nicht mehr (im intendierten Sinn) korrekt arbeiten.

[2] Die notwendige Möglichkeit zur Erweiterung der Menge der Hauptzustände und der Zustandsvariablen und die damit verbundene erweiterte Initialisierung des jeweiligen Moduls macht weitere Änderungen der Spezifikation notwendig. Zu den Details sei auf [BrGo94a, BrGo94b] verwiesen.

[3] Voraussetzung ist, daß alle Transitionen in dieser Phase gleiche Prioritäten besitzen.

- *Welche Transitionen werden durch andere Features höherer Priorität völlig **unerreichbar** (vollständige Überlappung)?*

  Diese Spezialisierung des vorherigen Punktes ermöglicht zusätzlich eine automatische Optimierung des Laufzeitverhaltens einer Implementation der Spezifikation, da diese Transitionen entfernt werden können, ohne die Semantik der Spezifikation zu verändern, und dadurch bei der Ausführung der Implementation die unnötige ständige Prüfung der Ausführbarkeit eingespart werden kann.

Im folgenden bezeichnen wir Konstellationen, in denen Indeterminismus oder eine Überlappung vorliegt, auch als **Interferenz** zwischen den beteiligten Transitionen.

Die Erstellung eines Werkzeuges zur statischen Prüfung dieser Punkte anhand einer Estelle-Spezifikation ist das Ziel dieser Arbeit. Aufgrund der mangelnden allgemeinen Berechenbarkeit dieser Problemstellungen sind jedoch einige Einschränkungen der Aufgabenstellung notwendig. Diese werden im folgenden Abschnitt dargestellt.

1.2 Konkretisierung der Aufgabenstellung

Ausgangspunkt der Analyse ist eine Estelle-Spezifikation, die anhand des o.g. Spezifikationsstils erstellt wurde und nur Transitionen aus *aktiven* Features enthält.

Es gibt zwei grundlegende Hindernisse bei der allgemeingültigen Lösung der oben genannten Punkte:

- Die Erfüllbarkeit eines Estelle-Ausdrucks ist nicht entscheidbar.
- Die Erreichbarkeit eines Zustandes einer Spezifikation ist nicht entscheidbar.

Der Beweis zur ersten Aussage kann leicht indirekt geführt werden:

> Angenommen, die Erfüllbarkeit eines Estelle-Ausdrucks ist entscheidbar, d.h. es gibt ein Programm P, das unter Eingabe eines boolwertigen Estelle-Ausdrucks E in endlicher Zeit hält und folgende totale Funktion berechnet:
>
> $$\Phi_P(E) = \begin{cases} true & \text{falls} \quad \Phi_E \text{ erfüllbar.} \\ false & \text{sonst.} \end{cases}$$
>
> O.B.d.A.[4] sei dieses Programm als boolwertige Estelle-Funktion formuliert.

[4] Estelle ist (Turing-) vollständig.

Nach dem Diagonalisierungslemma existiert folglich ein Programm R (ebenfalls boolwertige Estelle-Funktion, s.o.), das unter Eingabe eines boolwertigen Estelle-Ausdrucks E in endlicher Zeit hält und folgende totale Funktion berechnet:

$$\Phi_R(E) = \begin{cases} true & \text{falls} \quad R = E \text{ und } \Phi_P(R) = false. \\ false & \text{sonst.} \end{cases}$$

Somit gilt, daß Φ_R erfüllbar ist gdw. $\Phi_P(R) = false$.

Damit ergibt sich das Ergebnis von $\Phi_P(R)$ mit

$$\Phi_P(R) = \begin{cases} true & \text{gdw.} \quad \Phi_R \text{ erfüllbar} & \text{gdw.} \quad \Phi_P(R) = false. \\ false & \text{gdw.} \quad \Phi_R \text{ nicht erfüllbar} & \text{gdw.} \quad \Phi_P(R) = true. \end{cases}$$

Da beide Fälle widersprüchlich sind, kann Φ_P nicht total sein. Damit ist die Annahme widerlegt, d.h. es gibt keine derartige Entscheidungsfunktion.

Die zweite Aussage kann leicht mit Hilfe der ersten bewiesen werden, da z.B. auch die Erfüllbarkeit einer PROVIDED-Klausel allgemein unentscheidbar ist und damit auch nicht entscheidbar ist, ob eine Transition und der damit verbundene Zustandswechsel ausgeführt werden kann.

Aus diesen Gründen kann es nicht das Ziel sein, *sicher* zu entscheiden, ob eine Überlappung oder Indeterminismus zwischen Transitionen vorliegt, sondern es können nur alle Fälle, in denen dies *möglich* ist und die dazu zu erfüllenden *Bedingungen* ermittelt werden. Ob diese Bedingungen erfüllbar sind oder die Zustände, unter denen sie erfüllt werden, überhaupt durch die Spezifikation erreicht werden können, kann nicht eindeutig entschieden werden. Daher werden alle Situationen gemeldet, unter denen Indeterminismus oder Überlappungen zwischen Transitionen verschiedener Features *nicht sicher ausgeschlossen* werden können. Diese Vorgehensweise bezeichnen wir als **offensive Meldestrategie**. Bei der Entfernung von Transitionen, die vollständig überlappt werden, wird dagegen die entgegengesetzte Strategie verfolgt: Um eine Veränderung der Semantik der Spezifikation durch das Entfernen von Transitionen auszuschließen, werden nur solche Transitionen automatisch entfernt, die *sicher nicht ausführbar* sind. Dies bezeichnen wir als **defensive Optimierungsstrategie**.

Mit Hilfe des Begriffs der **effektiven Schaltbedingung**[5] einer Transition, d.h. der Bedingung, unter der die Transition tatsächlich ausführbar ist (ohne z.B. durch eine andere Transition überlappt zu werden) können beide Strategien auf die Frage nach der *möglichen Erfüllbarkeit* bzw. *sicheren Unerfüllbarkeit* eines Ausdrucks zurückgeführt werden:

- Eine mögliche Interferenz liegt vor, wenn die Interferenzbedingungen potentiell erfüllbar sind, d.h. ihre *Unerfüllbarkeit* nicht sicher nachgewiesen werden kann.

[5] siehe auch Kapitel 2

- Die Nichtausführbarkeit einer Transition kann nur dann angenommen werden, wenn die *Unerfüllbarkeit* ihrer **effektiven Schaltbedingung** nachgewiesen wurde.

In beiden Fällen betrachten wir jedoch lediglich die *(prädikatenlogische) Erfüllbarkeit*[6] der abgeleiteten Bedingungen, nicht jedoch die *Erreichbarkeit einer erfüllenden Situation* durch die Spezifikation. So ist z.B. die Bedingung $x = 10$ prädikatenlogisch erfüllbar, es ist jedoch durchaus möglich, daß in einer konkreten Spezifikation die Variable x niemals den Wert 10 annehmen kann. Nachdem die Erreichbarkeit eines konkreten Zustands einer Spezifikation jedoch nicht entscheidbar ist, wird zunächst auch nur die prädikatenlogische Erfüllbarkeit vom Analysewerkzeug untersucht.[7]

Dieses Vorgehen entspricht vollständig den o.g. Strategien: Da die *Erfüllbarkeit einer Bedingung* eine schwächere Anforderung als die *Erreichbarkeit einer erfüllenden Situation* ist[8], werden im „Zweifelsfall" gemäß der offensiven Meldestrategie mögliche Interferenzen gemeldet (Interferenzbedingung nicht widerlegbar) und gemäß der defensiven Optimierungsstrategie möglicherweis ausführbare Transitionen nicht entfernt (effektive Schaltbedingung nicht widerlegbar).

Durch die offensive Meldestrategie wird somit das Problem umgangen, daß es nicht entscheidbar ist, ob die Bedingung für eine Interferenz erfüllbar ist bzw. ob einer der dazu notwendigen Zustände von der Spezifikation erreicht werden kann: Die Beurteilung der Erfüllbarkeit einer Interferenzbedingung (bzw. die Erreichbarkeit eines erfüllenden Zustandes durch die Spezifikation) wird in letzter Instanz dem Benutzer überlassen, sofern die Bedingung nicht bereits automatisch als sicher unerfüllbar erkannt wurde. Um die Zahl dieser Meldungen jedoch möglichst gering zu halten, liegt das Hauptaugenmerk darauf, in einer Vielzahl häufig auftretender Konstellationen unerfüllbare Bedingungen auch als solche erkennen zu können. Dabei dienen existierende Beispielspezifikationen als Maßstab. Dementsprechend wichtig ist eine gute Erweiterbarkeit, durch die jederzeit eine Anpassung an spezielle Bedürfnisse möglich ist.

Später wird in Kapitel 2.3.7 mit den sogenannten „**Zusicherungen**" eine Möglichkeit vorgestellt, bei der Analyse neben der reinen Erfüllbarkeit auch die Erreichbarkeit eines erfüllenden Zustands (teilweise) zu berücksichtigen, um so die Qualität der Analyse zu verbessern.

Wie bereits erläutert erfolgt die Analyse **statisch**, d.h. es werden keine Laufzeitsimulationen durchgeführt oder Ereignisfolgen über mehrere Transitionen hinweg verfolgt. Darüber hinaus wird (mit wenigen Ausnahmen) im Hinblick auf die mangelnde Berechenbarkeit auch darauf verzichtet, Informationen über mögliche Zustandsentwicklungen wie Modulinstanziierungen, Verbindungsauf- und Abbautätigkeiten, die Erzeugung von Nachrichten oder den Werteverlauf

[6] Mit der *Erfüllbarkeit* eines Ausdrucks ist im folgenden immer die *Erfüllbarkeit in der durch die Estelle-Semantik definierten Algebra* gemeint.

[7] Eine Ausnahme bilden lediglich die Hauptzustände, siehe Kapitel 2.3.2.

[8] Für eine gegebene Bedingung impliziert die *Erreichbarkeit einer erfüllenden Situation* natürlich ihre *Erfüllbarkeit*.

von Variablen zu sammeln. Da zudem in diesem Kontext nur Interaktionen zwischen Features jeweils einer Modulinstanz analysiert werden sollen, erfolgt die Analyse ebenfalls ausschließlich **modullokal**. Einflüsse zwischen Modulen[9] werden nicht mit einbezogen.

Diese Einschränkungen ermöglichen es, die Analyse weitgehend auf die Betrachtung der Transitionsklauseln zu beschränken und damit die Vorteile von Estelle in Hinblick auf derartige Untersuchungen auszunutzen: In einer Sprache wie z.B. C++ wäre es i.a. kaum möglich, ohne eine umfassende Analyse der Wirkungen von Anweisungsfolgen wesentliche Aussagen über die Ausführbarkeit von Codeteilen zu machen. Die starke Strukturierung von Estelle unterstützt jedoch ein derartiges Vorgehen ganz erheblich. Es wird sich zeigen, daß trotz der oben gemachten Einschränkungen sehr konkrete Aussagen über die Interaktionen zwischen Features gemacht werden können.

1.3 Technisches Umfeld

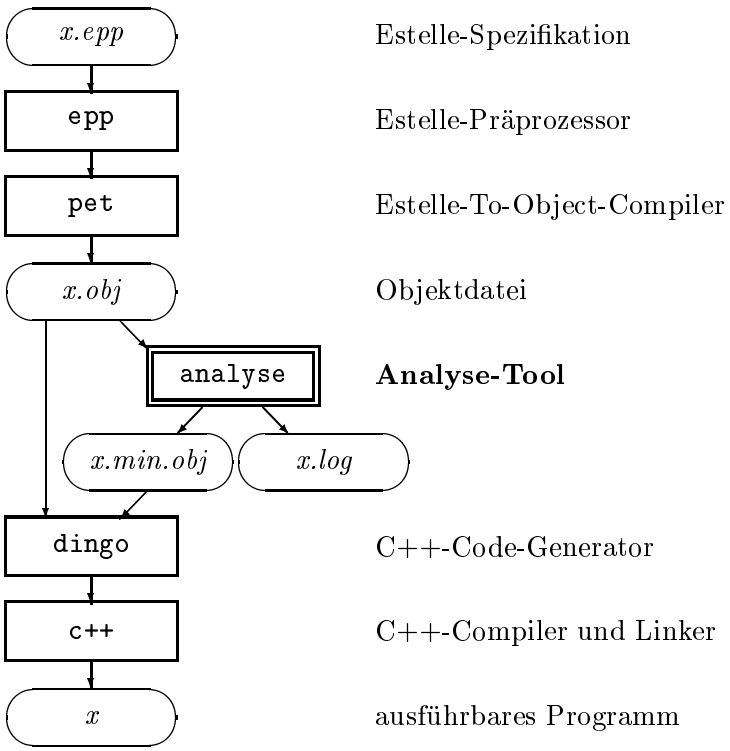


Abbildung 1.1: Das Analysewerkzeug im Datenfluß von PET-DINGO

Der Ausgangspunkt der Analyse ist eine Estelle-Spezifikation in Form einer Objektdatei, wie sie als Zwischenprodukt bei der Übersetzung durch das PET-DINGO-Toolkit entsteht (siehe Abbildung 1.1). EPP ist ein Estelle-Präprozessor ([BrGo94a, BrGo94b]), mit dessen Hilfe die Transitionen der einzelnen

[9] z.B. exportierte Variablen, Vater-Sohn-Priorität, Vorbedingungen für die Erzeugung von Nachrichten, die Existenz von Verbindungen und die Instanziierung von Modulen

Features textuell ein- oder ausgeschlossen werden können. Ergebnis ist wiederum ein Estelle-Spezifikationstext. PET-DINGO ist ein Werkzeug zur automatischen Erstellung von Prototyp-Implementierungen von Estelle-Spezifikationen. Es beinhaltet folgende Programme:

- **PET** (*Portable Estelle Compiler*) stellt den Compiler-Frontend dar. Dieses Programm übersetzt eine Estelle-Spezifikation in das o.g. Objektformat, das dann in eine Datei ausgegeben wird ([SiSt91b]).
- **DINGO** (*Distributed Implementation Generator*) kann aus einer solchen Objektdatei eine Implementation in Form von C++-Quelltext erzeugen, der mit Hilfe entsprechender Compiler in lauffähige Programme umgesetzt werden kann ([SiSt91a]).
- **PDRESTORE** kann aus einer Objektdatei wieder einen Estelle-Spezifikationstext gewinnen.

Die Funktionalität zur Handhabung des Objektformats ist als getrennte objektorientierte Bibliothek implementiert, die im folgenden als **PET-Klassenbibliothek** bezeichnet wird. Sie ist – genau wie das gesamte PET-DINGO-Toolkit – in C++ ([Str86, BaTö90, PlBr90]) implementiert.

Das Analysewerkzeug soll als Eingabe eine solche Objektdatei akzeptieren. Zum Zugriff setzt es auf der PET-Klassenbibliothek auf. Als Ausgabe soll neben einem Protokoll der Analyseergebnisse eine neue Objektdatei erzeugt werden, die zur Ausgangsspezifikation semantisch äquivalent ist, in der jedoch alle Transitionen entfernt wurden, die als unerreichbar erkannt werden konnten. Diese Ausgabedatei soll dann direkt als Eingabe für PDRESTORE oder DINGO genutzt werden können (siehe Abbildung 1.1).

1.4 Gliederung

In den folgenden Kapiteln wird die Vorgehensweise bei der Entwicklung des Analysewerkzeugs von den abstrakten Ideen hin zur konkreten Realisierung schrittweise dargestellt.

Im Kapitel *„Die Theorie des Analysevorgangs“* werden die grundsätzlichen Ideen zur automatischen Erkennung der o.g. Punkte dargestellt. Dabei wird (wenn auch nur in zweiter Linie) auch auf die Einschränkungen der praktischen Realisierung eingegangen.

Im darauffolgenden Kapitel *„Implementation des Analysewerkzeugs“* werden die einzelnen Komponenten des Analysewerkzeugs und deren Zusammenwirken beschrieben. Dabei wird auch eine Anleitung zum praktischen Einsatz des Programms gegeben.

Nach dem vierten Kapitel *„Zusammenfassung und Ausblick“* wird dann im Anhang schließlich noch eine kurze Einführung in die Reduktionsordnungen der RPO-Klasse[10] gegeben.

[10] wird in Abschnitt 2.4.3 benutzt

Kapitel 2

Die Theorie des Analysevorgangs

In diesem Kapitel wird vorgestellt, *welche Informationen* für das angestrebte Analyseergebnis ermittelt werden müssen und *wie* diese aus der Estelle-Spezifikation gewonnen werden können. Dabei steht hier zunächst die Frage nach dem theoretisch Ableitbaren[1] im Vordergrund; nur in zweiter Linie werden Zugeständnisse an die praktische Realisierbarkeit diskutiert. Die Implementation des Werkzeugs wird dann das Thema des darauffolgenden Kapitels sein.

Ziel der Analyse ist es,

1. **potentiellen Indeterminismus** und **potentielle Überlappungen** zwischen Transitionen verschiedener Features zu entdecken und die Bedingungen anzugeben, unter denen diese auftreten können;
2. zu den überlappten Transitionen jeweils die **effektiven Schaltbedingungen** zu bestimmen, d.h. festzustellen, unter welchen Bedingungen die Transition tatsächlich ausführbar ist, ohne daß sie durch eine andere Transition (höherer Priorität) überlappt wird;
3. zu erkennen, welche Transitionen **sicher nicht ausführbar** sind, d.h. bei welchen Transitionen die *effektiven* Schaltbedingungen unerfüllbar sind;
4. nicht ausführbare Transitionen optional aus dem Binärformat der Spezifikation zu entfernen, um dadurch eine **Laufzeitoptimierung** einer Implementation der Spezifikation zu erreichen.

Man beachte bei diesen Punkten, wann von „möglichen“ und wann von „sicheren“ Ereignissen die Rede ist: *Potentieller* Indeterminismus oder *potentielle* Überlappungen werden gemeldet, wenn die abgeleitete Bedingung für ihr Auftreten nicht widerlegt werden kann[2], Transitionen werden jedoch nur dann als

[1] zum gesamten Kapitel siehe [Bör87, Sie90, Put88, Ave95]

[2] offensives Meldeverhalten

unerreichbar gekennzeichnet bzw. entfernt, wenn sie *sicher* unerreichbar sind, d.h. wenn ihre effektive Schaltbedingung *nicht potentiell erfüllbar* ist.[3].

Die oben genannten **Bedingungen** (Indeterminismus-, Überlappungs- und effektive Schaltbedingungen) beschränken sich dabei nur auf Hauptzustände und auf boolsche Ausdrücke. Das Auftreten oder Nichtauftreten von Nachrichten (WHEN-Klausel) sowie das Voranschreiten von Zeit (DELAY-Klausel) werden darin nicht formuliert. Entsprechend ist es möglich, daß in diesen Bedingungen nicht alle Anforderungen angegeben sind: Zum Beispiel kann eine Überlappungsbedingung vom regelmäßigen Auftreten einer Nachricht abhängen. Eine solche **unsichere** Überlappungsbedingung wird zwar (optional) vom Analysewerkzeug gemeldet, sie hat jedoch keinen Einfluß auf die effektive Schaltbarkeit, da sie nicht *sicher* vorliegt, sobald die abgeleitete Bedingung erfüllt ist (siehe Abschnitt 2.3.3).

In allen Fällen wird jedoch das Zurückhaltungsgebot bei der Optimierung strikt eingehalten, da nur so eine Veränderung der Semantik durch die Entfernung von Transitionen vermieden werden kann.

Im folgenden wird dargestellt, anhand *welcher* Elemente einer Estelle-Spezifikation die zur Analyse notwendigen Informationen ermittelt werden können und *wie* daraus dann die Beantwortung der obigen Punkte möglich ist. Dazu wird zunächst in mehreren Verfeinerungsschritten demonstriert, wie aus einer ganzen Spezifikation, einem Modulrumpf, den Transitionen und schließlich aus den Transitionsklauseln jeweils relevante Informationen abgeleitet werden können. Die obigen Fragen werden dabei am Ende auf die Frage nach der Erfüllbarkeit existenzquantifizierter Ausdrücke zurückgeführt. Diese Erfüllbarkeit wird aufgrund ihrer mangelnden Berechenbarkeit die eigentliche Grenze der exakten Analysemöglichkeiten darstellen. An dieser Stelle bleiben dann nur noch heuristische Methoden, um zumindest für konkrete Problemklassen eine exakte[4] Analyse führen zu können.

Ausgangspunkt soll jedoch zunächst die Frage nach der Analyse einer kompletten Spezifikation sein.

2.1 Globale Analyse

Eine Estelle-Spezifikation setzt sich aus einer Fülle von Definitionen zusammen, die in einem **Baum von Modulrumpf-Definitionen** organisiert sind (Abbildung 2.1). Die Wurzel des Baumes ist die Spezifikation selbst, die ebenfalls die Funktion eines Modulrumpfs hat. Innerhalb der Modulrümpfe sind **Transitionen** definiert, die eigentlich aktiven Elemente der Module. Transitionen beschreiben mögliche Zustandsübergänge der Modulinstanzen.

Wie bereits zu Beginn erläutert, beschränkt sich die Analyse auf Interferenzen zwischen Features *innerhalb* jeweils *eines* Moduls. Deshalb genügt es, die

[3] defensives Optimierungsverhalten

[4] Exakt in dem Sinne, daß nur *echter* Indeterminismus bzw. *echte* Überlappungen gemeldet werden und *alle* nicht ausführbaren Transitionen erkannt werden (s.o.).

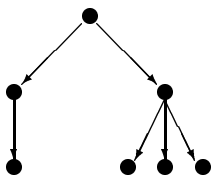


Abbildung 2.1: Modul-Baumstruktur einer Estelle-Spezifikation

Module jeweils isoliert voneinander zu analysieren. Dazu wird zunächst rekursiv dieser Baum durchlaufen, um dann innerhalb eines jeden Modulrumpfs alle folgenden Analyseschritte ausschließlich *lokal* durchzuführen.

2.2 Modullokale Analyse

Bei der lokalen Analyse eines *Modulrumpfs* sind (neben den Zusicherungen, s.u.) nur die *Transitionen* unmittelbar von Bedeutung. Untermodule sind bei der lokalen Analyse ausdrücklich ausgeklammert und alle übrigen Definitionen spielen nur insofern eine Rolle, wie sie durch die Transitionen referenziert werden.

Analog dazu spielen innerhalb der *Transitionen* auch nur die *Transitionsklauseln* bei der Analyse eine Rolle, da der Transitionsblock ausdrücklich aus der Analyse ausgeschlossen wurde und alle anderen Definitionen *von sich aus* keinen Einfluß auf die Semantik der Spezifikation haben. Erst wenn Funktionen oder Konstanten in einer Klausel referenziert werden, werden sie innerhalb der Analyse untersucht. Eine Ausnahme sind wiederum die Zusicherungen, die hier wie auch oben bei der Analyse zwar ausgewertet werden, hinsichtlich der Semantik der Spezifikation jedoch Kommentarcharakter besitzen.

Bei der modullokalen Analyse müssen also die Transitionsklauseln von Transitionen verschiedener Features miteinander in Beziehung gesetzt werden. Dabei sollen zunächst möglicher Indeterminismus und mögliche Überlappungen erkannt werden. *Indeterminismus* im hier gebrauchten Sinne liegt in einem Modul vor, wenn mindestens zwei Transitionen verschiedener Features zum gleichen Zeitpunkt schaltbar sind und daher nicht feststeht, welche der Transitionen ausgewählt wird. Wenn die Transitionen verschiedene Prioritäten haben und nur dieser Umstand Indeterminismus in einem Zustand verhindert, so liegt eine *Überlappung* vor.

Mit einem **Zustand**[5] wird hier die Gesamtheit aller Hauptzustände, Variablenwerte, Modulinstanziierungen, Verbindungsstrukturen und Warteschlangeninhalte der gesamten Spezifikation bezeichnet[6]. Ein konkreter Zustand bestimmt[7] also, welche Transitionen bereit bzw. schaltbar werden, das Schalten von Transitionen ist identisch mit einem definierten Zustandsübergang. In Spezifikationen

[5] Nicht zu verwechseln mit dem *Hauptzustand*!

[6] Aufgrund der DELAY-Transitionsklausel gehört eigentlich auch die Zeit mit zum Zustand der Spezifikation (siehe Abschnitt 2.3.4). Sie wird hier zunächst zurückgestellt.

[7] durch indeterministische Funktionen leider nicht immer eindeutig; siehe Abschnitt 2.4.3

mit Indeterminismus kann auch die Zustandsentwicklung indeterministisch verlaufen. Man kann jedoch in einem solchen Fall alle möglichen Folgezustände betrachten und so Mengen von möglichen Zustandsverläufen entwickeln. Daraus ergibt sich die **Menge der erreichbaren Zustände**.

Die **lokale Schaltbedingung einer Transition**[8] beschreibt eine Teilmenge des Zustandsraumes, in der die Transitionsklauseln `WHEN`, `FROM` und `PROVIDED` erfüllt sind. In Abbildung 2.2 wurde der Zustandsraum eindimensional als horizontale Achse dargestellt. Die Transitionen belegen eine Teilmenge[9] dieser Achse. Auf der vertikalen Achse wurde die Priorität der Transitionen angegeben. Jede Transition hat dabei genau eine Priorität.

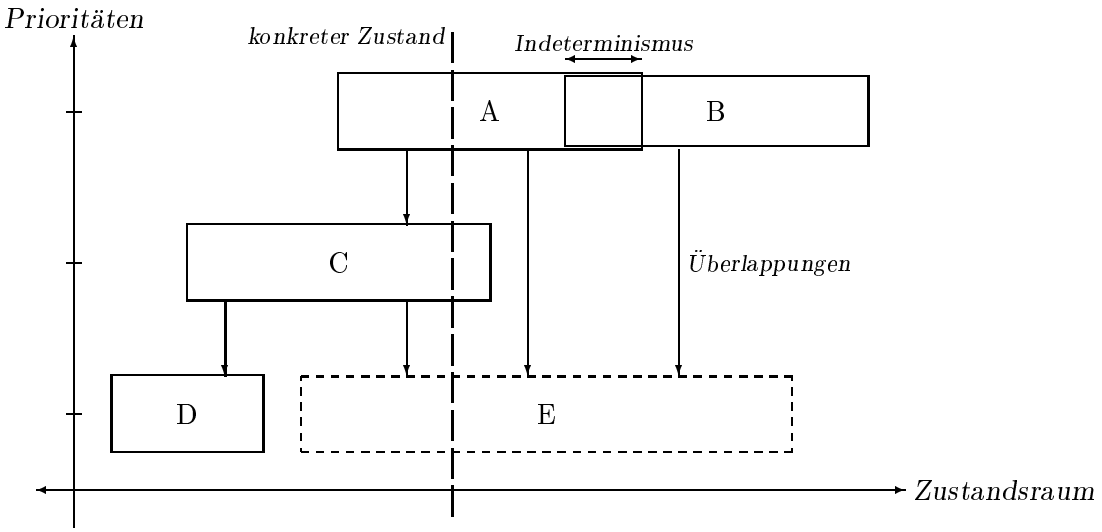


Abbildung 2.2: Überlappungen und Indeterminismus zwischen Transitionen

Man erkennt leicht, daß Indeterminismen (bzw. Überlappungen) sich an gemeinsam erfüllbaren lokalen Schaltbedingungen zeigen, wobei die Prioritäten gleich (bzw. verschieden) sein müssen. In der Abbildung besteht Indeterminismus zwischen A und B, (partielle) Überlappungen gibt es auf C (durch A), D (durch C) und E (durch A, B, C). Die effektive Schaltbedingung einer Transition ist auf solche Zustände beschränkt, in denen keine Transition mit höherer Priorität bereit ist. Im Bild sind die effektiven Schaltbedingungen von C, D und E eingeschränkt, die von E ist sogar unerfüllbar geworden.

Anhand dieses Bildes kann man sich auch einige Folgerungen über Interferenzen verdeutlichen:

- Gibt es einen Zustand, in dem mehrere Transitionen gleicher Priorität zugleich schaltbar werden können, so besteht der Indeterminismus zwischen *jedem Paar* dieser Transitionen.

[8] o.B.d.A. ohne ANY-Klauseln (ansonsten Expansion vornehmen)
[9] im Bild nur zusammenhängend dargestellt, im allgemeinen jedoch zerteilt

- Wird eine Transition in einem Zustand durch mehrere andere Transitionen überlappt, so besteht die Überlappungsbeziehung ebenfalls *paarweise* mit jeder einzelnen dieser Transitionen höherer Priorität.

- Indeterminismus und Überlappungen sind *für konkrete Zustände transitiv*. So wird in Abbildung 2.2 im markierten Zustand die Transition C von A überlappt und E von C. Daraus folgt, daß in diesem Zustand auch E von A überlappt wird. Ohne Beschränkung auf *einzelne Zustände* besteht keine Transitivität (partieller) Überlappungen: Im Beispiel gilt „A überlappt C“ und „C überlappt D“, jedoch nicht „A überlappt D“! (Man beachte, daß die Überlappungen jeweils nur partiell sind.)

Daraus ergeben sich wichtige Folgerungen für die Durchführung der Analyse und der nachfolgenden Entfernung inaktiver Transitionen. Zum einen genügt es zur Erkennung von Indeterminismus und Überlappungen, die *Transitionen nur paarweise* zu *vergleichen*. Es ist daher nicht notwendig, komplexe Interaktionen zwischen drei oder mehr Transitionen zu untersuchen. Dies ist eine wesentliche Vereinfachung der Analyse. Zum anderen ermöglicht erst die Transitivität der Überlappungen die korrekte Entfernung von vollständig überlappten Transitionen, die selbst wiederum andere Transitionen überlappen. Ihre Entfernung hat keine semantischen Auswirkungen, da sie selbst nicht ausführbar sind, aber auch alle von ihnen überlappten Transitionen nach ihrer Entfernung immer noch im gleichem Maße überlappt werden (siehe Abschnitt 2.5).

Man kann sich anhand des Beispiels jedoch auch leicht verdeutlichen, daß *vollständige Überlappungen* durch den paarweisen Vergleich von Transitionen nicht immer unmittelbar zu erkennen sind: Wie bereits erwähnt wird die Transition E durch A, B und C *gemeinsam* vollständig überlappt. Dies ist durch paarweisen Vergleich zwischen Transitionen nicht *unmittelbar* feststellbar, da keine der Transitionen A, B und C *alleine* E vollständig überlappen.

Schränkt man die lokale Schaltbedingung einer Transition jedoch um all diejenigen Zustände ein, in denen sie überlappt wird, so erhält man die **effektive Schaltbedingung**. Diese kann durch *sukzessiven paarweisen Vergleich* der Transitionen ermittelt werden. Vollständige Überlappungen manifestieren sich dann (indirekt) als unerfüllbare effektive Schaltbedingungen.

Somit kann die modullokale Analyse vollständig auf den paarweisen Vergleich von Transitionen reduziert werden. Da innerhalb der Transitionen nur die Transitionsklauseln direkt berücksichtigt werden, wird der Vergleich anhand dieser Klauseln geführt.

Die oben definierten Bedingungen für Indeterminismus und Überlappungen sind natürlich auf die *erreichbaren Zustände* beschränkt. Ebenso müssen die effektiven Schaltbedingungen in der Menge der erreichbaren Zustände erfüllbar sein, um die Transition überhaupt ausführen zu können. Da bei der Analyse jedoch nur sehr eingeschränkte Informationen über die Erreichbarkeit von Zuständen vorliegen (das Fortschreiten der Zustände durch die Transitionsblöcke wird nicht analysiert), können diese Bedingungen auch nur eingeschränkt überprüft wer-

den. Lediglich unerreichbare *Haupt*zustände können – sofern es aus den Transitionsklauseln ersichtlich ist – erkannt werden. So kann zum Beispiel nicht erkannt werden, ob eine Schaltbedingung wie $x = 0$ im Betrieb einer Spezifikation überhaupt jemals erfüllt werden kann (möglicherweise erreicht die Variable x niemals den Wert 0).

Im Zweifelsfall wird ein Zustand als erreichbar eingestuft, zumal dies im Einklang mit der offensiven Meldestrategie und der defensiven Optimierungsstrategie steht: Im Zweifelsfall kann eine Indeterminismus- oder Überlappungsbedingung eintreten und wird daher gemeldet; ebenso kann im Zweifelsfall jedoch auch eine effektive Schaltbedingung erreicht werden, die Transition darf also nicht entfernt werden. Im folgenden wird jedoch nur dann zwischen *„Zuständen“* und *„erreichbaren Zuständen“* unterschieden, wenn dies im Sinne der Analyse eine Rolle spielt.[10]

Eine Sonderstellung unter den Transitionen nehmen die **Initialisierungs-Transitionen** ein. Bei der Modulinstanziierung wird *genau eine* der Initialisierungstransitionen ausgeführt (sofern mindestens eine definiert wurde). Diese dürfen nur TO- und PROVIDED-Klauseln besitzen, wobei sich die PROVIDED-Klauseln in dem bei der Modulinitialisierung herrschenden *präinitialen Zustand* sinnvoller Weise meist auf die Modulparameter beziehen.[11] Da die Werte dieser bei der Modulinstanziierung übergebenen Parameter dem Analysewerkzeug unbekannt sind, können normalerweise auch keine Aussagen über die *Ausführbarkeit von Initialisierungstransitionen* gemacht werden. Daher werden Initialisierungstransitionen immer als ausführbar betrachtet[12].

Da zudem Interferenzen mit „normalen“ Transitionen nicht auftreten können[13], spielen Initialisierungstransitionen bei der Analyse zunächst keine Rolle. Lediglich bei der Bestimmung der erreichbaren Hauptzustände müssen ihre TO-Klauseln ausgewertet werden.[14]

2.3 Vergleich von Transitionsklauseln

Zum Vergleich zweier Transitionen werden die Transitionsklauseln gleichen Typs ebenfalls *paarweise* miteinander verglichen. Dabei kann oft als Ersatz für eine fehlende (explizite) Klausel eine dazu äquivalente Klausel angenommen werden. Zum Beispiel kann das Fehlen einer PROVIDED-Klausel als „`PROVIDED TRUE`“ interpretiert werden. Bei anderen Klauseln (z.B. WHEN-Klausel) gibt es keine äquivalente explizite Klausel.

[10] Also bei Hauptzuständen und bei den Zusicherungen (s.u.).

[11] Neben den Modulparametern haben zu diesem Zeitpunkt nur lokale Konstanten einen vordefinierten Wert. Ein PROVIDED-Ausdruck ohne Bezug auf die Modulparameter hat daher einen konstanten Ergebniswert.

[12] Wird eine offensichtlich unerfüllbare PROVIDED-Bedingung erkannt, so gibt das Analysewerkzeug lediglich eine Warnung aus.

[13] Initialisierungstransitionen können nur *während* der Modulinstanziierung ausgeführt werden, die übrigen Transitionen nur *danach*.

[14] siehe Abschnitt 2.3.2

Ziel des Vergleichs ist es, zunächst festzustellen, ob zwei Transitionen überhaupt **zueinander „passen“**, d.h. ob es überhaupt (erreichbare) Zustände gibt, die beide Klauseln zugleich erfüllen. So passen zum Beispiel die Klauseln „`FROM state1`“ und „`FROM state2`“ nicht zueinander, da sich die beiden Hauptzustände `state1` und `state2` gegenseitig ausschließen.

Bei vielen zueinander passenden Zuständen können auch Bedingungen für die Zustände angegeben werden, in denen beide Klauseln zugleich erfüllt sind (**Interferenzbedingung**). Die Bestimmung dieser Bedingungen ist das zweite Ziel des Vergleichs. So passen die Klauseln „`PROVIDED` $x \geq 0$“ und „`PROVIDED` $x \leq 10$“ zueinander (z.B. die Belegung 5 für x erfüllt beide) und liefern die Interferenzbedingung $(x \geq 0) \wedge (x \leq 10)$.

Im folgenden wird für jeden Transitionsklausel-Typ angegeben, wann zwei Klauseln *zueinander passen* und welche *Interferenzbedingungen* abgeleitet werden können. Im Anschluß daran wird gezeigt, wie diese Informationen zusammengefügt werden können.

2.3.1 Prioritäten von Transitionen: Die `PRIORITY`-Klausel

Die PRIORITY-Klausel enthält als Argument eine ganzzahlige Konstante größer oder gleich Null, die die Priorität der Transition angibt. Je größer diese Zahl ist, desto kleiner ist die Priorität der Transition, eine fehlende PRIORITY-Klausel führt zu einer geringeren Priorität als jede explizite Angabe und entspricht damit also „`PRIORITY` ∞“.

Passen bei einem Paar von Transitionen alle übrigen Klauseln zueinander, so kann möglicher Indeterminismus oder eine mögliche Überlappung vorliegen. Bei *gleicher Priorität* kann es sich nur um Indeterminismus handeln. Haben die Transitionen jedoch *nicht nachweislich gleiche Prioritäten*, so kann nicht immer bestimmt werden, ob eine Überlappung, Indeterminismus oder keine Interferenz vorliegt. Zum einen ist nicht immer feststellbar, ob die Prioritäten tatsächlich verschieden sind und welche ggfs. die größere ist (z.B. bei ANY-Konstanten), zum anderen kann es auch bei eindeutig verschiedenen Prioritäten Indeterminismus geben (zusammen mit DELAY-Klauseln, siehe Abschnitt 2.3.4).

Leicht zu entscheiden ist das Prioritätsverhältnis, wenn höchstens eine der Transitionen eine *explizite* PRIORITY-Klausel besitzt: Eine explizite Angabe bewirkt, wie bereits erwähnt, eine höhere Priorität als eine implizite; hat keine der Transitionen eine PRIORITY-Klausel, so sind die Prioritäten gleich.

Haben jedoch *beide* zu vergleichenden Transitionen eine PRIORITY-Klausel, so kann nur dann eindeutig entschieden werden, welche die Transition mit höherer Priorität ist, wenn die Werte der beiden Konstanten zum Zeitpunkt der Analyse feststehen. Leider ist es in Estelle jedoch nicht immer möglich, diese Entscheidung anhand der Spezifikation zu treffen, da unvollständig spezifizierte Konstanten (*ANY-Konstanten*) zwar syntaktisch als Konstanten gelten und daher in PRIORITY-Klauseln zum Einsatz kommen können, ihr Wert jedoch zum Zeitpunkt der Analyse noch nicht feststeht (Abbildung 2.3). Es kann lediglich die Gleichheit zweier (möglicherweise auch mittelbarer) Referenzen auf

die gleiche Konstante sicher erkannt werden, weitere Aussagen zu den beiden unvollständig spezifizierten Konstanten sind nicht möglich.

```
CONST Prio1 = ANY INTEGER;
      Prio2 = Prio1;
      Prio3 = 10;

TRANS                TRANS                TRANS
   PRIORITY Prio1       PRIORITY Prio2       PRIORITY Prio3
   NAME t1:             NAME t2:             NAME t3:
      BEGIN                BEGIN                BEGIN
      END;                 END;                 END;
```

Abbildung 2.3: Transitionen mit vergleichbaren (`t1` mit `t2`) und unvergleichbaren Prioritäten (`t1`/`t2` mit `t3`)

In den Fällen, in denen zwei zueinander passende Transitionen in der oben beschriebenen Prioritäts-Partialordnung unvergleichbar sind, werden sowohl Überlappungen in beide Richtungen (!) als auch Indeterminismus gemeldet, wobei als Bedingungen jeweils die notwendigen Werteverhältnisse explizit hinzugenommen werden. Zwischen den Transitionen `t1` und `t3` bestehen die folgenden Interferenzen:

Interferenz-Art				Bedingung
Indeterminismus zwischen	`t1`	und	`t3`	`Prio1` $= 10$
Überlappung von	`t1`	über	`t3`	`Prio1` < 10
Überlappung von	`t3`	über	`t1`	`Prio1` > 10

In der konkreten Implementation der Spezifikation wird natürlich höchstens einer der drei Interferenz-Arten eintreten, da der Konstanten `Prio1` dann ein fester Wert zugeordnet wird. Die Möglichkeit des Indeterminismus bzw. der Überlappungen kann jedoch nicht ausgeschlossen werden und wird *offensiv* gemeldet.

Die effektiven Schaltbedingungen lauten daher für die drei Transitionen folgendermaßen:

Transition	effektive Schaltbedingung	überlappt durch
`t1`	$\neg$(`Prio1` > 10)	`t3`
`t2`	$\neg$(`Prio1` > 10)	`t3`
`t3`	$\neg$(`Prio1` < 10)	`t1`, `t2`

Die obigen effektiven Provided-Ausdrücke sind im allgemeinen jedoch im Rahmen der Analyse nicht als widersprüchlich nachweisbar und daher wird keine der Transitionen entfernt werden. Dies entspricht dem *defensiven* Optimierungsverhalten.

Dabei dient die Ausgabe aller drei möglichen Beziehungen nicht nur dazu, dem Implementeur einer Spezifikation die exakten Bedingungen für das Auftreten der verschiedenen Interferenz-Möglichkeiten anzugeben. Kann nämlich aus anderer Quelle ein Widerspruch zu einer oder mehreren der obigen Beziehungen zwischen den Prioritätskonstanten abgeleitet werden, so wird unter Umständen nachträglich noch einer der Interferenztypen als zutreffend erkannt. Wird zum Beispiel die Zusicherung (s.u.) angegeben, daß `Prio1` > 20 gilt, so kann die Überlappung von `t3` über `t1` und `t2` als die einzige zutreffende sicher erkannt und daraufhin `t1` und `t2` aus der Spezifikation entfernt werden.

Neben ihrer Estelle-Semantik hat die PRIORITY-Klausel noch eine weitere wichtige Bedeutung für die Analyse. Über sie wird (gemäß des von [BrGo94a, BrGo94b] angegebenen Spezifikationsstils) die **Feature-Zugehörigkeit** einer Transition angegeben: Zwei Transitionen gehören dem selben Feature an, wenn sie PRIORITY-Klauseln mit Konstantenreferenzen besitzen und dabei von beiden die selbe Konstantendefinition (direkt) referenziert wird. Dabei werden Konstanten, die nur indirekt als gleich definiert sind (z.B. `Prio1` und `Prio2` im obigen Beispiel), bezüglich der Featurezugehörigkeit als *verschieden* angesehen.

Insbesondere bilden Transitionen ohne PRIORITY-Klausel oder solche mit direkter Angabe einer Zahlenkonstanten (z.B. „`PRIORITY 10`“) implizit jeweils ihr eigenes Feature. Die Instanzen einer Transition mit ANY-Klausel werden in dieser Hinsicht wie *eine Transition* behandelt und gehören entsprechend immer zum selben Feature (auch ohne PRIORITY-Klausel). Interferenzen zwischen diesen werden daher nicht geprüft (siehe auch Abschnitt 2.3.6).

Wie man leicht erkennt, ist es so nicht möglich, den Transitionen eines Features *verschiedene* Prioritäten zu geben. Um dies später einmal umgehen zu können, wird bei den referenzierten Prioritätskonstanten nicht auf Gleichheit der referenzierten Definition, sondern auf Namensgleichheit geprüft. Sobald eine Namenskonvention für die verschiedenen Prioritätskonstanten eines Features festgelegt wurde, kann dieser Namensvergleich angepaßt werden. So müßten unter der Konvention in Abbildung 2.4 nur die angegebenen Konstantennamen bis zum ersten "`_`" übereinstimmen, um die Zugehörigkeit zum selben Feature anzuzeigen (siehe Abschnitt 3.3).

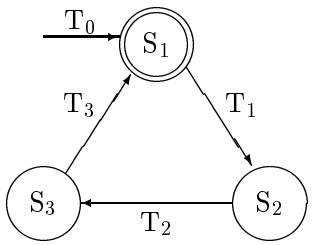
feature-priority-clause	::= "`PRIORITY`" feature-priority-constant
feature-priority-constant	::= unsigned-integer | feature-priority-constant-id
feature-priority-constant-id	::= feature-id["`_`"local-priority-id]
feature-id	::= Char {Char | Digit}
local-priority-id	::= {Char | Digit | "`_`"}
Char	::= "`a`" | ... | "`z`"
Digit	::= "`0`" | ... | "`9`"

Abbildung 2.4: Spezialisierte (Feature-) Namen für Prioritätskonstanten

2.3.2 Hauptzustandsentwicklung: Die FROM- und TO-Klauseln

FROM- und TO-Klauseln treten nur in Modulrümpfen mit explizit definierten Hauptzuständen auf. Sie beschreiben die Schaltbedingungen bzgl. des Hauptzustandes (FROM-Klausel) und den Folgezustand nach dem Schalten der Transition (TO-Klausel).

Ob zwei FROM-Klauseln zueinander passen, kann relativ einfach ermittelt werden: Interferenzen zwischen zwei Transitionen können nur dann auftreten, wenn es erreichbare Hauptzustände gibt, die in den FROM-Klauseln **beider** Transitionen enthalten sind. Dabei bedeutet das Fehlen einer expliziten FROM-Klausel, daß die Transition aus *jedem* Hauptzustand heraus ausführbar ist und damit bezüglich der FROM-Klausel auch zu jeder anderen Transition paßt.




```
STATES S1, S2, S3;

INITIALIZE      TRANS           TRANS           TRANS
                   FROM S1         FROM S2         FROM S3
   TO S1           TO S2           TO S3           TO S1
   NAME T0:        NAME T1:        NAME T2:        NAME T3:
      BEGIN           BEGIN           BEGIN           BEGIN
      END;            END;            END;            END;
```

Abbildung 2.5: Zustandsübergangsgraph zu einer Transitionsmenge

Die TO-Klausel hat dagegen keinen unmittelbaren Einfluß auf die Schaltbarkeit der Transition. Lediglich durch den Übergangsgraphen der erreichbaren Hauptzustände, die von den FROM- und TO-Klauseln aller (ausführbarer) Transitionen des Modulrumpfs erzeugt wird, hat die TO-Klausel indirekt Einfluß auf die Ausführbarkeit von Transitionen. Wird zum Beispiel in Abbildung 2.5 die Transition T_1 aus irgendeinem Grund niemals schaltbar, so können die Zustände S_2 und S_3 nie erreicht werden und indirekt werden damit auch die Transitionen T_2 und T_3 nicht ausführbar.

Transitionen mit FROM-Klauseln, die nicht mindestens einen *erreichbaren* Hauptzustand beinhalten, sind daher niemals schaltbar. Dabei sind nur solche Hauptzustände erreichbar, die – ausgehend vom initialen Hauptzustand – transitiv durch (potentiell) ausführbare Transitionen erreicht werden können.

Es ist dabei zu beachten, daß die Analyse aufgrund der mangelnden Entscheidbarkeit mit den abgeschwächten Mengenprädikaten ***potentiell*** *erreich-*

bare Hauptzustände[15] und ***potentiell** schaltbare Transitionen* arbeitet. Zu Beginn der Analyse werden alle Hauptzustände und Transitionen als potentiell erreichbar eingestuft. Erst im späteren Verlauf werden diese Mengen so weit wie möglich verkleinert: Dabei werden alle Transitionen, die einmal als unerreichbar erkannt und daraufhin aus der Menge der potentiell erreichbaren Transitionen entfernt wurden, nie wieder in diese Menge aufgenommen. Folglich können bei den darauffolgenden erneuten Berechnungen[16] der daraus resultierenden Menge von potentiell erreichbaren Hauptzuständen keine neuen Elemente hinzukommen, d.h. auch diese Menge kann höchstens kleiner werden.

2.3.3 Nachrichtenbehandlung: Die `WHEN`-Klausel

Die WHEN-Klausel macht die Ausführbarkeit einer Transition davon abhängig, daß eine bestimmte Nachricht in einer bestimmten Eingangswarteschlange des Moduls vorhanden ist. Die WHEN-Klausel setzt sich aus mehreren (zum Teil optionalen) Elementen zusammen:

- Ein *Interaktionspunkt* oder ein *Array von Interaktionspunkten.*
- Ein oder mehrere Ausdrücke, die als *Indizes* für das Array genutzt werden (falls es sich im vorigen Punkt um ein Array von Interaktionspunkten handelt).
- Ein *Nachrichtentyp* (auch *Interaktionstyp* genannt).
- *Nachrichtenargumente.*

Da über die Verbindungsstrukturen und die Erzeugung von Nachrichten keine Informationen ermittelt werden, wird grundsätzlich davon ausgegangen, daß jede Nachricht auch erzeugt werden kann. Die WHEN-Klausel hat also in diesem Sinne keinen Einfluß auf die Analyse der *Erreichbarkeit* von Transitionen. Dagegen hat sie wesentlichen Einfluß auf mögliche Interaktionen zwischen Transitionen.

Indeterminismus zwischen zwei Transitionen kann durch WHEN-Klauseln nicht eingeschränkt werden, wenn **keine** oder nur **genau eine** der beiden Transitionen eine solche Klausel besitzt. Im zweiten Fall genügt es nämlich zur konkurrierenden Ausführbarkeit, daß neben den sonstigen Indeterminismusbedingungen durch eine entsprechende Nachricht auch die WHEN-Klausel erfüllt ist.

Haben jedoch *beide* Transitionen eine WHEN-Klausel, so gibt es Konstellationen, in denen diese beiden Klauseln niemals zugleich erfüllt werden können: Beziehen sich die beiden WHEN-Klauseln auf die selbe Warteschlange[17], so

[15] Dies ist eine Obermenge der **tatsächlich** erreichbaren Hauptzustände, welche selbst wiederum eine Projektion der Menge der erreichbaren Zustände darstellt.

[16] Die Menge der (potentiell) erreichbaren Hauptzustände wird jedesmal erneut als transitive Hülle der (potentiell) ausführbaren Hauptzustandsübergänge berechnet.

[17] Die Gleichheit der Warteschlangen impliziert nicht notwendigerweise die Gleichheit der IPs und umgekehrt!

können sie nur dann gemeinsam bereit werden, wenn sie am gleichen *Interaktionspunkt* mit den gleichen *Indexwerten* die gleiche *Nachricht* erwarten. Ist diese Anforderung verletzt, so können niemals beide WHEN-Bedingungen zugleich erfüllt werden, da die Estelle-Warteschlangen FIFO-Semantik haben und deshalb zu *einem* Zeitpunkt auch nur höchstens *eine* Nachricht bereitsteht.

Andererseits gibt es auch Konstellationen, in denen die beiden WHEN-Klauseln um genau die *selbe Nachricht* (also selber Nachrichtentyp, selber Interaktionspunkt und selbe Indizes) konkurrieren, so daß beide WHEN-Bedingungen *immer nur zugleich* erfüllt sind.

Die Unterscheidung der drei Fälle, daß die *gleichzeitige Erfüllung* der beiden WHEN-Bedingungen

- *unmöglich*
- *möglich aber nicht sicher*
- *sicher*[18]

ist, ist dabei insbesondere für die Bestimmung der **Überlappungsbedingungen** wichtig: potentielle Überlappungen können bereits *gemeldet* werden, wenn sie *möglich* (also **sicher** oder **unsicher**) sind. Sie dürfen jedoch nur dann auf die *effektive PROVIDED-Bedingung* der überlappten Transition Einfluß haben, wenn die Überlappung **sicher** ist.

In der effektiven PROVIDED-Bedingung gibt es keine Bezüge auf das Auftreten oder Nichtauftreten von Nachrichten. Eine *sichere potentielle Überlappung* ist immer gegeben, wenn die Erfüllung der evtl. abgeleiteten Bedingungen garantiert, daß die Überlappung auch eintritt. Bei einer *unsicheren potentiellen Überlappung* hängt dies von der Nachrichtensituation ab: Für die überlappende Transition muß in den entsprechenden Momenten auch immer eine Nachricht vorhanden sein, um ihre WHEN-Klausel zu erfüllen. Da man davon im allgemeinen nicht ausgehen kann, wird die mögliche Überlappung zwar gemeldet, jedoch hat sie keinen Einfluß auf die *effektive PROVIDED-Bedingung*

Ähnlich wird auch in den Fällen vorgegangen, in denen *nur eine* oder *keine* der Transitionen eine WHEN-Klausel hat: Solange die überlappende Transition keine WHEN-Klausel besitzt, wird die Überlappung nicht eingeschränkt. Besitzt jedoch nur die überlappende Transition eine solche Klausel, so ist die Überlappung nur noch dann gegeben, wenn die entsprechende Nachricht immer anliegt, sobald die überlappte Transition bereit werden kann (*unsichere potentielle Überlappung*).

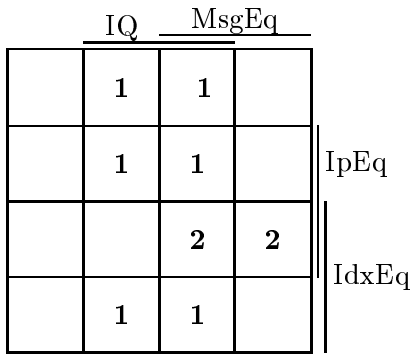
Vom Analysewerkzeug werden unsichere Überlappungsmeldungen nur optional ausgegeben und zudem auch explizit als „unsicher“ markiert.

Wie bestimmt man nun, welcher der oben genannten drei Fälle bei einem gegebenen Paar von WHEN-Klauseln vorliegt? Wie bereits angedeutet spielen dabei die folgenden Prädikate eine Rolle:

[18] „Sicher“ bedeutet nicht, daß eine der WHEN-Klauseln irgendwann erfüllt sein *muß*, sondern daß sie immer *nur gemeinsam* erfüllt sein können.

- Die Gleichheit der Interaktionspunkte (bzw. der Arrays von Interaktionspunkten).
- Die Gleicheit der Nachrichtentypen.
- Die paarweise Gleichheit der Indizes (bei zwei gleichen Arrays von Interaktionspunkten).[19]
- Die Zugehörigkeit zur gleichen Warteschlange.

Die ersten beiden Punkte sind anhand der WHEN-Klauseln unmittelbar entscheidbar. Die *paarweise Gleichheit der Indizes* läßt sich jedoch (bis auf Ausnahmen[20]) nur als Konjunktion von Gleichungen zwischen den zusammengehörigen Ausdrücken darstellen. Diese bildet dann ggfs. einen Teil der Interferenzbedingung der WHEN-Klausel. Auch ob zwei Interaktionspunkte der selben Warteschlange angehören, kann von der Gleichheit der Interaktionspunkt-Indizes abhängen: Sind Interaktionspunkte und Indizes jeweils gleich, so kann natürlich nur die selbe Warteschlange betroffen sein. Ansonsten sind die Warteschlangen nur dann gleich, wenn die Interaktionspunkte die Attributierung „`COMMON QUEUE`“ tragen.[21]



1, 2	gleichzeitige Erfüllung der WHEN-Klauseln *möglich*
2	gleichzeitige Erfüllung der WHEN-Klauseln *sicher*

IQ	mindestens ein Interaktionspunkt hat eine „`INDIVIDUAL QUEUE`“
MsgEq	gleiche Nachrichtentypen
IpEq	gleiche Interaktionspunkte (bzw. -arrays)
IdxEq	kein Array oder gleiche Indizes im Array

Abbildung 2.6: Möglichkeit von Interferenzen mit WHEN-Klauseln

In Abbildung 2.6 ist dargestellt, wann die gleichzeitige Erfüllung zweier WHEN-Klauseln *„sicher“* („2“), *„möglich aber nicht sicher“* („1“) oder *„unmöglich“* (leere Felder) ist.

[19] handelt es sich nicht um Arrays, so gilt die Gleichheit als erfüllt

[20] keine Arrays und damit auch keine Indexausdrücke oder beide WHEN-Klauseln enthalten Konstanten als Index

[21] globale Festlegung per „`DEFAULT COMMON QUEUE`“-Anweisung oder lokal bei der Interaktionspunktdefinition

Damit die gleichzeitige Erfüllung beider WHEN-Bedingungen *möglich* ist, muß einer der Fälle „1“ oder „2“ in dem Diagramm zutreffen. Es gibt dabei nur zwei Situationen, in denen die Indexausdrücke dabei von Bedeutung sind:

- Gleiche Interaktionspunkte mit „`INDIVIDUAL QUEUE`“-Attribut und verschiedene Nachrichtentypen: Interferenzen sind nur bei *unterschiedlichen* Indizes möglich.
- Gleiche Interaktionspunkte mit „`COMMON QUEUE`“-Attribut und gleiche Nachrichtentypen: Interferenzen sind nur bei *gleichen* Indizes möglich.

Außerhalb dieser beiden Fälle sind Interferenzen genau dann möglich, wenn mindestens einer der Interaktionspunkte das „`INDIVIDUAL QUEUE`“-Attribut hat.

Die *sichere* gleichzeitige Erfüllbarkeit beider WHEN-Bedingungen (Fall „2“) ist gegeben, wenn die Interaktionspunkte, Nachrichtentypen und Indexpaare jeweils gleich sind.

Zusammenfassend verläuft die Analyse der WHEN-Klauseln also folgendermaßen: Hat höchstens eine Transition eine WHEN-Klausel, so kann die Situation sofort entschieden werden: Indeterminismus wird nie eingeschränkt, Überlappungen sind immer möglich (und werden gemeldet), jedoch kann vom *sicheren* Vorliegen einer Überlappung (zur Bildung der effektiven PROVIDED-Bedingung der überlappten Transition) nur ausgegangen werden, wenn die überlappende Transition *keine* WHEN-Klausel hat.

Haben beide Transitionen WHEN-Klauseln, so muß geprüft werden, welcher der drei Fälle aus Abbildung 2.6 vorliegt und welche Bedingungen[22] mit ihm einhergehen: Für die Erzeugung von Indeterminismus- oder Überlappungsmeldungen genügt wiederum die *Möglichkeit*, also die Fälle „1“ oder „2“. Einen Einfluß auf die Bildung der effektiven PROVIDED-Bedingung haben die WHEN-Klauseln jedoch nur dann, wenn die Überlappung bei Erfüllung der entstehenden Bedingung *sicher* vorliegt (nur Fall „2“).

Im Beispiel aus Abbildung 2.7 sind zwei Transitionen mit WHEN-Klauseln dargestellt. Anhand des Diagramms in Abbildung 2.6 erkennt man leicht, daß eine Überlappung vorliegt, wenn die Indexausdrücke paarweise gleich sind, und daß diese Überlappung im obigen Sinne *sicher* ist. Dies liefert zusätzliche Gleichungen, mit denen die sonstigen Überlappungsbedingungen konjunktiv verknüpft werden. So läßt sich zur Überlappung von t2 über t1 die Bedingung $((a1 = a2) \wedge (b1 = b2))$ ableiten.

Es wird sich später zeigen, daß durch solche Gleichungen unter bestimmten Umständen einige Probleme beseitigt werden können, die durch den Einsatz von ANY-Klauseln entstehen (siehe Abschnitt 2.3.6).

[22] (Un-) Gleicheit von Indexausdrücken

```
IP ipx[ ... ]: ... COMMON QUEUE
...
CONST PrioHi = 1;
      PrioLo = 2;
VAR   a1, a2, b1, b2: INTEGER;

TRANS                          TRANS
   PRIORITY PrioLo                PRIORITY PrioHi
   WHEN ipx[a1, b1].msgx          WHEN ipx[a2, b2].msgx
   NAME t1:                       NAME t2:
      BEGIN                          BEGIN
      END;                           END;
```

Abbildung 2.7: IP-Indexgleichheit bei Überlappungen

2.3.4 Verzögerte Schaltbarkeit: Die DELAY-Klausel

DELAY-Klauseln dienen dazu, eine zeitliche Verzögerung zwischen dem Eintritt der (Schalt-) Bereitschaft und der Schaltbarkeit einer Transition zu realisieren. Dazu werden ein oder zwei Argumentausdrücke[23] angegeben, mit denen Zeiträume bestimmt werden, um die die Transition verzögert schaltbar bzw. nur optional schaltbar wird. Das Fehlen einer expliziten DELAY-Klausel setzt diese Verzögerung auf Null und ist damit äquivalent zur trivialen Klausel „DELAY(0,0)“.

```
VAR x: BOOLEAN;

TRANS                          TRANS
   PRIORITY 1                     PRIORITY 2
   PROVIDED x                     PROVIDED x
   DELAY(0,10)                    DELAY(0,10)
   NAME t1:                       NAME t2:
      BEGIN                          BEGIN
      x := FALSE                     x := FALSE
      END;                           END;
```

Abbildung 2.8: Indeterminismus durch optionale Schaltbarkeit

DELAY-Klauseln stellen ein wichtiges Mittel zur Realisierung von (erwünschtem) Indeterminismus dar. Zum einen ist im Zeitraum mit optionaler Schaltbarkeit nicht festgelegt, ob und wann die Transition tatsächlich schaltbar ist. So

[23] Da für die verschiedenen syntaktischen Varianten jeweils eine Interpretation mit zwei INTEGER-Ausdrücken (zuzüglich ∞) angegeben werden kann, wird im folgenden nur das Format DELAY(*expr1, expr2*) behandelt.

ist es durchaus denkbar, daß in der Implementation einer Spezifikation aus zwei optional schaltbaren Transitionen diejenige mit geringerer Priorität ausgewählt wird (in Abbildung 2.8 ist `t2` daher ausführbar)!

Zudem ist die effektive Dauer eines Delay-Zeitraums in Bezug auf die Dauer des Auswahlverfahrens und die Zeit bis zum Eintritt der Schaltwirkung ebenfalls unbekannt. So ist z.B. in Abbildung 2.9 anhand der Spezifikation nicht entscheidbar, welche der beiden Transitionen überhaupt ausführbar sind. Ohne seine DELAY-Klausel würde `t1` die andere Transition vollständig überlappen. Mit der Verzögerung ist das Verhalten jedoch unklar: Schreitet die (Delay-) Zeit im Verhältnis zur Dauer bis zur nächsten Auswahlphase sehr schnell voran, so kann t2 immer noch durch t1 überlappt werden; schreitet diese Zeit sehr langsam voran, so kann umgekehrt sogar t1 durch t2 unerreichbar werden (wenn `t2` ausgeführt wird, so wird der Timer von `t1` ebenfalls zurückgesetzt). Im allgemeinen besteht dadurch möglicher Indeterminismus in der Auswahl der zu schaltenden Transition.

```
VAR x: BOOLEAN;

TRANS                          TRANS
   PRIORITY 1                     PRIORITY 2
   PROVIDED x                     PROVIDED x
   DELAY(10,10)                   DELAY(5,5)
   NAME t1:                       NAME t2:
      BEGIN                          BEGIN
      x := FALSE                     x := FALSE
      END;                           END;
```

Abbildung 2.9: Indeterminismus durch unbekannte Zeitskalierung

Aber auch in den Fällen, in denen keine optionale Schaltbarkeit vorliegt und auch die Delay-Zeit der Transition mit höherer Priorität kürzer oder gleich der Delay-Zeit der anderen Transition ist, kann nur selten eine sichere Aussage über eine Überlappung gemacht werden. In den obigen Beispielen hatten die Transitionspaare jeweils äquivalente DELAY- und FROM-Klauseln, zudem wurde durch die Ausführung einer Transition eine der Schaltbedingungen ungültig. Dadurch werden die Delay-Timer der Transitionen immer zugleich gestartet und beendet. Ist dies nicht mehr der Fall (wie z.B. in Abbildung 2.10), so können die Timer unter Umständen unabhängig ablaufen, es sind keine Aussagen über Überlappungen mehr möglich. (In dem Beispiel feuern beide Transitionen unabhängig voneinander immer wieder, da keine Beziehung zwischen den Steuerungen der beiden Timer besteht.)

Da bei der Analyse jedoch keine Informationen über die Aktivitäten im Transitionsblock gesammelt werden, kann im allgemeinen auch keine Aussage über das Zusammenspiel der DELAY-Timer gemacht werden. Hat lediglich die Transition mit niedrigerer Priorität eine DELAY-Klausel, so beeinträchtigt diese in

```
TRANS                          TRANS
   PRIORITY 1                     PRIORITY 2
   DELAY(7,7)                     DELAY(10,10)
   NAME t1:                       NAME t2:
      BEGIN                          BEGIN
      END;                           END;
```

Abbildung 2.10: Keine Überlappungsmöglichkeit durch asynchronen Timer-Ablauf

keinem Fall die Überlappung. Hat jedoch die Transition mit höherer Priorität (oder beide Transitionen) eine nichttriviale[24] DELAY-Klausel, so tritt die Überlappung nur noch sehr bedingt auf. Bei der Analyse kann sie grundsätzlich nur dann sicher erkannt werden, wenn folgende Punkte erfüllt sind:

- Das Feuern der Transition mit höherer Priorität setzt in jedem Fall den Timer der anderen Transition zurück, indem deren Schaltbereitschaft beendet wird. Dies kann nur bei entsprechenden FROM- und TO-Klauseln erkannt werden. Ein Rücksetzen des Timers durch die PROVIDED-Klausel kann nur im Transitionsblock initiiert werden und ist daher in der Analyse nicht erkennbar.
- Der Timer der Transition niederer Priorität wird nie vor dem mit höherer Priorität gestartet. Das bedeutet, daß die lokalen Schaltbedingungen der Transition niederer Priorität die der anderen Transition *implizieren* müssen.
- Der Timer der Transition mit niedrigerer Priorität läuft nie kürzer als der der anderen Transition. Dies kann nur erkannt werden, wenn die Auswertung der DELAY-Argumente Konstanten ergeben oder ein symbolischer Vergleich über $>$ möglich ist.

Aus diesen Gründen können im Zusammenhang mit DELAY-Klauseln oft nur **sehr schwache Aussagen** zu Überlappungen von Transitionen gemacht werden. Insbesondere kann an dieser Stelle, wie im obigen Beispiel gezeigt wurde, gleichzeitig möglicher Indeterminismus *und* eine mögliche Überlappung zwischen zwei Transitionen auftreten. (Natürlich kann in der konkreten Implementation der Spezifikation höchstens einer der beiden Fälle eintreten.)

Daher werden beim Vergleich von Transitionen mit DELAY-Klauseln folgende Konventionen zugrundegelegt:

- Mögliche Überlappungen werden nur dann betrachtet, wenn höchstens *die überlappte* Transition eine DELAY-Klausel hat.

[24] nicht DELAY(0,0) bzw. keine DELAY-Klausel

- Indeterminismus, der ohne Delay-Klauseln bestehen würde, wird durch diese niemals eingeschränkt. So kann aufgrund der unbekannten Zeitskala auch ein noch so großer Delay-Wert nicht sicher verhindern, daß zwei Transitionen nicht doch konkurrierend zueinander schaltbar werden. Daher wird auch die Ausgabe entsprechender Meldungen durch diese Klausel nicht eingeschränkt.
- Eine Transition mit nichttrivialer DELAY-Klausel kann Indeterminismus zu einer Transition *geringerer Priorität* haben, es können jedoch auch Überlappungen in beide Richtungen vorliegen, wie in den obigen Beispielen gezeigt wurde. Aber auch bei gleicher Priorität sind im Zusammenhang mit Delay-Klauseln Überlappungen möglich. Da darüber nur sehr schwache Aussagen gemacht werden können, wird vom Analysewerkzeug dazu nur optional eine entsprechende Indeterminismus- oder Überlappungsmeldung ausgegeben.

2.3.5 Boolsche Ausdrücke als Schaltbedingung: Die `PROVIDED`-Klausel

Mit Hilfe der PROVIDED-Klausel kann die Schaltbereitschaft einer Transition vom Ergebnis eines Ausdrucks mit dem Ergebnistyp `BOOLEAN` abhängig gemacht werden. Die Transition kann nur dann bereit bzw. schaltbar werden, wenn die Auswertung `TRUE` ergibt. Das Fehlen einer expliziten PROVIDED-Klausel ist dabei äquivalent zur Klausel „`PROVIDED TRUE`“. Eine weitere Form ist „`PROVIDED OTHERWISE`“. Dies ist eine Abkürzung für eine PROVIDED-Klausel, die einer Negation der Konjunktion bestimmter anderer PROVIDED-Klauseln entspricht. Im folgenden wird davon ausgegangen, daß die Klausel entsprechend expandiert wurde.[25]

```
TRANS                          TRANS
   PROVIDED A                     PROVIDED B
   NAME t1:                       NAME t2:
      BEGIN                          BEGIN
      END;                           END;
```

Abbildung 2.11: Interferenzbedingungen aus PROVIDED-Klauseln

Zwei PROVIDED-Klauseln passen zueinander, wenn die Konjunktion ihrer Bedingungen erfüllbar ist. Im Sinne der offensiven Meldestrategie genügt es dabei, wenn der Ausdruck nicht als unerfüllbar bewiesen werden kann. Diese Konjunktion der Bedingungen liefert dabei auch bereits die Bedingung für den Potentiellen Indeterminismus bzw. für die potentielle Überlappung der beiden Transitionen. Unmittelbare Interferenzen zwischen zwei Transitionen können nämlich

[25] Im Abschnitt 2.5 wird noch einmal genauer auf diese Form eingegangen.

nur in den Situationen bestehen, in denen beide PROVIDED-Bedingungen zugleich erfüllt sind.

Aus dem Beispiel in Abbildung 2.11 kann daher die Bedingung $A \wedge B$ für eine Interferenz[26] zwischen `t1` und `t2` abgeleitet werden. Ist diese Bedingung im Rahmen des erreichbaren Zustandsraums nicht erfüllbar, so besteht zwischen den beiden Transitionen auch keine direkte Interferenz. Kann die Unerfüllbarkeit nicht nachgewiesen werden, so muß eine (potentielle) Interferenz mit eben dieser Interferenzbedingung angenommen werden.

Die PROVIDED-Klausel bildet auch die Grundlage zur Bildung der *effektiven PROVIDED-Klausel* für die Transition. In dieser werden die Bedingungen gesammelt, unter der die Transition *tatsächlich* ausgeführt werden kann, also ohne von einer anderen Transition überlappt zu werden. Dazu wird die PROVIDED-Bedingung jeder passenden Transition mit höherer Priorität invertiert und dann mit der effektiven PROVIDED-Klausel konjugiert.

```
VAR x, y, z: INTEGER;
TRANS                TRANS                TRANS
   PRIORITY 1           PRIORITY 1           PRIORITY 2
   PROVIDED x ≤ y       PROVIDED y ≤ z       PROVIDED z > x
   NAME t1:             NAME t2:             NAME t3:
      BEGIN                BEGIN                BEGIN
      END;                 END;                 END;
```

Abbildung 2.12: Vollständige Überlappung von `t3`

Im Beispiel in Abbildung 2.12 wird die Transition `t3` durch die beiden anderen Transitionen jeweils partiell überlappt. Die abgeleiteten Bedingungen lauten dabei:

Indeterminismus zwischen	`t1` und `t2`	falls $(x \leq y) \wedge (y \leq z)$
Überlappung von	`t1` über `t3`	falls $(x \leq y) \wedge (z > x)$
Überlappung von	`t2` über `t3`	falls $(y \leq z) \wedge (z > x)$

Diese Bedingungen sind nicht widerlegbar, es werden also die entsprechenden Meldungen über den möglichen Indeterminismus und die möglichen Überlappungen erzeugt. Die aus der Überlappung durch `t1` und `t2` resultierende effektive PROVIDED-Klausel von `t3` lautet

$$(z > x) \wedge \underbrace{\neg(x \leq y)}_{=(x>y)} \wedge \underbrace{\neg(y \leq z)}_{=(y>z)}$$

und ist damit unerfüllbar, da über die Transitivität $x > x$ daraus ableitbar ist. Die Transition wurde daher vollständig überlappt und ist nicht mehr ausführbar.

[26] Da die Prioritäten gleich sind, kann es nur Indeterminismus sein (s.o.)

Aufbauend auf der effektiven PROVIDED-Klausel wird später die *erweiterte* effektive PROVIDED-Klausel eingeführt, mit der zusätzlich noch Hauptzustände eingebettet werden können. Genaueres dazu und zu der Technik der Erkennung widersprüchlicher Aussagen folgt in Abschnitt 2.4.

2.3.6 Quantifizierung von Transitionen: Die ANY-Klausel

Die ANY-Klausel erlaubt es, ein ganzes Bündel von Transitionen gleichzeitig zu definieren. Dazu werden innerhalb dieser Klausel eine oder mehrere ANY-Variablen über Teilbereichs- bzw. Aufzählungstypen definiert. Die Semantik dieses Konstrukts entspricht der einer Menge von normalen Transitionen (die **Instanzen** der ursprünglichen Transition), bei denen zu jeder möglichen Wertekombination der ANY-Variablen Konstanten eingesetzt wurden.

In der Praxis vermeiden die meisten Estelle-Compiler diese Expansion jedoch und implementieren statt dessen nur *eine* (parametrierbare) Transition, die dann durch verschachtelte Schleifen mehrfach zum Einsatz kommt. Der Grund dafür ist, daß ansonsten häufig extrem viele[27] Instanzen der Transition entstehen können. Daher wird auch bei der Analyse aus Effizienzgründen auf eine Expansion verzichtet. Die andernfalls resultierende große Zahl von Transitionen würde zudem bei der Analyse unter Umständen eine unüberschaubare Flut von Interferenzmeldungen verursachen.

Zum Analysezeitpunkt kommt beim Versuch einer Expansion oft noch ein zweites Problem hinzu: In vielen Fällen wird über ANY-quantifizierte Transitionen die n-fache Replikation einer Funktionalität realisiert, z.B. n getrennte Kanäle zur Datenübertragung in Form eines Arrays von Interaktionspunkten. Dabei spielt der konkrete Wert von n für die Spezifikation zunächst keine Rolle, und die Konstante wird in der Spezifikation nur *unvollständig* definiert[28] (siehe Beispiel in Abbildung 2.13). Natürlich ist es so nicht mehr möglich, allein anhand der Spezifikation diese Transition zu expandieren, um die ANY-Klausel zu eliminieren.

Aus diesen Gründen wird im Analysewerkzeug auf die Expansion ANY-quantifizierter Transitionen verzichtet und die Behandlung der Klausel erfolgt symbolisch. Zudem werden nur solche Überlappungen betrachtet, die *alle Instanzen* der Transition zugleich betreffen. Dies führt leider jedoch zu einigen Einschränkungen bei der Analyse.

Der eigentliche Zweck der ANY-Klauseln liegt – wie bereits erläutert – darin, ein adäquates Mittel zum Zugriff auf ein *Array von Interaktionspunkten* innerhalb der WHEN-Klausel zu ermöglichen. Dazu werden die ANY-Variablen direkt als Index in diesem Array benutzt, wodurch jedem der einzelnen Interaktionspunkte eine eigene Instanz zugeordnet wird. Gibt es dann eine Überlappung zwischen zwei derart definierten Transitionen mit äquivalenten WHEN-Klauseln, so überlappen sich jeweils die zueinander gehörigen Instanzen der

[27] $O(e^n)$ bezüglich der Anzahl der ANY-Variablen

[28] Erst bei der Implementation einer solchen Spezifikation müssen dann konkrete Werte für die unvollständig spezifizierten Konstanten festgelegt werden.

```
CONST NumberOfChannels = ANY INTEGER;
TYPE ChannelRange = 1 .. NumberOfChannels;
....
IP IPChannels: ARRAY [ChannelRange] OF ....
....
TRANS
   ANY Channel: ChannelRange
   WHEN IPChannels[Channel].frame
      BEGIN
      END;
```

Abbildung 2.13: Nicht expandierbare ANY-Transition

Transitionen paarweise. So kann in Abbildung 2.14 die Überlappung von `t2` über `t1` erkannt werden, da offensichtlich jede *Instanz* von `t1` von einer Instanz von `t2` überlappt wird. Die Zuordnung der einzelnen Instanzen erfolgt über die Interaktionspunkt-Indizes, da diese jeweils identisch sein müssen (siehe Abschnitt 2.3.3).

```
TYPE IdxRange = 0..9;
IP IpArr: ARRAY [IdxRange] ....;

TRANS                TRANS                TRANS
   PRIORITY 2           PRIORITY 1           PRIORITY 1
   ANY i1: IdxRange     ANY i2: IdxRange     ANY i3: IdxRange
   WHEN IpArr[i1].msg   WHEN IpArr[i2].msg   WHEN IpArr[(i3+5) MOD 10].msg
   NAME t1:             NAME t2:             NAME t3:
      BEGIN                BEGIN                BEGIN
      END;                 END;                 END;
```

Abbildung 2.14: Analysierbare (`t2`/`t1`) und nicht analysierbare (`t3`/`t1`) Überlappungen bei ANY-Klauseln

Die Überlappung von `t3` über `t1` ist dagegen nicht ohne weiteres erkennbar: Nur die Tatsache, daß `((i3+5) MOD 10)` eine *bijektive* Abbildung auf `IdxRange` ist, bedingt, daß auch hier jede Instanz von `t1` durch eine Instanz von `t3` überlappt wird. In der Analyse ist diese Bedingung jedoch nicht allgemein überprüfbar[29]. Die Überlappung einer Transition mit einer ANY-Klausel darf jedoch nur dann angenommen werden, wenn *alle Instanzen* überlappt wurden.

Hat also eine Transition eine ANY- und eine WHEN-Klausel, und werden durch ihre Instanzen mehrere Interaktionspunkte referenziert[30], so besteht eine

[29] Ersetzt man z.B. `((i3+5) MOD 10)` durch `((i3*5) MOD 10)`, so werden nur noch zwei der zehn Instanzen (nämlich die zu $i_1 \in \{0, 5\}$) überlappt.

[30] Die ANY-Variable muß also Einfluß auf die Interaktionspunkt-Indizes haben (siehe obige

Überlappung aller ihrer Instanzen nur dann, wenn die überlappende Transition entweder *spontan* ist oder wenn sie die gleichen Interaktionspunkte und Interaktionspunktindizes[31] hat. Insbesondere muß im zweiten Fall die überlappende Transition ebenfalls eine ANY-Klausel haben[32].

Beim Analysewerkzeug wurden die Überlappungen im Zusammenhang mit ANY-Klauseln daher auf die folgenden Bedingungen eingeschränkt:

- Eine Überlappung zwischen zwei Transitionen mit ANY- und WHEN-Klauseln wird nur dann angenommen, wenn für die jeweils zueinander gehörigen Interaktionspunkt-Indizes gilt, daß
  - beides *ANY-Variablen mit gleichem Wertebereich* sind oder
  - beide *unabhängig von allen ANY-Variablen* sind.

  Insbesondere können als Index bei der *überlappenden* Transition keine zusammengesetzten Ausdrücke mit ANY-Variablen analysiert werden (siehe obiges Beispiel).[33]
- ANY-Variablen, die nicht als Interaktionspunkt-Index in der WHEN-Klausel benutzt werden, werden wie ANY-Konstanten (z.B. „ANY INTEGER“) behandelt.
- Ist die überlappende Transition spontan, so werden in beiden Transitionen *alle* ANY-Variablen wie ANY-Konstanten behandelt, auch wenn diese in der überlappten Transition als Interaktionspunkt-Index in einer WHEN-Klausel dienen.

Die Idee hinter diesen Anforderungen ist, die Bindung der ANY-Variablen durch die Überlappungsbedingungen der WHEN-Klauseln auszunutzen. Ist dies nicht möglich, so wird die ANY-Variable wie eine ANY-Konstante behandelt: Gibt es keine Belegung der ANY-Konstanten, die die effektive Schaltbedingung der überlappten Transition erfüllen kann, so wäre auch keine der zugehörigen Instanzen ausführbar.

Durch diese Anforderung alleine wären jedoch bestimmte Fälle nicht zu entscheiden wie das Beispiel[34] in Abbildung 2.15 zeigt. Bei genauerer Betrachtung erkennt man, daß `t2` von `t1` vollständig überlappt wird, zwischen `t3` und den beiden anderen Transitionen jedoch weder Überlappung noch Indeterminismus

Beispiele).

[31] oder eine Obermenge der Indexwerte abdeckt

[32] Ausnahme: Pathologische Fälle, z.B. eine ANY-Variable mit nur einem einzigen Wert als Definitionsbereich kann natürlich auch durch eine Transition ohne ANY-Klausel überlappt werden.

[33] Zusammen mit Informationen über die *Wertebereiche der ANY-Variablen* und die *Indexbereiche der Interaktionspunkt-Indizes* sind weitere Fälle denkbar, bei denen eine Überlappung erkannt werden könnte. Diese werden jedoch in der Implementation des Analysewerkzeugs noch nicht untersucht.

[34] Die Vorlage zu dem Beispiel ist die Spezifikation aus [BrGo94a].

besteht. Der Grund dafür ist, daß wegen der WHEN-Klauseln nur diejenigen Instanzen miteinander interferieren können, die die gleichen Werte in ihren ANY-Variablen haben. Werden die ANY-Variablen jedoch unabhängig voneinander behandelt, so kann die Überlappung nicht sicher festgestellt werden, da damit die Information verloren geht, daß in der Provided-Klausel jeweils das gleiche Array-Element von `Status` referenziert wird. Erst durch die Identifikation der zusammengehörigen ANY-Variablen anhand ihres Einsatzes in der WHEN-Klausel kann erkannt werden, daß in jedem Paar von bzgl. der WHEN-Klausel zusammengehörigen Instanzen jeweils das gleiche Array-Element referenziert wird (z.B. $\mathtt{i}_1$ und $\mathtt{i}_3$ bei $\mathtt{Status[i}_1\mathtt{]}$ und $\mathtt{Status[i}_3\mathtt{]}$).

```
CONST IdxMax =   ANY INTEGER;
TYPE  IdxRange = 1 .. IdxMax;
      StatusTp = (passiv, aktiv);
VAR   Status :   ARRAY [IdxRange] of StatusTp;
IP IpIn: ARRAY [IdxRange] .... COMMON QUEUE;

TRANS                TRANS                TRANS
   PRIORITY 1           PRIORITY 2           PRIORITY 2
   ANY i1: IdxRange     ANY i2: IdxRange     ANY i3: IdxRange
   WHEN IpIn[i1].msg    WHEN IpIn[i2].msg    WHEN IpIn[i3].msg
   PROVIDED Status[i1]  PROVIDED Status[i2]  PROVIDED Status[i3]
            = aktiv              = aktiv              = passiv
   NAME t1:             NAME t2:             NAME t3:
      BEGIN                BEGIN                BEGIN
      END;                 END;                 END;
```

Abbildung 2.15: Bindung von ANY-Variablen durch WHEN-Klauseln

Bei der Bestimmung von *Indeterminismus* zwischen Transitionen wird ähnlich vorgegangen. Hier genügt es jedoch bereits, wenn zwischen *einem* Paar von Transitionen Indeterminismus möglich ist. Daher kann die ANY-Klausel vollständig als ANY-Konstante behandelt werden. Man beachte in diesem Zusammenhang, daß durchaus auch *zwischen Instanzen einer Transition* Indeterminismus bestehen kann. Da die Instanzen jedoch alle zum selben Feature gehören, spielt dies bei der Analyse keine Rolle.

2.3.7 Formulierung von Zusicherungen

Bei den bisherigen Überlegungen wurde immer von *erreichbaren Zuständen* gesprochen, unter denen die Schaltbarkeit der Transitionen betrachtet wird. Welche Zustände jedoch erreichbar und welche unerreichbar sind, kann in der Analyse nicht sicher vollständig bestimmt werden, da dies eine Erkennung aller möglichen Zustandsverläufe der Spezifikation voraussetzt und damit prinzipiell nicht lösbar ist.

Somit muß jeder Zustand als erreichbar und jede Zustandsbedingung als potentiell erfüllbar gelten, solange nicht das Gegenteil bewiesen werden kann. Leider

schränkt dies den Nutzen der Analyse massiv ein, denn in vielen Fällen werden mögliche Interferenzen gemeldet, obwohl die dazu notwendigen Bedingungen zur Laufzeit der Spezifikation nie auftreten, oder es werden Überlappungen nicht als vollständig erkannt, obwohl die ermittelte effektive Schaltbedingung in Wirklichkeit nie erfüllt werden kann.

Viele dieser Bedingungen beruhen auf Zusammenhängen, die zwar dem Autor der Spezifikation bewußt sind, jedoch nicht explizit formuliert wurden und daher bei der Analyse oft auch nicht erkannt werden können. So werden häufig bei der Entwicklung der Spezifikation Integritätsbedingungen berücksichtigt, die nur schwer in Estelle explizit zu formulieren sind[35]. Ebenso werden (Programm-, Prozedur- oder Schleifen-) Invarianten zwar meist bewußt berücksichtigt, aber meist nicht explizit formuliert, da in Estelle weder die Notwendigkeit dafür besteht noch die nötige Ausdrucksmöglichkeit vorliegt.

Als einfaches Beispiel dafür existiere zwischen den Inhalten zweier Variablen implizit immer die feste Beziehung $x > y$, da die Variablen entsprechend initialisiert werden und alle Wertzuweisungen diese Beziehung erhalten. Die irgendwann ermittelte Interferenzbedingung $(x = z) \wedge (y = z)$, die in Widerspruch zu dieser Invariante steht, könnte (in Unkenntnis der obigen Beziehung) als erfüllbar eingestuft werden.

Aber auch andere Schaltbedingungen können betroffen sein: So kann die Erreichbarkeit eines bestimmten Hauptzustandes oder der mögliche Empfang einer bestimmten Nachricht durchaus von *impliziten Bedingungen* abhängen. So könnte zum Beispiel an einer Dienstschnittstelle die Konvention bestehen, daß der Dienstnutzer (gesteuert durch eine Sendekreditvergabe) nur dann neue Daten liefert, wenn der Diensterbringer noch freie Puffer besitzt. Bei konformem Verhalten des Dienstnutzers wäre die Datenannahme-Transition nur dann ausführbar, wenn die Anzahl der freien Puffer größer Null wäre. Diese implizite Schaltbedingung ergibt sich aus dem Zusammenspiel mehrerer Protokollautomaten und ist nur sehr schwer automatisch zu erkennen.

Solche Konstellationen treten relativ häufig auf und äußern sich in Indeterminismus- und Überlappungsmeldungen, die vom Benutzer bei genauerer Betrachtung als unerfüllbar erkannt werden. Es wäre außerordentlich nützlich, wenn es in solchen Situationen möglich wäre, derartige implizite Invarianten dem Analysewerkzeug als boolwertige Ausdrücke (im folgenden **Zusicherungen** oder **Assertions** genannt) explizit angeben zu können. Entsprechend der Beschränkung der Analyse auf die *Schaltbarkeit* von Transitionen werden im folgenden lediglich Zusicherungen betrachtet, die für dem Zeitpunkt der *Transitionsauswahl* gelten. Dabei sollten die Zusicherungen einerseits zur besseren Handhabung direkt im Quelltext der Spezifikation gemacht werden können, andererseits muß ihre Formulierung in Hinsicht auf Syntax und Semantik der Spezifikation *Kommentarcharakter* haben, da sie ja lediglich zur Unterstützung des Analysewerkzeugs dient und auch andere Werkzeuge nicht beeinträchtigt werden dürfen.

[35] Es können lediglich über Teilbereichstypen die Ober- und Untergrenzen des zulässigen Wertebereichs angegeben werden.

Eine naheliegende Lösung dafür wäre, diese Zusicherungen als qualifizierten Kommentar (z.B. „`{@assert: (x>y) AND (a<=17)}`“) abzulegen, zumal unter PET-DINGO ebenfalls ähnliche Kommentare eingesetzt werden[36]. Dieses Verfahren führt jedoch zu zwei Problemen: Zum einen gibt es **keine Namenskonventionen**, die bei der Markierung solcher qualifizierter Kommentare genutzt werden können. Dies führt dazu, daß andere Werkzeuge möglicherweise eine ähnliche Kennung benutzen und es damit zu Konflikten bezüglich der Syntax und der Semantik der Pseudokommentare kommt. Weitaus schwerwiegender ist jedoch das Problem, daß dieser Zusicherungsausdruck von PET (durchaus korrekt) als normaler Kommentar behandelt wird und daher zur Laufzeit des Analysewerkzeugs als *String* vorliegt. Die in den Zusicherungen aufgeführten Konstanten, Variablen, Funktionen und sonstigen Objekte sind also nur nach ihrem *Namen* und nicht nach ihrer **eindeutigen Objekt-ID**[37] bekannt. Aber auch das *Scoping* ist nach dem PET-Compilerlauf verloren, d.h. es gibt keine Informationen mehr darüber, welche Deklarationen an welcher Stelle sichtbar sind. Dadurch ist es sehr schwierig, zur Laufzeit des Analysewerkzeugs noch festzustellen, was z.B. mit dem o.g. Namen „x“ tatsächlich gemeint ist.

Als Alternative dazu könnten die Zusicherungen einfach konjunktiv verknüpft in die PROVIDED-Klausel der jeweiligen Transition aufgenommen werden. Dadurch könnten die impliziten Schaltbedingungen explizit angegeben und so auch dem Analysewerkzeug bekannt gemacht werden. Dies würde zu keiner Semantikänderung führen, da die Zusicherungen beim Schalten der Transition per Definition wahr sein müssen[38]. Jedoch würde sich zum einen aufgrund der fehlenden Unterscheidbarkeit zwischen den (notwendigen) PROVIDED-Bedingungen und den ausformulierten Zusicherungen nicht gerade die Lesbarkeit der Spezifikation erhöhen, zum anderen würde i.a. dadurch die Laufzeiteffizienz von Implementationen nachhaltig beeinträchtigt werden, da die Zusicherungen zur Laufzeit immer wieder ausgewertet werden müßten, ohne jemals Einfluß auf die Schaltbarkeit zu nehmen. Nur bei Fehlern in der Spezifikation oder den Zusicherungen würde sich ein Unterschied ergeben. Die Erkennung dieser Fehler wird durch diese Methode i.a. jedoch nicht erleichtert.

Wir haben jedoch eine elegante Alternative entwickelt, durch die die obigen Nachteile umgangen werden können: Zusicherungen werden als **Estelle-Funktionen** definiert, die eine leere Argumentliste und den Rückgabetyp `BOOLEAN` haben. Diese Funktionen müssen nirgends benutzt werden (es spricht jedoch auch nichts dagegen), einzig ihre Definition dient als logischer Zusicherungsausdruck. Sofern diese Funktionen so formuliert sind, daß ihr Rückgabe-

[36] Um bei der Modulinstanziierung den Knoten angeben zu können, auf dem der entsprechende Prozeß ablaufen soll.

[37] In der PET-Klassenbibliothek dient der Zeiger auf die Objektdeklaration als eindeutige Objekt-ID.

[38] Lediglich im Zusammenhang mit dem Startzeitpunkt von DELAY-Timern (dies geschieht i.a. *vor* dem Zeitpunkt des Feuerns der Transition) sind Konstellationen denkbar, in denen dieses Verfahren zu Semantikänderungen führen kann. Die unten definierten Zusicherungen gelten für den *Schaltzeitpunkt* und müssen nicht während der gesamten Bereitschaft erfüllt sein.

wert als geschlossener Ausdruck[39] dargestellt werden kann, kann aus ihnen die gewünschte Zusicherungsbedingung als ein Ausdruck gewonnen werden, in dem alle Objektreferenzen als eindeutige IDs vorliegen. Insbesondere können diese Funktionen auch zwischen den Transitionsklauseln und dem Transitionsblock definiert werden, so daß alle Objekte, auf die in einer PROVIDED-Klausel Bezug genommen werden kann, im Funktionskörper ebenfalls referenziert werden können. So können z.B. auch die Konstanten aus ANY-Klauseln oder Argumente von Nachrichten genutzt werden.

Syntaktisch werden die Zusicherungsfunktionen durch ihre Position und ihren Namen gekennzeichnet:

- Der Header der Zusicherungsfunktionen muß folgendermaßen lauten:

  „`FUNCTION trans_assertion:BOOLEAN;`“ (innerhalb von Transitionen)
  „`FUNCTION body_assertion:BOOLEAN;`“ (unmittelbar im Modulrumpf)

- Eine Zusicherungsfunktion *innerhalb* einer Transitionen gilt *nur für diese* (siehe Abbildung 2.16).

- Die Zusicherungsfunktion im Modulrumpf (*vor* allen Transitionsdefinitionen) gilt für *alle* Transitionen des Moduls (bis auf die Initialisierungstransition). Werden in den Transitionen zusätzlich noch lokale Zusicherungen formuliert, so gelten jeweils beide gemeinsam (konjunktiv verknüpft).

Die **Semantik der Zusicherungsfunktionen** ist sehr einfach: Würde die Funktion zu Beginn des Transitionsblocks aufgerufen werden, so müßte sie garantiert `TRUE` zurückliefern! Dies bedeutet, daß die (effektive) Provided-Klausel mit dem durch diese Funktion definierten Ausdruck konjugiert werden kann, ohne die Semantik der Transition zu ändern[40]. Und genau diese Konjunktion wird bei der Analyse gebildet, um die PROVIDED-Klausel mit den nun ausformulierten impliziten Bedingungen der Zusicherung zu verschärfen.

Die oben genannte Anforderung an die Semantik der Zusicherungsfunktion ist essentiell: Gibt es eine Situation, in der die Zusicherungsfunktion `FALSE` zurückliefert, obwohl die Transition ausgeführt wird, so ist die Analyse wahrscheinlich ebenfalls nicht zutreffend. Liefert zum Beispiel eine Zusicherungsfunktion immer `FALSE` zurück, so wird die effektive PROVIDED-Klausel ebenfalls zu `FALSE` reduziert und die Transition wird als unerreichbar gekennzeichnet, gleichgültig, ob das zutreffend ist oder nicht.

Da die Korrektheit der Zusicherungsfunktionen so wichtig ist, ist es sinnvoll, diese testen zu können. Dabei muß für jede Zusicherung die Einhaltung der o.g. semantischen Anforderungen überprüft werden. Die Erzeugung eines Testprogramms ist dabei sehr einfach automatisch möglich, da dazu lediglich zu Beginn

[39] zur Funktionsexpansion siehe Abschnitt 2.4.3

[40] Bis auf einige (pathologische) Situationen mit DELAY-Klauseln (s.o.); spielt bei der Analyse keine Rolle (siehe Abschnitt 2.3.4).

```
VAR x: INTEGER;
    y: ARRAY [1..10] OF INTEGER;

TRANS
   ANY i:1..3
   WHEN ipx[i].msgx(a, b)
   FUNCTION trans_assertion:BOOLEAN;
      BEGIN
      IF (x >= 1) AND (x <= 10) THEN
         trans_assertion := (a>b) AND (b>y[x])
      ELSE
         trans_assertion := TRUE
      END;
   NAME test:
      BEGIN
      ....
      END
```

Abbildung 2.16: Beispiel für eine Zusicherungsfunktion in einer Transition

jedes Transitionsblocks die Zusicherungsfunktion aufgerufen und der Rückgabewert geprüft werden muß. Wird irgendwann einmal ein `FALSE` zurückgegeben, so ist die Zusicherungsfunktion unzutreffend: Entweder die Funktion wurde falsch formuliert, oder es existieren andere Fehler in der Spezifikation, so daß die intendierten Integritätsbedingungen bei der Ausführung der Transition verletzt wurden.

Die Zusicherungsfunktionen liefern also neben der Unterstützung des Analysevorgangs auch noch eine Möglichkeit, Integritätsbedingungen explizit zu formulieren und während der Testphase automatisch zu prüfen. Man beachte dabei die unterschiedliche Nutzung der Funktionen innerhalb der (statischen) Analyse und während des (dynamischen) Tests!

Die Realisierung dieser Testfunktionalität ist durch eine relativ geringfügige Modifikation von DINGO möglich. Außerhalb der Testphase führen die Zusicherungen jedoch zu keinerlei Laufzeitnachteilen, da der Testcode vollständig entfernt werden kann.[41]

2.4 Aufbereitung der Vergleichsergebnisse

Aus dem oben geschilderten Vergleich einzelner Transitionsklauseln werden neben der Entscheidung, ob zwei Transitionen überhaupt zueinander passen, auch eine Fülle von Informationen für die Art der Interferenzen abgeleitet. Es soll

[41] Da diese Erweiterung nicht zum eigentlichen Ziel dieser Arbeit gehört, wurde auf eine entsprechende Anpassung von DINGO zunächst verzichtet.

jetzt gezeigt werden, wie diese Bedingungen ausgewertet werden und was aus ihnen bezüglich des Hauptziels – der Erkennung von nicht ausführbaren Transitionen – geschlossen werden kann.

2.4.1 Die erweiterte effektive PROVIDED-Bedingung

Zur Klärung der Frage, ob eine Transition überhaupt ausführbar ist, wird im Laufe der Analyse zu jeder Transition eine *effektive PROVIDED-Bedingung* gebildet. Diese besteht aus der Konjunktion der eigentlichen PROVIDED-Bedingung mit den negierten Schaltbedingungen aller Transitionen, die diese Transition überlappen. So erhält man eine Bedingung dafür, wann die Transition überhaupt noch ausführbar ist: Die Transition darf aus der Spezifikation entfernt werden, wenn diese Bedingung *sicher* unerfüllbar ist.

```
STATES s1, s2, s3, s4, s5;
VAR    x: INTEGER;

TRANS              TRANS              TRANS
   PRIORITY 1         PRIORITY 1         PRIORITY 2
   FROM s1, s2        FROM s2, s3        FROM s1, s2, s3, s4, s5
   PROVIDED x = 1     PROVIDED x = 2     PROVIDED (x = 1) OR (x = 2)
   NAME t1:           NAME t2:           NAME t3:
      BEGIN              BEGIN              BEGIN
      END;               END;               END;
```

Abbildung 2.17: Gemischte Zustands-/Bedingungs-Überlappung

Diese Vorgehensweise ist jedoch nur dann zulässig, wenn die überlappenden Transitionen jeweils alle (erreichbaren) Hauptzustände abdecken, die in der FROM-Klausel der überlappten Transition angegeben sind. So hängt im Beispiel aus Abbildung 2.17 die effektive PROVIDED-Bedingung von `t3` vom Hauptzustand ab:

Hauptzustand	überlappt durch	effektive PROVIDED-Bedingung
s1	t1	$x = 2$
s2	t1, t2	`FALSE`
s3	t2	$x = 1$
s4, s5	–	$(x = 1) \vee (x = 2)$

Wie man sieht, ist es nicht möglich, ohne Bezug auf die Hauptzustände eine sinnvolle Aussage über die effektive PROVIDED-Bedingung zu machen: Ignoriert man bei der Überlappung die Hauptzustände, so ist die resultierende Bedingung (hier `FALSE`) nicht zutreffend. Betrachtet man statt dessen lediglich solche Überlappungen, die alle Hauptzustände abdecken (tritt im Beispiel *nie* ein), so kann man im allgemeinen nur sehr schwache Aussagen ableiten.

Daher wird die effektive PROVIDED-Bedingung um eben diese Angaben über zugehörige Hauptzustände erweitert. Dabei ist die entstehende **erweiterte effektive PROVIDED-Bedingung** B eine Menge von Paaren, die je eine (nicht leere) Menge von erreichbaren Hauptzuständen S_i und die zugehörige effektive PROVIDED-Bedingung B_i enthalten. Die S_i sind dabei paarweise disjunkt.

$$B = \{(S_i, B_i) | i = 1 \ldots n\} \qquad \forall (i \neq j) : S_i \cap S_j = \emptyset \qquad \forall i : S_i \neq \emptyset$$

Die Bedingung zum obigen Beispiel lautet dann folgendermaßen:

$$B = \{(\{s1\}, x = 2), (\{s3\}, x = 1), (\{s4, s5\}, (x = 1) \vee (x = 2))\}$$

Wird die erweiterte Bedingung erneut überlappt, so müssen diejenigen Paare, die betroffene Hauptzustände referenzieren, bearbeitet und eventuell gespalten bzw. verschmolzen werden. Wird zu den Transitionen `t1`, `t2` und `t3` noch die Transition `t4` aus Abbildung 2.18 hinzugefügt, so ergibt sich die folgende erweiterte Bedingung für `t3`:

$$B' = \{(\{s1, s4\}, x = 2), (\{s3\}, x = 1), (\{s5\}, (x = 1) \vee (x = 2))\}$$

Die erweiterte effektive PROVIDED-Bedingung erlaubt so die korrekte Darstellung der effektiven Schaltbedingung. Ist diese Menge leer (alle effektiven PROVIDED-Bedingungen zu erreichbaren Hauptzuständen sind widerlegt), so ist die Transition unerreichbar.

```
TRANS
   PRIORITY 1
   FROM s4
   PROVIDED x = 1
   NAME t4:
      BEGIN
      END;
```

Abbildung 2.18: Erweiterung zur gemischten Zustands-/Bedingungs-Überlappung

2.4.2 Aussagenlogische Interpretation von Ausdrücken: Disjunktive Normalformen

Bei der Auswertung von Interferenzbedingungen ergeben sich immer wieder boolsche Ausdrücke, deren Erfüllbarkeit getestet werden muß. Da diese Ausdrücke prädikatenlogisch interpretiert werden müssen, ist ihre Erfüllbarkeit jedoch grundsätzlich unentscheidbar. Um zumindest offensichtlich widersprüchliche Aussagen bei der Analyse ausschließen zu können, genügt es, nach Beweisen für die *sichere Unerfüllbarkeit* eines Ausdrucks zu suchen. Kann ein solcher Beweis nicht gefunden werden, so gilt er im Zweifelsfall als *potentiell erfüllbar*.

Die Widersprüchlichkeit des Ausdrucks soll dabei *immanent*, also ohne äußere Zusatzbedingungen gezeigt werden.

Es empfiehlt sich dabei, die Ausdrücke zunächst **aussagenlogisch** zu betrachten. Dabei werden lediglich `NOT`-, `AND`- und `OR`-Ausdrücke rekursiv bearbeitet. Alle übrigen (Teil-) Ausdrücke sind *aussagenlogische Atome*, die zunächst nicht weiter analysiert werden sollen. Erst später werden dann unter allen erfüllenden aussagenlogischen Belegungen diejenigen eliminiert, die **prädikatenlogisch** unter der durch die Estelle-Semantik definierten Algebra widersprüchlich sind.

Als Beispiel betrachte man den Ausdruck

$$((x = 1) \vee (x = 2)) \wedge (x = 3)$$

Die aussagenlogische Betrachtung dieses Terms ergibt die drei Atome $x = 1$, $x = 2$ und $x = 3$. Die erfüllenden Belegungen sind in der folgenden Tabelle dargestellt:

$x = 1$	$x = 2$	$x = 3$
F	W	W
W	F	W
W	W	W

Im zweiten Schritt ergibt sich aus der Estelle-Semantik, daß die Konstanten 1, 2 und 3 jeweils paarweise verschieden und somit (zusammen mit der Gleichheitsaxiomatik) alle diese Belegungen widersprüchlich sind. Der Beweis der Unerfüllbarkeit ist damit erbracht.

Bei dem dargestellten Übergang von der aussagenlogischen zur prädikatenlogischen Interpretation wird der Widerspruchsbeweis für jede einzelne der (aussagenlogisch) erfüllenden Belegungen separat geführt. Dies bietet sich an, da so für jeden Einzelbeweis eine Menge von aussagenlogisch atomaren Termen vorliegen, die alle zugleich jeweils ein festes Ergebnis (W oder F) liefern müssen. Kommt es in einer Kombination zu Widersprüchen, so kann die entsprechende Belegung aus der Tabelle gestrichen werden. Konnten alle Zeilen gestrichen werden, so ist die Gesamtaussage unerfüllbar.

Es ist jedoch nicht gerade effizient, die aussagenlogische Interpretation der Ausdrücke anhand einer Tabelle im obigen Stil durch Aufzählen aller erfüllenden Belegungen zu führen. Wenn die Anzahl der aussagenlogisch atomaren Terme zunimmt, so steigt neben der Tabellenbreite auch die Tabellenlänge[42], obwohl meist viele der Zeilen sich nur an wenigen Stellen unterscheiden. Mit der Breite der Tabelle steigt zudem noch der Aufwand zur Widerlegung einzelner Zeilen an.

Wesentlich günstiger ist eine Darstellung der Aussage als **disjunktive Normalform**[43] (**DNF**). Diese besteht aus einer Disjunktion von Konjunktionen über (zum Teil negierten) atomaren Aussagen, wobei in den einzelnen

[42] Die Tabellenlänge steigt abhängig von der Tabellenbreite mit $O(e^n)$.

[43] siehe auch [Put88]

Konjunktionen im allgemeinen nicht jeweils alle Aussagen benutzt werden. Diese ermöglichen es (bei guter Ausdrucksminimierung[44]), die Komplexität der Ausdrücke relativ klein zu halten.

Nachdem eine DNF für einen Ausdruck entwickelt wurde, muß jede Konjunktion auf Widersprüchlichkeit getestet werden. Dabei müssen alle Atome unter einer gleichen prädikatenlogischen Belegung ihren jeweiligen Wahrheitswert liefern. Ist dies nicht möglich, so ist wird die Konjunktion aus der DNF entfernt. Die DNF für das obige Beispiel lautet

$$\begin{array}{ll} ((x = 1) \wedge (x = 3)) & \vee \\ ((x = 2) \wedge (x = 3)) & \end{array}$$

Der Widerspruch in beiden Konjunktionen liefert wie oben den Beweis für die Unerfüllbarkeit. Jedoch ist bereits in dem kleinen Beispiel zu erkennen, daß die Länge und Anzahl von Konjunktionen abnimmt. Neben dem geringeren Aufwand erleichtert dies auch das Lesen des Analyseprotokolls ganz erheblich.

2.4.3 Ausdruckstransformation und Gleichheit von Ausdrücken

Im vorigen Kapitel wurde dargestellt, wie aus den aussagenlogisch interpretierten Formeln disjunktive Normalformen über aussagenlogischen Atomen gebildet werden. In vielen Fällen enthalten diese Atome jedoch noch Anteile, die noch weiter zerlegt werden können, indem sie durch äquivalente Ausdrücke ersetzt werden.

So können atomare Ausdrücke mit konstantem Ergebnis (z.B. „$x \neq x$“) oder leicht transformierbare Ausdrücke (z.B. der unschöne Vergleich von Ausdrücken mit `TRUE` oder `FALSE` wie bei „$(x > 0) = TRUE$“) **umgewandelt** werden, noch bevor die Bildung der DNF durchgeführt wird. Dadurch kann oft schon in einer frühen Phase bereits ein Widerspruch erkannt und damit die Komplexität der DNFs reduziert werden. So ist zum Beispiel der Ausdruck $(x = y) \wedge (x \neq y)$ *aussagenlogisch* erfüllbar, da hier *zwei verschiedene* atomare Aussagen auftreten. Wird jedoch die Ungleichung durch eine gleichwertige negierte Gleichung ersetzt, so kann in dem resultierenden Ausdruck $((x = y) \wedge \neg(x = y))$ auf aussagenlogischer Ebene bereits ein Widerspruch erkannt werden.

Aber auch andere Ausdrücke können transformiert werden, z.B. um konstantwertige Terme bereits vorab auszuwerten (z.B. „$1 + 2 * 3$“) oder den *Test auf Gleichheit* zu erleichtern. Es ist nämlich für die oben gezeigte Umsetzung in DNFs gerade wichtig, die Gleichheit bzw. **Äquivalenz** zweier Terme feststellen zu können, da dadurch erst aussagenlogische Widersprüche erkannt werden können. So ist z.B. der Ausdruck „$(x = y) \wedge \neg(y = x)$“ aussagenlogisch erfüllbar, da es wiederum *zwei verschiedene* atomare Aussagen darin gibt. Erst unter

[44] Die Darstellung in der Tabelle ist genau genommen bereits eine DNF, nämlich die *ausgezeichnete* disjunktive Normalform (*ADNF*), in der jede Konjunktion alle atomaren Ausdrücke enthält. Der Vorteil der (normalen) DNF ergibt sich erst durch die Möglichkeit der Minimierung.

Ausnutzung der Kommutativität der Äquivalenzrelation „=“ kann der Widerspruch erkannt werden.

Es ist daher naheliegend, die Formeln vor ihrer Umwandlung in DNFs mit bestimmten Regeln zu bearbeiten, um so jeweils eine **äquivalente Formel** zu erhalten, die ein Maximum an Erkenntnissen bereits anhand der Bildung einer DNF zuläßt. Diese Regeln sind, wie in den einführenden Beispielen bereits absehbar, **Substitutionen von Teiltermen**[45].

Das gewählte Vorgehen ermöglicht auch die mehrfache Anwendung von Regeln an der selben Stelle der Formel, wodurch die Regeln sehr einfach gehalten werden können. Dazu wird rekursiv die Baumstruktur des Terms durchlaufen, und von den Blättern zur Wurzel hin werden Substitutionen durchgeführt. Der einzelne Substitutionsschritt wird dabei auf die Wurzel eines (Teil-) Terms angewandt, dessen echte Teilterme bereits vollständig bearbeitet wurden. Es werden nun so lange passende Substitutionsregeln benutzt, wie deren Anwendung möglich ist. Dabei ist bei einigen wenigen Regeln eine nochmalige Substitution innerhalb der Teilterme notwendig (s.u.).

Es ist offensichtlich essentiell, daß die Regeln zum einen die Semantik des Teilausdrucks jeweils nicht verändern (*Korrektheit*) und zum anderen keine unendlich fortgesetzte Regelanwendung auftreten kann (*Termination*). Die **Korrektheit** ist eine lokale Eigenschaft jeder Regel und muß anhand der Ausdruckssemantik von Estelle gezeigt werden. Die **Termination** ergibt sich jedoch aus der Gesamtheit aller Regeln und ist daher nicht leicht zu überblicken. Es empfiehlt sich, ein formales Kriterium zu ihrer Überprüfung festzulegen, um unendliche Substitutionsfolgen auszuschließen, wie sie mit den Assoziativitäts-Regeln

$$(x + (y + z)) \rightarrow ((x + y) + z) \qquad \text{und} \qquad ((x + y) + z) \rightarrow (x + (y + z))$$

oder der fortgesetzten Expansion von rekursiven Funktionsaufrufen möglich wären. Dieses Kriterium lautet, daß es eine **Reduktionsordnung**[46] gibt, die zu jeder Regel $l \rightarrow r$ die Beziehung $l > r$ erbringt, so daß bei jeder Regelanwendung der Gesamtterm verkleinert wird ([Ave95]). Da es in einer solchen Ordnung keine unendlich absteigenden Ketten gibt, sichert sie die Termination der Substitutionen. Am Ende des Abschnitts wird für die implementierten Regeln eine solche Ordnung (eine *Rekursive Pfadordnung* (*RPO*)) angegeben. Bei der Erweiterung der Substitutionsregeln sollte die Einhaltung dieser Ordnung sichergestellt oder ggfs. eine stärkere Ordnung festgelegt werden.

Termersetzungsregeln

Nun jedoch zu den **implementierten Termersetzungsregeln**. Ihre Auswahl erfolgte relativ willkürlich aufgrund von Konstellationen in konkreter Beispielspezifikationen. Sie können natürlich nur einen kleinen Ausschnitt der sinnvollen und korrekten Regeln darstellen. Die Term-Variablen (u, v, b_i usw.) stehen dabei (sofern nicht anders gekennzeichnet) für beliebige Teilterme. Man beachte

[45] im folgenden als „Substitutionen“ oder auch als „Termersetzungen“ bezeichnet.

[46] Eine Noethersche, stabile und monotone Partialordnung auf Termen. Siehe Anhang A

auch den Unterschied zwischen den Operatoren $-^1 u$ (Negation einer Zahl) und $u -^2 v$ (Differenz zweier Zahlen).

Allgemein:

R_1:	$\neg(\mathcal{T})$	$\rightarrow$	$\mathcal{F}$
R_2:	$\neg(\mathcal{F})$	$\rightarrow$	$\mathcal{T}$
R_3:	$\neg(\neg(u))$	$\rightarrow$	u
R_4:	$-^1(-^1(u))$	$\rightarrow$	u
R_5:	$u -^2 v$	$\rightarrow$	$u + (-^1 v)$
R_6:	$u \neq v$	$\rightarrow$	$\neg(u = v)$
R_7:	$u > u$	$\rightarrow$	$\mathcal{F}$
R_8:	$u = u$	$\rightarrow$	$\mathcal{T}$
R_9:	$u < v$	$\rightarrow$	$v > u$
R_{10}:	$u \geq v$	$\rightarrow$	$\neg(v > u)$
R_{11}:	$u \leq v$	$\rightarrow$	$\neg(u > v)$
R_{12}:	$\mathcal{T}?u{:}v$	$\rightarrow$	u
R_{13}:	$\mathcal{F}?u{:}v$	$\rightarrow$	v
R_{14}:	$b?u{:}u$	$\rightarrow$	u
R_{15}:	$\vee(u)$	$\rightarrow$	u
R_{16}:	$\vee(\cdots, \mathcal{T}, \cdots)$	$\rightarrow$	$\mathcal{T}$
R_{17}:	$\vee(\cdots, \mathcal{F}, \cdots)$	$\rightarrow$	$\vee(\cdots, \cdots)$
R_{18}:	$\wedge(u)$	$\rightarrow$	u
R_{20}:	$\wedge(\cdots, \mathcal{F}, \cdots)$	$\rightarrow$	$\mathcal{F}$
R_{21}:	$\wedge(\cdots, \mathcal{T}, \cdots)$	$\rightarrow$	$\wedge(\cdots, \cdots)$
R_{22}:	$+(u)$	$\rightarrow$	u
R_{23}:	$+(\cdots, 0, \cdots)$	$\rightarrow$	$+(\cdots, \cdots)$
R_{24}:	$*(u)$	$\rightarrow$	u
R_{25}:	$*(\cdots, 0, \cdots)$	$\rightarrow$	0
R_{26}:	$*(\cdots, 1, \cdots)$	$\rightarrow$	$*(\cdots, \cdots)$
R_{27}:	u `IN` $[v_1, \cdots, v_n]$	$\rightarrow$	$\vee(w_1, \cdots, w_n)$

$$w_i \equiv \begin{cases} ((u \geq l_i) \wedge (u \leq h_i)) & : v_i \equiv (l_i \ .. \ h_i) \\ (u = v_i) & : sonst \end{cases}$$

Wenn b_1, b_2 und b_3 Ausdrücke vom Typ `BOOLEAN` sind:

R_{28}:	$b_1?b_2{:}b_3$	$\rightarrow$	$(b_1 \wedge b_2) \vee (\neg b_1 \wedge b_3)$
R_{29}:	$b_1 = b_2$	$\rightarrow$	$(b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_2)$
R_{30}:	$b_1 > b_2$	$\rightarrow$	$b_1 \wedge \neg b_2$

Für Konstanten c_i und $\odot \in \{+, *, \vee, \wedge\}$:

R_{31}:	$\odot(\cdots, \odot(\cdots), \cdots)$	$\rightarrow$	$\odot(\cdots, \cdots, \cdots)$	
R_{32}:	$\odot(\cdots, c_1, \cdots, c_2, \cdots)$	$\rightarrow$	$\odot(\cdots, c_3, \cdots, \cdots)$	mit $c_3 = c_1 \odot c_2$
R_{33}:	$-^1(c_1)$	$\rightarrow$	c_2	mit $c_2 = -c_1$
R_{34}:	$c_1 = c_2$	$\rightarrow$	$\begin{cases} \mathcal{T} & : \ c_1 = c_2 \\ \mathcal{F} & : \ sonst \end{cases}$	
R_{35}:	$c_1 > c_2$	$\rightarrow$	$\begin{cases} \mathcal{T} & : \ c_1 > c_2 \\ \mathcal{F} & : \ sonst \end{cases}$	

Der **Bedingungs-Operator**[47] $(b?u{:}v)$ hat dabei die aus der Sprache C bekannte Semantik: Wird der boolsche Ausdruck b als *wahr* ausgewertet, so ist das Ergebnis des Ausdrucks das von u, ansonsten das von v. Für ihn gibt es zwar kein Ausdrucksäquivalent in Estelle, jedoch ist er bei der Analyse notwendig, um Funktionsaufrufe unter bestimmten Situationen noch geschlossen expandieren zu können.[48]

Daneben wurde auch die Syntax der Operatoren $+$, $*$, $\vee$ und $\wedge$ erweitert. Diese kommutativen und assoziativen Operatoren haben jetzt eine variable Stelligkeit. Dadurch kann mit Hilfe einer *Abflachung* verhindert werden, daß Ausdrücke wie

$$u_1 + u_2 + u_3 \qquad \text{und} \qquad u_3 + u_2 + u_1$$

aufgrund der impliziten Klammerung (links-assoziativ) nur schwer als gleich (-wertig) erkennbar sind. Beim Test auf Gleichheit erfordert die zweistellige Darstellung dieser Operatoren eine aufwendige Suche in den Ausdrucksbäumen (siehe Abbildung 2.19). Wird der Operator jedoch mit variabler Stelligkeit formuliert, so kann die Assoziativität auf die Kommutativität zurückgeführt werden. Damit lauten die beiden Ausdrücke dann

$$+(u_1, u_2, u_3) \qquad \text{und} \qquad +(u_3, u_2, u_1)$$

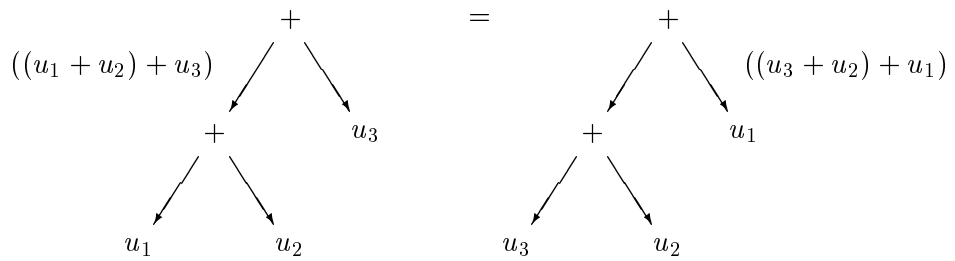


Abbildung 2.19: Probleme bei der Erkennung einer Ausdrucksäquivalenz

Gleichheit bzw. Gleichwertigkeit von Termen

Allgemein sind zwei Ausdrücke mit gleichem Top-Operator **gleich (-wertig)**, wenn auch alle Argumente in der gegebenen Reihenfolge gleichwertig sind (rekursive Prüfung). Handelt es sich bei dem Top-Operator um $+$, $*$, $\vee$ oder $\wedge$, so muß die Argumentreihenfolge der beiden Ausdrücke nicht übereinstimmen: Es genügt, wenn die Argumente *„modulo Permutation" der Argumentreihenfolge* paarweise gleichwertig sind.

Eine Besonderheit ergibt sich beim Test auf Gleichheit bei *Funktionsaufrufen.* Zunächst scheint es naheliegend, Funktionsaufrufe wie alle anderen Operatoren zu behandeln: Zwei Ausdrücke mit Funktionsaufrufen als Top-Operatoren

[47] aus den Regeln R_{12} bis R_{14} und R_{28}
[48] siehe Abschnitt 2.4.4

können als gleich angenommen werden (hinreichende Bedingung), wenn die aufgerufenen Funktionen identisch und alle Argumentausdrücke jeweils gleich sind (Rekursion). Dies scheint im Hinblick auf die Seiteneffektfreiheit der Funktionen angemessen, da die Ausführung eines Funktionsaufrufs keine Zustandsänderung bewirkt und daher beide Funktionsaufrufe garantiert im gleichen Kontext stattfinden.

Es ist jedoch in Estelle möglich, **indeterministische Funktionen** zu formulieren. Dies bewirkt, daß in einem gegebenen Zustand der mehrmalige Aufruf der *selben Funktion* mit den *selben Parametern* durchaus unterschiedliche Ergebnisse liefern kann. Folglich ist z.B. die Aussage $f(x) \neq f(x)$ durchaus erfüllbar, sofern $f(\ldots)$ indeterministische Ergebnisse liefert.

Indeterministische Funktionen können auf verschiedene Arten als benutzerdefinierte Funktionen[49] realisiert werden. So kann die Verwendung von Variablen, auf die noch kein Wert zugewiesen wurde, zu indeterministischen Ergebnissen führen. Ein Spezialfall dazu liegt vor, wenn nicht in jedem Fall ein Wert auf die Rückgabewert-Variable einer Funktion zugewiesen wird. Solche Konstellationen sind jedoch als Spezifikationsfehler einzustufen und werden daher im folgenden nicht näher untersucht.

```
FUNCTION random : BOOLEAN;
    BEGIN
        FORONE ret : BOOLEAN SUCHTHAT TRUE DO
            random := ret;
    END;
```

Abbildung 2.20: Definition einer indeterministischen Funktion

Nutzt man jedoch die Möglichkeiten zur expliziten Formulierung von Indeterminismus in Estelle (`all`, `forone`), so kann man bzgl. des Estelle-Standards zulässige[50] Funktionen mit indeterministischem Ergebnis formulieren. Ein Beispiel dazu ist in Abbildung 2.20 gegeben.

Dies hat negative Auswirkungen auf die Analysemöglichkeiten von (nicht expandierbaren) Funktionsaufrufen. So kann z.B. über die Gültigkeit einer PROVIDED-Klausel „$f(x) = f(x)$“ keine Aussage gemacht werden, solange nicht eindeutig nachgewiesen werden kann, daß $f(\ldots)$ ein deterministisches Antwortverhalten hat. Ebenso treten Probleme im Zusammenhang mit der Gleichheit von aussagenlogisch atomaren Aussagen auf: Betrachtet man aufbauend auf der Funktionsdefinition aus Abbildung 2.20 den Ausdruck „`random() OR NOT random()`“ ohne Rücksicht auf den Indeterminismus rein aussagenlogisch,

[49] Alle Operatoren und vordefinierten Funktionen sind dagegen immer deterministisch (es gibt u.a. keine vordefinierte `RAND()`-Funktion).

[50] Genauer: Der Estelle-Standard ([ISO89]) macht überhaupt keine Aussagen zu dieser Thematik (s.u.).

so wird er fälschlich als gültig eingestuft.[51] Aufgrund des indeterministischen Verhaltens von *random* stellen die beiden Funktionsaufrufe jedoch zwei *verschiedene* Atome dar. Eine korrekte aussagenlogische Repräsentation wäre also „$random_1 \vee \neg random_2$“.

Es wäre jedoch ebenfalls nicht zutreffend, alle derartigen *Funktionsreferenzen* als *grundsätzlich verschieden* einzustufen, da durch das o.g. Reduktionssystem mehrere Referenzen auf einen Funktionsaufruf produziert werden können. So ist zum Beispiel der Ausdruck „`random IN [FALSE, TRUE]`“ in Estelle immer erfüllt. Das Reduktionssystem erschließt daraus die Aussage „$(random = TRUE) \vee (random = FALSE)$“ und schließlich „$random \vee \neg random$“. Eine Unterscheidung der beiden Referenzen auf *random* ist hier nicht zutreffend, da effektiv der *selbe Funktionsaufruf* und damit in jedem Fall *das selbe Ergebnis* gemeint ist.

Die Lösung dieses Problems besteht darin, den Aufrufen von indeterministischen Funktionen selbst eine eindeutige Identifikation zuzuordnen, wie sie in den obigen Beispielen demonstriert wurde. Danach gelten zwei Funktionsreferenzen nur dann als gleich, wenn auch diese Identifikationen übereinstimmen.[52] Dadurch wird eine adäquate Behandlung dieses Problems ermöglicht: Bei indeterministischen Funktionen werden die *Funktionsaufrufe* und nicht die Funktionen selbst als Objekte mit unbekanntem aber *deterministischem Ergebnis* behandelt.

Dadurch werden die Auswirkungen indeterministischer Funktionen genau beschrieben. Insbesondere werden auch *Anomalien*, wie sie zusammen mit „`PROVIDED OTHERWISE`“-Klauseln auftreten können, korrekt erkannt. So können im Beispiel aus Abbildung 2.21 die Transitionen `t1` und `t2` trotz `OTHERWISE` beide *zugleich schaltbar* werden, da die Semantik von Estelle die PROVIDED-Klausel von `t2` wie ein „`PROVIDED NOT random`“ behandelt. Effektiv wird die indeterministische Funktion `random` also zweimal aufgerufen und kann daher auch zwei verschiedene Ergebnisse liefern. Dies führt dazu, daß unter Umständen in einer gegebenen Situation auch *beide* bzw. *keine* der beiden Transitionen schaltbar ist!

Die beschriebene Situation wird vom Analysewerkzeug korrekt behandelt und es wird die (erfüllbare) Interferenzbedingung $random_1 \wedge \neg random_2$ abgeleitet. Da jedoch nicht entscheidbar ist, ob eine Funktion deterministisch ist, müßte bei der Analyse im Zweifelsfall möglicher Indeterminismus immer angenommen werden. Dies schränkt natürlich die Möglichkeiten zur Erkennung gültiger bzw. unerfüllbarer Aussagen massiv ein.

Es bleibt die Frage nach der Verhältnismäßigkeit des Einsatzes solcher indeterministischer Funktionen in Schaltbedingungen, zumal neben den beschriebenen ungewöhnlichen Effekten bei „`PROVIDED OTHERWISE`“-Klauseln auch eine *Leerlauf-Schleifen-Optimierung* in Implementierungen von Spezifikationen

[51] Liefert der linke Aufruf von *random* $FALSE$ und der rechte $TRUE$, so wird der Ausdruck zu $FALSE$ evaluiert.

[52] In der Implementation wird dies über den Mechanismus zur Behandlung von Objektreferenzen behandelt (siehe Kapitel 3.2.3).

```
TRANS
    PROVIDED random
    NAME t1:
        BEGIN
        END;

    PROVIDED OTHERWISE
    NAME t2:
        BEGIN
        END;
```

Abbildung 2.21: OTHERWISE-Anomalie bei indeterministischen Bedingungen

praktisch unmöglich wird. Diese Optimierung besteht darin, alle Subsysteme, die sich in einem Zustand befinden, in dem sie keine Aktivität[53] mehr zeigen, aus weiteren Zyklen zur Bestimmung schaltbarer Transitionen auszuschließen, bis durch Interaktionen von außen eine Zustandsänderung eintritt. Ist die Schaltbarkeit von Transitionen jedoch möglicherweise indeterministisch, so kann diese Optimierung nicht stattfinden, da auch ohne Zustandsänderung des Subsystems Transitionen irgendwann schaltbar werden können.[54]

Der Estelle-Standard ([ISO89]) macht keine Aussagen über diese Thematik. Angesichts der restriktiven Handhabung von Seiteneffekten in Funktionen, der o.g. Nachteile und des eher fragwürdigen Nutzens hätten konsequenterweise auch indeterministische Funktionen ganz (oder zumindest ihr Einsatz in Transitionsklauseln) untersagt werden müssen. Es handelt sich hierbei wahrscheinlich um einen Fehler im Standard.

Vor diesem Hintergrund werden vom Analysewerkzeug benutzerdefinierte Funktionen als deterministisch betrachtet. Diese Voreinstellung kann jedoch bei Bedarf durch eine Kommandozeilen-Option übergangen werden (siehe Kapitel 3.4.2).

Beispiel zur Termersetzung

Zuletzt noch ein Beispiel für die Durchführung einer Termersetzungsfolge. Es werden lediglich die benutzten Regeln angegeben, die zugehörigen Stellen im Term ergeben sich aus dem Kontext. Es ist zu beachten, daß die Substitutionen beim Rekursions-Rücklauf erfolgen, also von den Blättern zur Wurzel des

[53] es laufen keine DELAY-Timer und im letzten Zyklus wurde keine schaltbare Transition gefunden

[54] Indeterministische Funktionen sind daher neben `PRIMITIVE`-Funktionen (siehe [SiSt91a]) ein weiterer Grund, in DINGO diese Form von Optimierung zu deaktivieren.

Ausdrucksbaumes hin. v und w stellen dabei beliebige Teilterme dar.

(1)	$(v \geq w) = \mathcal{F}$	
(2)	$(\neg(w > v)) = \mathcal{F}$	(R_{10})
(3)	$((\neg(w > v)) \wedge \mathcal{F}) \vee ((\neg\neg(w > v)) \wedge \neg\mathcal{F})$	(R_{29})
(4)	$(\mathcal{F} \vee ((\neg\neg(w > v)) \wedge \neg\mathcal{F})$	(R_{20})
(5)	$(\mathcal{F} \vee ((w > v) \wedge \neg\mathcal{F})$	(R_3)
(6)	$(\mathcal{F} \vee ((w > v) \wedge \mathcal{T})$	(R_2)
(7)	$(\mathcal{F} \vee (w > v))$	(R_{21})
(8)	$(w > v)$	(R_{17})

An dieser Stelle kann man auch erkennen, daß in bestimmten Situationen eine nochmalige Termersetzung innerhalb der Teilterme notwendig ist: Im obigen Beispiel wird in Schritt (2) durch Anwendung der Regel R_{29} aus einem Term der Struktur „$(\neg u) = \cdots$“ ein Term „$(\cdots) \vee ((\neg\neg u) \wedge (\cdots))$“. Wie man sieht ist *unterhalb* des Top-Operators $\vee$ der substituierbare Ausdruck „$\neg\neg u$“ entstanden. Wenn die Tests auf die Möglichkeit von Substitutionen zu diesem Zeitpunkt jedoch auf die (aktuelle) Top-Stelle beschränkt blieben[55], würde der Teilausdruck nicht mehr bearbeitet werden. Es ist jedoch auch nicht notwendig, den Teilausdruck „u“ selbst nochmals zu bearbeiten, da an diesem nichts verändert wurde.

Folglich müssen in solchen Situationen, in denen die *Tiefe des Ausdrucks vergrößert* wurde, an allen neu eingefügten Operatoren zusätzliche *lokale Substitutionsversuche* (d.h. ohne Rekursion in die Teilterme) vorgenommen werden. Zum Beispiel bei Regel R_{29} ist dies für die beiden $\neg$-Operatoren und die beiden $\wedge$-Operatoren notwendig. Dadurch können – wie im obigen Beispiel – sämtliche möglichen Termersetzungsschritte durchgeführt werden.

Korrektheit der Regeln

Die Korrektheit der einzelnen Regeln kann jeweils einzeln verifiziert werden. Dabei sind die Freiheit von Seiteneffekten und die Zuordnung eindeutiger Identifikatoren zu den Aufrufen indeterministischer Funktionen (s.o.) wichtige Vorbedingungen, da erst dadurch die Kommutativität der entsprechenden Operatoren oder die Möglichkeit, Teilausdrücke mehrfach auszuwerten (wie bei R_{27} bis R_{29}) gesichert wird.

An dieser Stelle wird lediglich beispielhaft ein Beweis für Regel R_{29} angegeben. Da diese Regel mit Variablen vom Typ `BOOLEAN` arbeitet, kann die Korrektheit mit Hilfe einer Aufzählung aller möglichen Variablen-Belegungen gezeigt werden:

[55] Die Substitutionsschritte werden beim *Rekursionsrücklauf* durchgeführt. Alle Substitutionen der Teilterme sind bereits abgeschlossen.

b_1	b_2	$b_1 = b_2$	$(b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_2)$
F	F	T	T
F	T	F	F
T	F	F	F
T	T	T	T

Der Beweis der Korrektheit ergibt sich aus der Gleichheit der Werte in den beiden rechten Spalten.

Termination der Regelanwendung

Wie bereits erläutert muß zum Nachweis der Termination der Substitutionen sichergestellt werden, daß keine unendlich fortgesetzte Regelanwendung möglich ist. Die Idee zum Nachweis der Termination ist, daß alle Regeln die von ihnen bearbeiteten Terme in einer bestimmten Hinsicht *verkleinern* müssen, diese Verkleinerung jedoch nicht unendlich oft möglich ist.

So werden die Terme bei vielen Regeln „kürzer“ (d.h. die Anzahl der Funktionen, Variablen und Konstanten nimmt ab), bei einigen anderen jedoch können die Terme zugunsten einer besseren aussagenlogischen Zerlegbarkeit deutlich komplexer werden. So tauchen bei der Expansion einer Gleichung zwischen boolschen Ausdrücken (R_{29}) die beiden boolschen Teilausdrücke im Ergebnis doppelt so oft auf, so daß sich die Länge des Ausdrucks bei ihrer Anwendung mehr als verdoppelt. Noch extremer kann die Situation bei der *Expansion von Funktionsaufrufen* werden, da hier (fast) beliebig neue Terme eingebracht werden können.

Es ist also offensichtlich nicht ganz einfach, eine passende Ordnung anzugeben, zumal dann auch noch der Beweis erbracht werden muß, daß für *jede beliebige* Regelanwendung der Term bezüglich dieser Ordnung verkleinert wird.

Bei der unten angegebenen Partialordnung auf Termen handelt es sich jedoch um eine *Reduktionsordnung* (siehe Anhang A). Dies stellt sicher,

- daß nur *Regeln* daraufhin überprüft werden müssen, ob die linke Seite größer als die rechte ist. Die Verträglichkeit mit Substitutionen und der Termstruktur stellt sicher, daß dies dann auch für alle *Regelanwendungen* gilt;
- daß es keine unendlich fortgesetzte Regelanwendungen gibt, wenn alle Regelanwendungen die Terme bezüglich der Reduktionsordnung verkleinern (Noethersch).

Eine passende Reduktionsordnung ist die **Rekursive Pfadordnung** $>_{rpo}$ zur Präzedenz $\succ$. Dabei ist $\succ$ eine Quasi-Ordnung auf den Funktionen, Konstanten und Operatoren, mit

$$\begin{array}{lll} f \succ g & \text{gdw} & \exists i,j: \quad (i > j) \wedge (f \in M_i) \wedge (g \in M_j) \\ f \approx g & \text{gdw} & \exists i: \quad (f \in M_i) \wedge (g \in M_i) \end{array}$$

$$
\begin{aligned}
M_1 &= \{c \mid c \text{ ist eine Konstante (incl. } \mathcal{T} \text{ und } \mathcal{F})\} \\
M_2 &= \{\vee, \wedge, \neg\} \\
M_3 &= \{>\} \\
M_4 &= \{+, -^1, =, <, \geq, \leq, \text{ „..“ }, [\,]\} \\
M_5 &= \{?, -^2, \neq\}
\end{aligned}
$$

Zusätzlich erhalten alle *benutzerdefinierten Funktionen* eine Stellung innerhalb von $\succ$, die sie größer als alle (direkt oder indirekt) darin benutzten Funktionen und Operatoren macht.[56] Zusammen mit Bedingung α (für Parameter-Referenzierungen) bzw. β (für alle anderen Ausdrücke) der RPO-Definition (siehe Anhang A) ergibt sich, daß die Expansion Funktionsaufrufe bezüglich $>_{rpo}$ verkleinert.

Es kann gezeigt werden, daß für alle angegebenen Regeln die linke Seite echt größer bezüglich der o.a. RPO als die rechte Seite ist. Dies soll beispielhaft anhand der Regel R_{29} vorgeführt werden:

(1)		$=$ $\succ$	$\neg$	
(2)		$=$ $\succ$	$\wedge$	
(3)		$=$ $\succ$	$\vee$	
(4)	$b_1 = b_2$	$>_{rpo}$	b_1	(α)
(5)	$b_1 = b_2$	$>_{rpo}$	b_2	(α)
(6)	$b_1 = b_2$	$>_{rpo}$	$\neg b_1$	$(\beta : (1), (4))$
(7)	$b_1 = b_2$	$>_{rpo}$	$\neg b_2$	$(\beta : (1), (5))$
(8)	$b_1 = b_2$	$>_{rpo}$	$b_1 \wedge b_2$	$(\beta : (2), (4), (5))$
(9)	$b_1 = b_2$	$>_{rpo}$	$\neg b_1 \wedge \neg b_2$	$(\beta : (2), (6), (7))$
(10)	$b_1 = b_2$	$>_{rpo}$	$(b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_2)$	$(\beta : (2), (8), (9))$

Damit führt jede Regelanwendung zu einer Verkleinerung des Terms bzgl. der RPO. Da diese Ordnung Noethersch ist, kann es keine unendliche Folge von Regelanwendungen geben.

2.4.4 Expansion von Funktionsaufrufen

Die Expansion von Funktionsaufrufen innerhalb von Ausdrücken ist ein wichtiger Schritt bei der Analyse. Hierbei soll statt des Aufrufs ein geschlossener äquivalenter Ausdruck eingesetzt werden, so daß mehr Informationen über das Ergebnis der Funktion gewonnen werden können.

Ohne Kenntnis der Semantik einer aufgerufenen Funktion kann bestenfalls die Ergebnisgleichheit zweier Aufrufe dieser Funktion mit den selben Parametern angenommen werden; wenn die Funktion indeterministisch sein könnte, so ist nicht einmal das möglich.

Kann jedoch ein geschlossener Ausdruck für das Ergebnis eines Funktionsaufrufs formuliert werden, so kann das „Innenleben“ der Funktion im Rahmen der normalen Termanalyse vollständig untersucht werden. Leider ist es nicht immer möglich, einen solchen Ausdruck anzugeben.

[56] Natürlich ist dies nur für weder direkt noch indirekt rekursive Funktionen möglich (s.u.)!

Die Implementation des Analysewerkzeugs beschränkt sich bei der Expansion von Funktionen zunächst auf solche Fälle, für die ein *äquivalenter Estelle-Ausdruck* angegeben werden kann. Der Ausdruck enthält dann i.a. noch Referenzen auf die formalen Parameter. Diese werden dann bei der Expansion eines konkreten Funktionsaufrufs durch die jeweils zugehörigen Ausdrücke der *aktuellen Parameter* ersetzt. Enthält der resultierende Ausdruck selbst noch expandierbare Funktionsaufrufe, so werden diese bei der folgenden Substitution rekursiv ebenfalls expandiert.

Es ist offensichtlich, daß bei (direkt oder indirekt) rekursiven Funktionen diese Expansion nicht terminieren kann. Daher wird entsprechend des Rekursionsverbots (aus dem Terminationsbeweis der Substitutionen, siehe Abschnitt 2.4.3) die Expansion rekursiver Funktionen unterdrückt.[57]

Ein Beispiel für eine einfach zu expandierende Funktion ist in Abbildung 2.22 zu finden. Hier besteht der Funktionsblock lediglich aus einer Zuweisung an die Rückgabevariable. Die Expansion verläuft nach dem obigen Muster und liefert schließlich die PROVIDED-Bedingung

$$
\begin{array}{ll}
(& (((nextseq + 5) \text{ MOD } 20) \geq nextseq) \wedge \\
 & ((msg.seqnr \geq nextseq) \wedge (msg.seqnr \leq ((nextseq + 5) \text{ MOD } 20))) \\
) \vee & \\
(& (((nextseq + 5) \text{ MOD } 20) < nextseq) \wedge \\
 & ((msg.seqnr \geq nextseq) \vee (msg.seqnr \leq ((nextseq + 5) \text{ MOD } 20))) \\
) &
\end{array}
$$

Diese kann dann weiter substituiert und später in eine DNF umgewandelt werden.

Der Einsatz einer Funktion im obigen Beispiel diente nur der besseren Lesbarkeit der Spezifikation, denn offensichtlich hätte man auch direkt die expandierte Form in der PROVIDED-Klausel angeben können. Es gibt jedoch eine wichtige Klasse von PROVIDED-Bedingungen, in denen der Einsatz einer Hilfsfunktion unvermeidbar ist: Man kann in Estelle im allgemeinen keinen geschlossenen Ausdruck angeben, in dem eine **partielle Funktion** zu einer **totalen Funktion** erweitert wird. So kann zum Beispiel die folgende totale Erweiterung der (partiellen) Logarithmusfunktion $\ln(x)$ nicht[58] geschlossen ausgedrückt werden.

$$
\widehat{ln}(x) := \begin{cases} ln(x) & \text{für } x > 0 \\ 0 & \text{für } x \leq 0 \end{cases} \quad \rightsquigarrow \quad \widehat{ln} \supset ln
$$

Dazu fehlt in Estelle ein **Auswahl-Operator**, der abhängig vom Wert eines boolschen Ausdrucks nur einen von zwei Termen auswertet. In der Sprache C könnte die obige totale Erweiterung der Logarithmusfunktion folgendermaßen geschlossen formuliert werden: $((x > 0)\ ?\ ln(x) : 0)$.

[57] Genauer: Es wird nur die erste Stufe der Rekursion expandiert, alle folgenden Rekursionen werden als nichtexpandierbar gekennzeichnet.

[58] oder zumindest nicht naheliegend

```
FUNCTION cyclic_range_check(v, lo, hi: INTEGER) : BOOLEAN;
   BEGIN
   cyclic_range_check := ((hi ≥ lo) AND ((v ≥ lo) AND (v ≤ hi))) OR
                         ((hi < lo) AND ((v ≥ lo) OR  (v ≤ hi)))
   END

VAR nextseq: INTEGER;

TRANS
   WHEN ipin.frame(msg)
   PROVIDED cyclic_range_check(msg.seqnr,
                               nextseq,
                               (nextseq+5) MOD 20 )
      BEGIN
      END;
```

Abbildung 2.22: Expandierbare Funktion mit einfacher Struktur

Zur Auswahl zwischen *boolschen Ausdrücken* könnte man in C ähnlich wie im Beispiel aus Abbildung 2.22 auch die zur Auswertung eines Teilausdrucks notwendigen Bedingungen konjunktiv verknüpft voranstellen. Ein äquivalenter[59] Ausdruck zu $(b?b_t{:}b_f)$ wäre dann $((b \wedge b_t) \vee (\neg b \wedge b_f))$. Die Teilausdrücke b_t und b_f werden hier genau wie im vorigen Auswahloperator-Ausdruck nur bei entsprechendem Ergebnis von b ausgewertet, da es in C Regeln über die *Auswertereihenfolge* in Konjunktionen und Disjunktionen gibt: In einer Konjunktion wird der rechte Term nur ausgewertet, wenn die Auswertung des linken Terms `TRUE`[60] ergibt.

Leider gibt es in Estelle (genau wie im zugrundeliegenden Pascal) keine solche Regelung, so daß auch bei boolschen Ausdrücken möglicherweise *jeder* Teilterm ausgewertet wird, bevor das Ergebnis bestimmt wird. Eine Erweiterung partieller *boolscher* Ausdrücke ist also ebenfalls nicht in geschlossener Form möglich[61].

Es gibt eine Klasse von partiellen Ausdrücken, die in vielen Estelle-Spezifikationen benutzt werden: Der Zugriff auf Elemente eines **varianten Records**. Dies sind Datenstrukturen, auf deren Komponenten zum Teil nur unter bestimmten Bedingungen zugegriffen werden darf. So darf z.B. in der Record-Definition aus Abbildung 2.23 auf das Feld `positive` nur zugegriffen werden, wenn das Feld `tp` den Wert `ack` hat. Die Einhaltung dieser Konvention wird zum Teil sogar vom Laufzeitsystem der Implementation einer Spezifikation dynamisch überprüft, und eine Verletzung dieser Bedingung führt zu einem Programmabbruch.

[59] Sofern die Auswertung von b keine Seiteneffekte hat. Sonst ist eine Hilfsvariable nötig.

[60] genauer: Einen Wert ungleich Null (C-Semantik boolscher Werte)

[61] Im Beispiel aus Abbildung 2.22 sind alle Teilausdrücke total, daher genügte die geschlossene Darstellung des oben geschilderten Auswahlverfahrens.

```
TYPE MsgType = (data, ack);
     sduType = ...;
     VarRec  = RECORD
                  CASE tp: MsgType OF
                     data: (
                        sdu: sduType;
                     );
                     ack: (
                        positive: BOOLEAN; {pos. oder neg. Quittung}
                     );
               END;
```

Abbildung 2.23: Definition eines varianten Records

Da der Zugriff auf bestimmte Felder eines varianten Records somit eine partielle Funktion darstellt, ist in Estelle der Einsatz einer Funktion an dieser Stelle unvermeidlich. Diese Funktionen haben eine einfache Struktur: Sie bestehen aus Zuweisungen an die Rückgabewert-Variable, die in einem Baum von `IF-THEN-ELSE`-Statements angeordnet sind. So wird die Abfrage, ob es sich um eine positive Quittung handelt, durch die Funktion in Abbildung 2.24 realisiert.

```
FUNCTION is_positive_ack(msg: VarRec) : BOOLEAN;
   BEGIN
   IF msg.tp = ack THEN
      is_positive_ack := msg.positive
   ELSE
      is_positive_ack := FALSE
   END
```

Abbildung 2.24: Totale Erweiterung eines partiellen Ausdrucks

Das Analysewerkzeug beschränkt sich zur Zeit auf die Expansion von Funktionen, die gemäß der folgenden Syntaxdefinition[62] eine *x-function-declaration* (*„expandable function declaration“*) sind:

```
x-function-declaration  ::= function-heading ";" x-function-block.
x-function-block        ::= x-compound-statement.
x-compound-statement ::= "BEGIN" x-statement [";"] "END".
x-statement             ::= function-identifier ":=" expression |
                            "IF" expression "THEN" x-statement "ELSE" x-statement |
                            x-compound-statement.
```

Man beachte, daß durch diese Syntax bei der Ausführung der Funktion immer nur *genau eine* Zuweisung an die Rückgabewert-Variable *ausgeführt*

[62] Die Definition orientert sich an der Darstellung im Estelle-Standard ([ISO89])

wird. Insbesondere enthalten deshalb *ExpBlock*s auch nur eine Anweisung. Die `IF-THEN-ELSE`-Statements dürfen dabei beliebig verschachtelt werden, so können auch komplexe Fallunterscheidungen formuliert werden.

Der aus der Expansion resultierende Ausdruck läßt sich im allgemeinen natürlich nicht mehr als geschlossener Estelle-Ausdruck darstellen, denn gerade diese mangelnde Umsetzbarkeit bedingt ja erst die Notwendigkeit eines Funktionsaufrufs und ist somit die Hauptmotivation für eine Funktionsexpansion. In der *internen* Darstellung der Ausdrücke wurde daher der o.g. *Auswahloperator* eingeführt (siehe auch Abschnitt 2.4.3). Dieser erlaubt eine Nachbildung der Strukturen, wie sie sich aus den `IF-THEN-ELSE`-Ausdrücken in den expandierbaren Funktionen ergeben. So lautet zum Beispiel der zur Funktion in Abbildung 2.24 äquivalente (geschlossene) Ausdruck

$$(msg.tp = ack)\ ?\ msg.positive : \texttt{FALSE}$$

Dieser würde bei den darauf folgenden Substitutionsschritten weiter vereinfacht werden und schließlich $(msg.tp = ack) \wedge msg.positive$ liefern. In Estelle ist dieser Ausdruck nur partiell definiert, in der Analyse werden die Terme jedoch nur *symbolisch* bearbeitet, so daß es zu keinem Zugriffsfehler kommen kann. Die resultierenden Bedingungen werden nur als *Anforderungen* an einen erfüllenden Zustand der Spezifikation betrachtet. Dabei wird lediglich untersucht, ob diese Anforderungen offensichtlich widersprüchlich sind. Wie dies möglich ist, wird Thema des nächsten Abschnitts sein.

2.4.5 Widerlegbarkeit von Konjunktionen atomarer Aussagen

Bisher wurde dargestellt, wie

- aus den Schaltbedingungen von Transitionen (bzw. Transitionspaaren) *boolsche Ausdrücke abgeleitet* werden,
- diese Ausdrücke durch Termersetzungen *vereinfacht* werden können und schließlich
- in disjunktive Normalformen umgewandelt und *aussagenlogisch analysiert* werden.

Diese Vorgehensweise erlaubt es bereits, eine große Zahl von Widersprüchen zu erkennen, soweit sie im weiteren Sinne[63] aussagenlogischer Natur sind. Es bleibt jedoch eine Vielzahl von Widersprüchen, die nicht erkannt werden. So wird der Ausdruck $(1 > x) \wedge (x > 2)$ nach wie vor als erfüllbar eingestuft, da er in der Termersetzung nicht umgewandelt werden kann und aussagenlogisch erfüllbar ist. *Aussagenlogisch* können die beiden atomaren Ausdrücke $1 > x$ und $x > 2$ beide zugleich die Belegung $\mathcal{T}$ haben. Der *prädikatenlogische* Widerspruch ergibt sich erst aus der Transitivität von $>$, mit deren Hilfe der Ausdruck

[63] Die vorangehende Termersetzung liegt natürlich außerhalb der Aussagenlogik.

$1 > 2$ ableitbar ist. Dieser ist offensichtlich in der durch die Estelle-Semantik definierten Algebra unerfüllbar.

Allgemein ist die Erfüllbarkeit solcher prädikatenlogischer Ausdrücke unentscheidbar, daher kann immer nur nach einer bestimmten Klasse von Widersprüchen gezielt gesucht werden. Dabei ist die Darstellung als disjunktive Normalform nützlich, da hier die einzelnen Konjunktionen separat untersucht werden können: Damit ein Ausdruck in disjunktiver Normalform unerfüllbar ist, muß **jede** der Konjunktionen unerfüllbar sein. Man kann sich an dieser Stelle bei der Untersuchung auf *Mengen von aussagenlogisch atomaren boolschen Ausdrücken* beschränken, die *alle zugleich* (evtl. negiert[64]) *erfüllbar* sein müssen, um widerspruchsfrei zu sein. Wird innerhalb einer Konjunktion ein Widerspruch gefunden, so kann diese komplett aus der DNF entfernt werden. Wurden alle Konjunktionen entfernt, so ist die ganze DNF unerfüllbar.

Durch die Substitutionen und die nachfolgende aussagenlogische Analyse wurden viele Operatoren eliminiert. An der Top-Stelle von DNF-Atomen können nur noch folgende Ausdruckstypen auftreten:

- `EXIST`-Ausdrücke
- die Operatoren `IN`[65], $=$, $>$
- Referenzen auf boolsche Variablen (auch Arrays und Strukturen) sowie boolsche ANY-Konstanten
- nicht expandierbare Funktionsaufrufe

In der Implementation des Analysewerkzeugs wurde zunächst einmal nur die wichtigste Klasse von Widerspruchsmerkmalen analysiert, nämlich die Verletzung der Eigenschaften der Relationen $=$ und $>$. Da die DNF-Atome auch negiert auftreten können, müssen dabei auch die Operatoren $\neq$[66] und $\geq$[67] behandelt werden.

Dabei können folgende Merkmale der Relationen ausgenutzt werden: Es gilt für alle Ausdrücke A, B und C:

[64] je nach negiertem oder nichtnegiertem Auftreten in der DNF

[65] Regel R_{27} ist nur anwendbar, wenn als rechtes Argument des `IN`-Operators *unmittelbar* ein Set steht (im Gegensatz zu einer *Verknüpfung* von Sets).

[66] $\neg(A = B) \rightsquigarrow (A \neq B)$

[67] $\neg(A > B) \rightsquigarrow (B \geq A)$

Reflexivität:	$(A = A)$	
	$(A \geq A)$	
Irreflexivität:	$\neg(A > A)$	
	$\neg(A \neq A)$	
Transitivität:	$(A = B) \wedge (B = C)$	$\leadsto (A = C)$
	$(A > B) \wedge (B > C)$	$\leadsto (A > C)$
	$(A \geq B) \wedge (B \geq C)$	$\leadsto (A \geq C)$
	$(A > B) \wedge (B \geq C)$	$\leadsto (A > C)$
	$(A \geq B) \wedge (B > C)$	$\leadsto (A > C)$
Symmetrie:	$(A = B)$	$\leadsto (B = A)$
	$(A \neq B)$	$\leadsto (B \neq A)$
Antisymmetrie:	$(A \geq B) \wedge (B \geq A)$	$\leadsto (A = B)$

Dabei bildet $\geq$ eine *Quasiordnung* zur *Äquivalenzrelation* $=$, da über explizite Gleichheitsaussagen Äquivalenzklassen gebildet werden. Im Analysewerkzeug wird dieser Umstand ausgenutzt, indem die Beziehungen $\neq$, $>$ und $\geq$ zwischen den **Äquivalenzklassen** und nicht zwischen den einzelnen Ausdrücken formuliert werden. Dazu wird jedem Teilausdruck aus einer Aussage

$$A \bowtie B \text{ mit } \bowtie \in \{=, \neq, >, \geq\}$$

eine eigene Äquivalenzklasse zugeordnet (sofern nicht sogar bereits eine Klasse mit diesem Element existiert) und die Aussage auf diese Äquivalenzklasse ($[A]$ bzw. $[B]$) bezogen ($\leadsto [A] \bowtie [B]$). Handelt es sich beim Operator $\bowtie$ um ein $=$, so werden die beiden Äquivalenzklassen *verschmolzen* und alle Beziehungen zwischen den beteiligten Klassen werden auf diese neuen Klassen abgeändert.

Im Anschluß daran werden gemäß des folgenden Algorithmus die o.g. Eigenschaften der Relationen $>$, $\geq$, und $\neq$ auf dieses System, bestehend aus Äquivalenzklassen und Relationen zwischen diesen, angewandt:

1. *Falls eine Verletzung der **Irreflexivitätsregeln** ermittelt werden kann: Abbruch mit „Widerspruch erkannt“.*

2. *Nacheinander alle anwendbaren **Transitivitätsregeln** anwenden und ggfs. neue Beziehungen einführen.*

3. *Falls in Schritt 2. neue $\geq$-Beziehungen entstanden sind: Alle anwendbaren **Antisymmetrieregeln** anwenden und ggfs. Äquivalenzklassen verschmelzen.*

4. *Falls sich bei Schritt 2. und 3. keine Änderungen ergeben haben: Abbruch mit „keinen Widerspruch erkannt“.*

5. *Weiter mit Schritt 1.*

Die Reflexivitäts- und Symmetrieregeln werden dabei implizit bei der Verwaltung der Äquivalenzklassen und Relationen berücksichtigt.

Zum Beispiel sei folgende Konjunktion aus einer zu testenden DNF gegeben:

$$(x = a + b) \wedge (y \geq x) \wedge (a + b \neq z) \wedge (z \geq y) \wedge (a + b \geq z) \wedge (y = c)$$

Daraus werden durch das oben geschilderte Verfahren folgenden Äquivalenzklassen (K_i) und Beziehungen gebildet:

$$\begin{array}{rl} K_1: & \{x, a+b\} \\ K_2: & \{y, c\} \\ K_3: & \{z\} \\ >: & \{\} \\ \geq: & \{(K_2, K_1),\ (K_3, K_2),\ (K_1, K_3)\} \\ \neq: & \{(K_1, K_3)\} \end{array}$$

Hier liegen noch keine unmittelbaren Widersprüche zur Irreflexivität von $\neq$ und $>$ vor. Nun können durch Anwendung der o.g. Relationseigenschaften weitere Beziehungen abgeleitet werden. So kann die Transitivität von $\geq$ ausgenützt werden und es ergibt sich[68]

$$\begin{array}{rll} \geq: & \{ & (K_2, K_1),\ (K_3, K_2),\ (K_1, K_3), \\ & & (K_2, K_3),\ (K_3, K_1),\ (K_1, K_2), \\ & & (K_1, K_1),\ (K_2, K_2),\ (K_3, K_3) \quad \} \end{array}$$

Daraus läßt sich über die Antisymmetrie die paarweise Gleichheit der drei Klassen ableiten. Diese werden daraufhin verschmolzen. Das resultierende System enthält dann in der $\neq$-Relation den erwarteten Widerspruch:

$$\begin{array}{rl} K_1: & \{x, a+b, y, c, z\} \\ >: & \{\} \\ \geq: & \{(K_1, K_1)\} \\ \neq: & \{(K_1, K_1)\} \end{array}$$

Zusätzlich zu diesen syntaktischen Prüfungen wird auch noch die Verträglichkeit der Äquivalenzklassen und der Beziehungen zwischen ihnen mit den Werten von *konstantwertigen Ausdrücken* getestet. So werden Widersprüche erkannt, wie sie zum Beispiel am Anfang dieses Abschnitts genannt wurden:
$(1 > x) \wedge (x > 2)$ liefert nach Anwendung der Transitivitätsregeln von $>$ unter anderem die Klassenbeziehung $\{1\} > \{2\}$. Analog würde der Ausdruck $(x = 1) \wedge (x = 2)$ zur Äquivalenzklasse $\{1, 2, x\}$ führen.

Um diese Widersprüche erkennen zu können wird Schritt 1. des o.g. Algorithmus durch folgende Anweisung ersetzt:

1.' Falls ***(a)*** eine Verletzung der Irreflexivitätsregeln ermittelt werden kann, ***(b)*** zwei konstantwertige Ausdrücke mit verschiedenen Werten in einer Äquivalenzklasse liegen oder ***(c)*** die $>$, $\geq$ oder $\neq$-Beziehungen zwischen den Werten von konstantwertigen Mitgliedern zweier Äquivalenzklassen (sofern vorhanden) verletzt werden:
Abbruch mit „Widerspruch erkannt".

[68] Die letzten drei Paare „$K_i \geq K_i$" sind implizit bereits in $\geq$ enthalten und werden deshalb im Programm nicht explizit gebildet.

Kann durch die oben geschilderten Verfahren kein Widerspruch in der Konjunktion gefunden werden, so muß sie weiter als *potentiell erfüllbar* gelten.

Es sind vielfältige **Erweiterungen** der Widerspruchserkennung denkbar. So könnte zum Beispiel die Erkennung der Unerfüllbarkeit von „$a+10 = a$“ relativ einfach implementiert werden, falls sich die Notwendigkeit dazu ergibt. Ebenso wäre es interessant, die bereits gebildeten Äquivalenzklassen dazu zu nutzen, die Erkennung der Äquivalenz von Teiltermen zu verbessern. Dadurch könnte zum Beispiel auch der Widerspruch in $(a + b = 1) \wedge (a = 1) \wedge (b = 2)$ gefunden werden, indem a und b jeweils durch die äquivalenten Konstanten ersetzt und schließlich die Aussage $3 = 1$ abgeleitet wird.

Da die Widerspruchserkennung eingekapselt ist, kann anhand eines konkreten Problems relativ leicht eine Erweiterung vorgenommen werden. Welche Widersprüche zusätzlich gesucht werden sollten, ist leicht anhand von manuell als unerfüllbar erkannten Aussagen aus dem Analyseprotokoll erkennbar. Die grundsätzlich bedingte Unvollständigkeit des Erkennungsverfahrens bleibt jedoch die eigentliche Grenze für die Möglichkeiten dieser Analyse.

2.5 Entfernung von „dead code“

Wurde eine Transition als unerreichbar erkannt, so soll sie optional zur Laufzeitoptimierung aus der Spezifikation entfernt werden können. Dabei muß jedoch sichergestellt werden, daß dadurch in keinem Fall eine Änderung der Semantik verursacht wird.

Zunächst garantiert die defensive Optimierungsstrategie, daß eine Transition nur dann als unerreichbar erkannt wird, wenn sie *unter keinen Umständen*[69] ausführbar werden kann. Da die zu entfernende Transition nicht feuern kann, kann sie auch keinen *aktiven* Einfluß auf das Verhalten der Spezifikation gehabt haben. Ihre Entfernung kann dennoch Auswirkungen auf die Spezifikation haben: Zum einen könnte eine Transition, auch ohne jemals zu schalten, Einfluß auf das Auswahlverfahren haben; zum anderen ist es durch bestimmte Elemente der Syntax von Estelle nicht immer ohne weiteres möglich, eine Transition ohne Änderung an den anderen Transitionen zu entfernen.

Der erste Punkt, die **semantische Wirkung** der Entfernung einer nicht schaltbaren Transition, erweist sich bei genauerer Betrachtung als unproblematisch. So kann eine solche Transition weder als Schaltwirkung (dazu müßte sie feuern, siehe Abschnitt 2.3.2), noch im Rahmen des Auswahlverfahrens (die Auswertung aller Klauseln erfolgt ohne Seiteneffekte) irgendwelche Zustandsänderungen verursachen. Aber auch auf das Auswahlverfahren selbst hat ihre Anwesenheit keine Auswirkungen. Zwar kann die Transition *bereit* werden, doch da sie selbst niemals ausgewählt wird, ändert sie nichts am Auswahlverhalten. Insbesondere gehen von einer nicht ausführbaren Transition weder relevante Überlappungen noch Indeterminismus aus. Jede Überlappung, die von der zu

[69] Korrektheit der Zusicherungen vorausgesetzt (s.o.)

entfernenden Transition im Rahmen des erreichbaren Zustandsraumes ausgeht, geht auch von den Transitionen aus, von denen sie selbst überlappt wurde. So wird zum Beispiel in Abbildung 2.25 auch nach dem Entfernen der (vollständig überlappten) Transition B die Transition C immer noch in gleichem Maße überlappt.

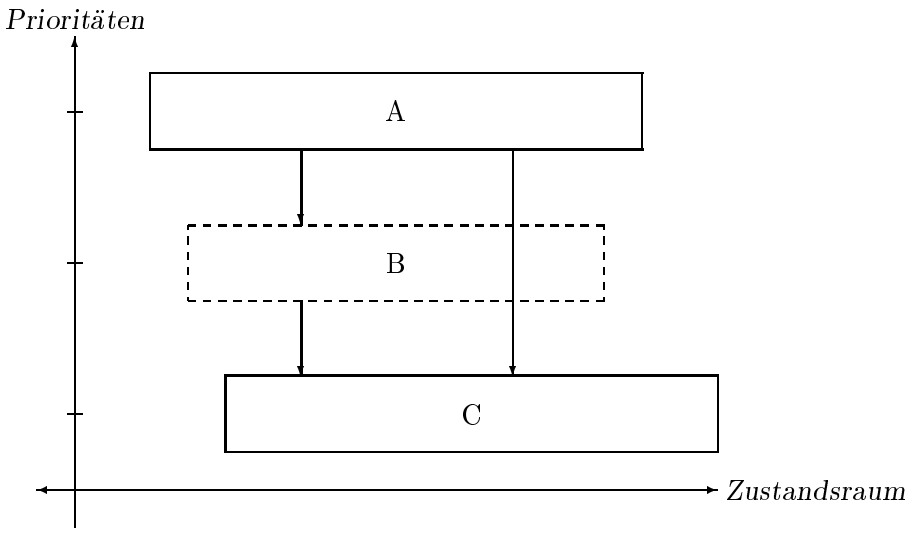


Abbildung 2.25: Transitivität von Überlappungen (für konkrete Zustände)

Problematischer ist jedoch die **syntaktische Wirkung** der Entfernung einer nicht schaltbaren Transition. So kann eine Transition nicht immer einfach aus dem Spezifikationstext entfernt werden, ohne die Syntax zu verletzen oder die Semantik zu verändern. Es gibt drei Konstellationen, unter denen die Entfernung einer nicht schaltbaren Transition zu Problemen führen kann:

- In Transitionen mit explizit definierten Hauptzuständen muß es auch eine explizite **Initialisierungstransition** geben. Jedoch werden solche Transitionen wie bereits erwähnt bei der Analyse immer als ausführbar betrachtet und müssen daher auch nie entfernt werden.

- In **geschachtelten Transitionen** haben einige oder alle Klauseln einer Transition auch für die folgenden Transitionen implizite Gültigkeit. So gilt in den Transitionen[70] aus Abbildung 2.26 die TO-Klausel von `t1` auch für die folgenden Transitionen `t2` und `t3`. Wird die Bedingung `b1` zusammen mit dem Hauptzustand S_1 als unerfüllbar erkannt und entfernt man daraufhin die Transition `t1` einschließlich aller ihrer Klauseln (womöglich sogar incl. des „`TRANS`“) aus der Spezifikation, so führt dies zu massiven Semantikänderungen für die folgenden Transitionen oder sogar zur syntaktischen Ungültigkeit der Spezifikation.

  Zur Lösung dieses Problems müssen zunächst die betroffenen Transitionsgruppen aufgelöst werden, indem jeder Transition *explizit* die für sie

[70] Man beachte das Fehlen des Schlüsselwortes „`TRANS`“ vor `t2` bzw. `t3`.

```
STATES S1, S2;
VAR    b1, b2: BOOLEAN;
....
TRANS
   TO S2
   PROVIDED b1
   FROM S1
   NAME t1:
      BEGIN
      END;

   PROVIDED b2
   NAME t2:
      BEGIN
      END;

   PROVIDED OTHERWISE
   FROM S2
   NAME t3:
      BEGIN
      END;
....
```

Abbildung 2.26: Transitionsgruppen und „`OTHERWISE`“

gültigen Klauseln zugeordnet werden. Dies führt zu keiner Semantikänderung, da die Transitionsgruppen lediglich eine abgekürzte Schreibweise dieser expandierten Form darstellen.

In der Implementation des Analysewerkzeugs ist dieser Schritt jedoch nicht nötig, da in dem bearbeiteten Objektformat[71] die Expansion bereits stattgefunden hat[72]. Dadurch ist die Entfernung von Transitionen aus Transitionsgruppen in dieser Hinsicht unproblematisch.

- Mit der Klausel „`PROVIDED OTHERWISE`“ wird innerhalb von Transitionsgruppen Bezug auf die PROVIDED-Klauseln der vorhergehenden Transitionen genommen. Sie ist äquivalent zur Klausel „`PROVIDED NOT (`$\mathtt{b}_1$ `OR ... OR` $\mathtt{b}_n$`)`“, wobei die $\mathtt{b}_i$ $(i = 1 \ldots n)$ die PROVIDED-Ausdrücke dieser vorherigen Transitionen sind. Wird eine dieser Transitionen entfernt, so ändert sich auch die Bedeutung der OTHERWISE-Klausel.

  In Abbildung 2.26 ist die PROVIDED-Klausel von Transition `t3` äquivalent zu „`PROVIDED NOT (b1 OR b2)`“. Wird die Transition `t1` wie im

[71] siehe Abschnitt 1.3

[72] Die Klauseln wurden dabei nicht kopiert, sondern die Transitionen haben nur (evtl. mehrfache) Referenzen auf ihre Klauseln. Die Klauseln „gehören“ jedoch dem Modulrumpf und sind daher durch das Entfernen von Transitionen nicht gefährdet.

obigen Beispiel entfernt (diesmal unter korrekter Behandlung der übrigen Klauseln), so führt dies zu einer abgeschwächten PROVIDED-Klausel von `t3`: „`PROVIDED NOT (b2)`“. Da `b1` im Hauptzustand S_2 jedoch möglicherweise erfüllbar ist (s.o.), ergibt sich damit eine Semantikverschiebung für `t3`.

Um dies zu umgehen, muß die OTHERWISE-Klausel vor dem Entfernen einer betroffenen Transition in der o.g. Weise expandiert werden. Dadurch kann eine Veränderung der Semantik der Klausel verhindert werden.

Das Analysewerkzeug kann also durch die Berücksichtigung der obigen Punkte eine nicht ausführbare Transition entfernen, ohne dadurch die Semantik der Spezifikation zu verändern.

Kapitel 3

Implementation des Analysewerkzeugs

In diesem Kapitel wird ein Überblick über die Implementation des Analysewerkzeugs und die verwendeten Algorithmen gegeben.[1] Obwohl die Implementation den Hauptanteil dieser Arbeit ausmacht, sollen an dieser Stelle nur die Grundkonzepte vermittelt werden, mit deren Hilfe die Orientierung im Quelltext erleichtert wird.

Zu allen weiterführenden Details sei an dieser Stelle auf den Quelltext verwiesen. Es sollte aufgrund der Bemühungen um eine klare Strukturierung und der durch die Objektorientierung erreichten hohen Abstraktion möglich sein, schnell ein Verständnis für die Abläufe bei der Analyse zu erlangen. Eine Übersicht über das technischen Umfeld ist in Kapitel 1.3 zu finden.

3.1 Allgemeine Konventionen

In diesem Abschnitt werden kurz einige Programmierkonventionen skizziert, die, soweit möglich, bei der Kodierung berücksichtigt wurden. Dabei dienen diese Regeln in erster Linie der leichteren Einarbeitung; sie haben es jedoch auch dem Autor gelegentlich ermöglicht, den Überblick über den eigenen Code wiederzugewinnen. Vor diesem Hintergrund scheint der erhöhte Aufwand, der sich durch die Anwendung dieser Regeln ergibt, sicherlich gerechtfertigt.

Eine ganz wesentliche Konvention lautet, alle übergebenen **Objekte als konstant zu deklarieren**, soweit ihr Wert eben nicht verändert werden soll. Speziell bei sehr komplexen Objektoperationen schützt dieses Vorgehen vor unerwünschten und zu Teil kaum überschaubaren Seiteneffekten. Der sich daraus ergebende Overhead ist jedoch leider nicht unerheblich, da viele der einfacheren Objektmethoden doppelt implementiert werden müssen, um adäquate Methoden für konstante und nicht-konstante Objektinstanzen anzubieten.

[1] zum gesamten Kapitel siehe [Str86, BaTö90, PlBr90, SiSt91a, SiSt91b]

Zum Beispiel muß die Funktion `owner()`[2], mit der der „Besitzer“ eines Listenknotens ermittelt werden kann, doppelt angelegt werden, um zu einem *const*-Knoten auch nur dessen *const*-Liste erhalten zu können:

```
...
const AListT<...>* owner() const {return pOwner;}
AListT<...>*       owner()       {return pOwner;}
...
```

Hier kann vom Compiler geprüft werden, welche Zugriffe auf den Besitzer eines (konstanten) Knotens überhaupt erlaubt sind! Bei konsequentem Einsatz dieser Vorgehensweise können viele Flüchtigkeitsfehler (auch konzeptioneller Art) schnell erkannt werden.

Leider wird in der PET-Klassenbibliothek praktisch nirgends mit const-Objekten gearbeitet, wodurch zudem konstante Referenzen auf PET-Objekte praktisch nicht sinnvoll genutzt werden können, da bei konstanten Objektinstanzen nur const-Memberfunktionen benutzt werden dürfen. Jedoch gerade die durch dieses Vorgehen unerkannt gebliebenen Fehler im PET verdeutlichen, wie wertvoll der durchgängige Einsatz von const-Attributen im Sinne der Qualitätssicherung sein kann.

Eine zweite wichtige Konvention lautet, möglichst **keine mehrfachen Referenzen** auf das selbe Objekt zu führen. Dies führt zwar an einigen Stellen zu einem erhöhten Speicherbedarf, ermöglicht es jedoch, referenzierte Objekte beim Ungültigwerden einer Referenz mitzulöschen, ohne auf mögliche andere Verweise Rücksicht nehmen zu müssen. Auch in diesem Fall resultieren einige kritische Fehler der PET-Klassenbibliothek darauf, daß zugunsten einer besseren Speicher- und Laufzeiteffizienz auf ein solches Vorgehen verzichtet wurde.

Diese Konvention erlaubt, zusammen mit konsequent eingeführten copy-Konstruktoren[3], äußerst mächtige Operationen mit höchster Abstraktion auszuführen. Ein schönes Beispiel dazu ist die Invertierung einer disjunktiven Normalform[4], die aufgrund der verschachtelten Listenstrukturen in nur wenigen Zeilen realisiert wird.

Zuletzt soll noch eine Regelung zur der Benennung von Variablen und Strukturelementen genannt werden. Hier wurde soweit möglich im Variablennamen der Typ des Datums durch ein **Präfix** angedeutet. Verwendet wurden dabei die folgenden Kürzel:

b	für `BOOL`-Variablen (semantische Konvention, formal gleich `int`)
sz	für nullterminierte Strings
n	für vorzeichenbehaftete Ganzzahlen
u	für vorzeichenlose Ganzzahlen
f	für Gleitkommazahlen

[2] Memberfunktion von `AListNodeT<...>` im Modul atools.cxx des Analyseprogramms

[3] copy-Konstruktoren erzeugen eine Kopie eines Objekts. Diese Kopie kann einstufig oder – wie hier propagiert – vollständig (rekursiv) implementiert werden.

[4] Memberfunktion `ADnf* ADnf::inverted()` im Modul `adnf` des Analyseprogramms

Zusätzlich werden die Präfixe `p` bei Zeigern und `a` bei Arrays angegeben. Dazu einige Beispieldeklarationen für Variablen:

```
BOOL bChanges;
char* pszName;
int* apnSize[10];
```

Bei lokalen Schleifenzählern (z.B. `i, pAkt`) und bei Variablen von zusammengesetzten Typen wird auf ein Präfix jedoch gelegentlich verzichtet (mit Ausnahme des `p` bei Zeigern, das immer benutzt wurde).

3.2 Modulstruktur

Der Quelltext gliedert sich hierarchisch in die vier Module *atools, adnf, aexpr* und *analyse*, wobei jedes der Module ausschließlich von den darunterliegenden abhängt (siehe Abbildung 3.1). Insbesondere sind nur die beiden obersten Module (*aexpr* und *analyse*) von der PET-Klassenbibliothek abhängig.

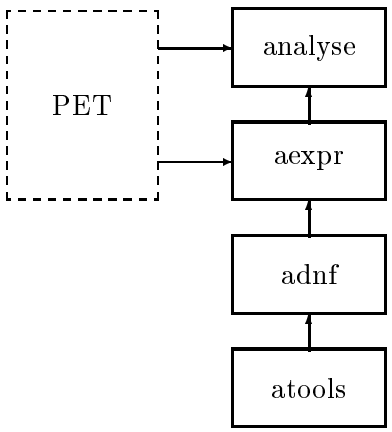


Abbildung 3.1: Modulstruktur des Analysewerkzeugs

Zur leichteren Erkennung sind alle zum Analysewerkzeug gehörenden Dateien (ebenso wie die Namen aller darin neu eingeführter C++-Klassen) mit einem führenden „*a*“ versehen. Zusammen mit dem make-file setzt sich das Analysewerkzeug daher aus folgenden Dateien zusammen:

`analyse.mak`		*Makefile des Projekts*
`analyse.hxx`	`analyse.cxx`	*Steuerung der Analyse*
`aexpr.hxx`	`aexpr.cxx`	*Verwaltung von Expressions*
`adnf.hxx`	`adnf.cxx`	*Verwaltung von DNFs*
`atools.hxx`	`atools.cxx`	*allgemeine Hilfsklassen*
`atools.def`		*Template-Implementation*

Die Aufgaben der einzelnen Module und die darin definierten Klassen und Funktionen werden in den folgenden Abschnitten genauer dargestellt.

3.2.1 Das Modul *atools*

Das Modul *atools* stellt einige allgemeine Klassen und Funktionen bereit, die in allen anderen Modulen Verwendung finden.

Die wichtigste Aufgabe ist die Bereitstellung von leistungsfähigen Klassen zur Verwaltung **doppelt verketteter Listen** (`AList` und `AListNode`). Diese erlaubt neben den üblichen Listenoperationen auch das Kopieren der gesamten Liste mitsamt der enthaltenen Daten, die Ausgabe eines Dumps der gesamten Liste in einen C++-Stream und die korrekte Behandlung von const-Attributen für Listen und Listenknoten.[5]

Der Struktur der Listenverwaltung liegt die bereits oben erwähnte Idee zugrunde, daß keine mehrfachen Referenzen auf ein Objekt bestehen sollen. Durch die sich daraus ergebende *1:n-Beziehung* zwischen Objektbesitzer und den Objekten selbst ist es möglich, die Objekte 1:1 mit den Listenknoten zu verbinden. Dies geschieht in C++ am einfachsten dadurch, daß die abhängigen Objekte **als Erben der Klasse AListNode** angelegt werden. Dabei ist es sehr einfach möglich, virtuelle Memberfunktionen von AListNode zu überladen, um dadurch Listeneigenschaften anzupassen. Folgende Memberfunktionen von AListNode kommen dafür in Frage:

- `virtual AListNode* copy() const;`
  Diese Funktion wird benutzt, um eine komplette Liste kopieren zu können. Dabei liefert die überladene Funktion eine vollständige Kopie der Objektinstanz.

- `virtual ~AListNode();`
  Durch den virtuellen Destruktor wird beim Löschen der Liste jeder Knoten korrekt abgebaut.

- `virtual ostream& dumpNode(...) const;`
  Gibt einen Dump eines Listenelements aus; über AList::dumpList(...) kann dann ein Dump der ganzen Liste erzeugt werden.

- `virtual BOOL insertBefore(const AListNode& node) const;`
  Gibt gewünschte Reihenfolge beim sortierten Einfügen in die Liste an.

Zur Verwaltung von typisierten Zeigern existieren zusätzlich die Klassen-Templates `AListT` und `AListNodeT`. Diesen Klassen wird als Template-Argument ein Zeigertyp übergeben, der dann typsicher als Nutzdatum verwaltet wird.[6] So erzeugt z.B. die Definitionen `AListT<const char *> StringList;` eine Liste von Zeigern auf (konstante) Strings. Dabei ist eine Typprüfung der Knoten- und Nutzdaten möglich, sodaß in diese Liste nur Knoten des Typs `AListNodeT<const char *>` eingefügt werden können und z.B. der Zugriff auf

[5] Zwar wird in der PET-Klassenbibliothek bereits eine doppelt verkettete Liste implementiert, jedoch unterstützt diese keine der anfangs geforderten Eigenschaften.

[6] Genauer: `AList` (bzw. `AListNode`) ist eine Spezialisierung von `AListT` (bzw. `AListNodeT`).

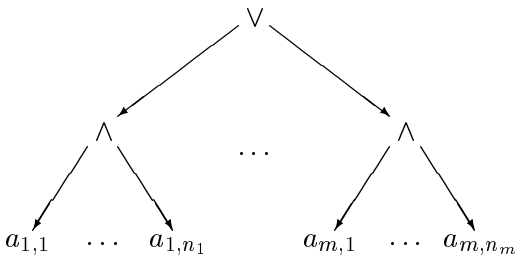
den ersten Zeiger der Liste über `StringList.first().data()` einen Zeiger mit korrektem Typ (im Beispiel „`const char *`“) liefert.

Neben den o.g. Klassen zur Listenverwaltung, dem Typ `BOOL` (Alias für `int` zur expliziten Darstellung von Wahrheitswerten) und den Konstanten `TRUE` und `FALSE` wird in diesem Modul noch die Klasse `AIndent` zur leichteren Verwaltung von *inkrementellen Einrückungen in Ausgaben* bereitgestellt. Dazu kann an Funktionen, die Ausgaben mit Einrückungen machen sollen, eine Instanz dieses Objekts übergeben werden, welche diese Einrückung (als String) beschreibt. Ruft diese Funktion nun eine andere Funktion auf, deren Ausgabe weiter eingerückt werden soll, so übergibt sie als Indent-Instanz z.B. das temporäre Objekt `AIndent(`*indent*`, "\t")`, wordurch eine relative Vergrößerung der Einrückung um den angegebenen String (hier ein Tab-Zeichen) erfolgt. Diese Methode wird später zur formatierten Ausgabe des Reports genutzt.

3.2.2 Das Modul *adnf*

Das Modul *adnf* ist ein von der PET-Klassenbibliothek völlig unabhängiges Modul, das lediglich auf der Listenverwaltung von *atools* aufbaut. Es stellt drei Klassen zur Verfügung, mit deren Hilfe **disjunktive Normalformen** (**DNF**s) auf hohem Abstraktionsniveau verwaltet werden können. Analog zu deren Definition (Abbildung 3.2) bilden die drei Klassen folgende Funktionalitäten:

- Referenzen auf (ausagenlogische) Atome (*class AAtom*)
- Die Konjunktion solcher Referenzen (*class AConAtom*)
- Die Disjunktion solcher Konjunktionen (*class ADnf*)



$a_{i,j} \in \{a, \overline{a} \mid a \text{ ist AL-Atom}\}$

Abbildung 3.2: Struktur der DNF

Dabei enthält ein Objekt vom Typ `ADnf` eine Liste von Objekten vom Typ `AConAtom`, die als ∨-verknüpft gelten. Die `AConAtom`-Instanzen enthalten wiederum jeweils Listen von `AAtom`-Instanzen, die als ∧-verknüpft gelten. Zu diesem Zweck beerben `ADnf` und `AConAtom` die Klasse `AList` (sie *enthalten* Listen), die Klassen `AConAtom` und `AAtom` beerben die Klasse `AListNode` (sie *sind* in Listen *enthalten*). Die Vererbungshierarchie ist in Abbildung 3.3 dargestellt.

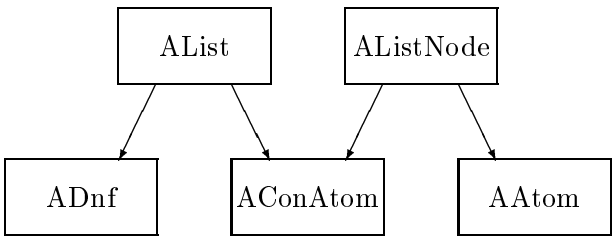


Abbildung 3.3: Vererbungshierarchie in `adnf.cxx`

Da in diesem Modul keine Informationen über Struktur und Typ der aussagenlogischen Atome vorliegen[7], werden die aussagenlogischen Atome durch **Zahlenwerte** vom Typ `unsigned` repräsentiert. Zwei Atome gelten dabei genau dann als gleich, wenn ihre Zahlenwerte übereinstimmen. Im (darauf aufbauenden) Modul *aexpr.cxx* werden diese Zahlenwerte später mit Ausdrücken vom Typ *AExpr* assoziiert werden.

Die Klasse **AAtom** realisiert eine Referenz auf ein solches Atom. Da die Atome *negiert* oder *nicht negiert* in der DNF auftreten können, wird neben der Zahlenrepräsentation der atomaren Aussage zusätzlich noch ein Flag geführt, das eine eventuelle Negation anzeigt. Dies spiegelt sich u.a. auch in dem Ergebnis des Vergleichsoperators „`==`“ zwischen zwei Objekten A_1 und A_2 vom Typ `AAtom` wieder:

$$(A_1 \texttt{ == } A_2) \rightarrow \begin{cases} +1 : A_1 \equiv A_2 \\ -1 : A_1 \equiv \overline{A_2} \\ 0 : \textit{sonst} \end{cases}$$

Die Klassen `AAtom` und `AConAtom` werden an dieser Stelle nicht näher behandelt, da auf sie (bis auf zwei Ausnahmen im Modul `aexpr`) nur über das Interface von `ADnf` zugegriffen wird.

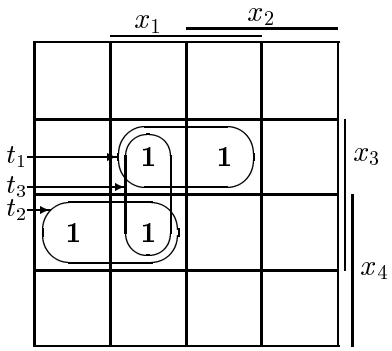
Die Klasse **ADnf** stellt mächtige Funktionen zur Bearbeitung der disjunktiven Normalformen bereit. So können ihre Instanzen mit nur einem Kommando vollständig kopiert, wieder gelöscht oder z.B. über Operatoren wie „`A &= B`“ (Konjunktion) und „`A |= B`“ (Disjunktion) verknüpft werden. Dabei werden sie vor dem Ende jeder Operation jeweils minimiert[8], um danach mit Prädikaten wie „`BOOL ADnf::isTrue()`“ oder „`BOOL ADnf::isFalse()`“ auf Konstante Ergebnisse getestet werden zu können. Man kann sogar über die Funktion „`ADnf* ADnf::inverted() const`“ eine negierte Form der DNF erzeugen.[9]

Als Besonderheit sei noch die Klasse `Minimizer` erwähnt, die innerhalb von `ADnf` deklariert ist. Sie dient einzig dazu, die Ausführung von Minimierungsschritten immer dann sicherzustellen, wenn eine Funktion *von außen* (d.h.

[7] Dieses Modul basiert lediglich auf `atools.cxx` (siehe Abbildung 3.1).

[8] Die Minimierung ist (zugunsten kurzer Ausführungszeiten) nicht ganz vollständig. Unter Umständen können Nicht-Primimplikanten (siehe Abbildung 3.4) enthalten sein. Diese beeinträchtigen jedoch nicht die Erkennung von konstanten Ergebnissen und können zudem bei Bedarf mit `void ADnf::removeNonPrimImp()` explizit entfernt werden.

[9] Da die Invertierung relativ aufwendig ist, wird danach jeweils eine Kopie der invertierten Form in einer Art lokalem *Cache* („`ADnf* ADnf::pInvertedCache`“) gespeichert.



$$\rightsquigarrow \underbrace{(x_1 \cdot x_3 \cdot \overline{x_4})}_{=:t_1} + \underbrace{(\overline{x_2} \cdot x_3 \cdot x_4)}_{=:t_2} + \underbrace{(x_1 \cdot \overline{x_2} \cdot x_3)}_{=:t_3}$$

t_3 ist kein *Primimplikant*, da er in $t_1 + t_2$ bereits enthalten ist. Da er jedoch weder in t_1 noch in t_2 *alleine* enthalten ist, ist er nur relativ aufwendig als solcher zu erkennen. `void ADnf::removeNonPrimImp()` erkennt und entfernt ihn jedoch.

Abbildung 3.4: Nicht-Primimplikant, der nicht automatisch minimiert wird

nicht aus `ADnf` selbst heraus) aufgerufen wurde. Dazu wird am Anfang jeder öffentlichen Memberfunktion eine Instanz dieses Objekts auf dem Stack angelegt. Diese Instanz wird niemals referenziert; ihr einziger Zweck ist es, bei ihrer Instanziierung (Funktionsaufruf) einen Verschachtelungszähler (`int ADnf::nNestedMinimize`) zu erhöhen und bei ihrer Zerstörung (Verlassen der Funktion) diesen Zähler wieder zu verringern, um dann gegebenenfalls (Verschachtelungszähler geht auf Null) die Minimierung auszulösen. Dadurch wird die Minimierung nur dann aufgerufen, wenn die Kontrolle an externe Funktionen zurückgegeben wird.[10] Diese Lösung hat neben ihrer Kompaktheit den Vorteil, daß der Destruktor vor dem Verlassen der Funktion garantiert[11] aufgerufen wird, ohne daß z.B. bei einer `return`-Anweisung die Minimierung „vergessen“ werden könnte.

3.2.3 Das Modul *aexpr*

Das Modul *aexpr* stellt die Funktionalitäten bereit, mit denen **Ausdrücke** analysiert werden können. Dazu gehören

- die *Vereinfachung* von Estelle-Ausdrücken,
- die Anwendung von *Substitutionen*, einschließlich der *Expansion von Funktionsaufrufen*,
- die Umwandlung von bool-wertigen Ausdrücken in *disjunktive Normalformen* und
- die Erkennung von *Widersprüchen* in den Konjunktionen von disjunktiven Normalformen.

[10] Die Minimierungsfunktion wird an einigen Stellen (Invertierung) zusätzlich explizit aufgerufen, um die Größe des Ausdrucks nicht übermäßig wachsen zu lassen.

[11] auch bei `return-` oder `throw`-Operationen, nicht jedoch bei `longjump(...)` (wird nicht benutzt!)

Interne Repräsentation von Ausdrücken

Die in der PET-Klassenbibliothek definierten Strukturen zur Darstellung von Ausdrücken erweisen sich bei näherer Betrachtung als ungeeignet für die Durchführung der Ausdrucksanalyse. Ihre starre Struktur macht dynamische Änderungen, wie sie für Substitutionen nötig sind, sehr schwierig. Zudem macht es die große Anzahl von unterschiedlichen Ausdrucksklassen extrem aufwendig, den Ausdrucksbaum zu durchlaufen[12]. Deshalb wurde die Klasse **AExpr** angelegt, die alle Typen von Ausdrücken in sich vereinigt. Dies wurde der Definition eines Klassenbaumes vorgezogen, da bei der Substitution häufig im Laufe des Umbaus eines Ausdrucks Zuweisungen von Ausdrucksknoten nötig sind. So wird zum Beispiel der Ausdruck $\neg(\neg(a = b))$ zu $a = b$ substituiert. Dabei muß jedoch die Adresse des Ausdrucks an Top-Stelle gleich bleiben, da evtl. externe Referenzen auf den Ausdruck bestehen. So bleibt in Abbildung 3.5 die Adresse des Top-Operators bei der Substitution unverändert, sodaß die Referenz durch den externen Zeiger P zutreffend bleibt.

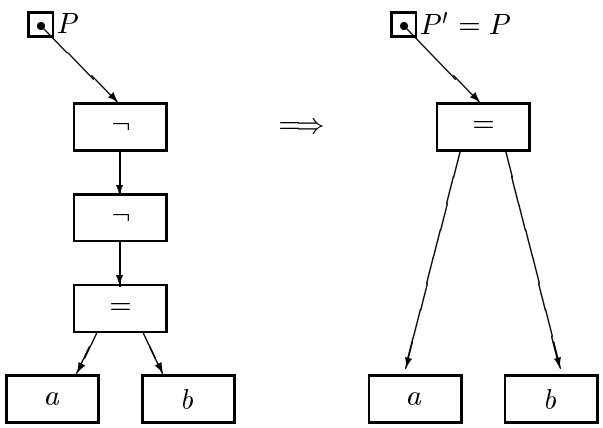


Abbildung 3.5: Beibehaltung der Adressen bei Substitutionen

Aus diesem Grund muß die Struktur aller Ausdruckstypen identisch sein, um eine solche Zuweisung eines Objekts (im Beispiel der „=“-Knoten) an den Speicherplatz eines anderen (im Beispiel der obere „¬“-Knoten) in jedem Fall möglich zu machen. Ausdrücke werden daher durch einen Baum von `AExpr`- und `AExprNode`-Instanzen gebildet. `AExprNode` ist dabei eine Fusion von `AExpr` und `AListNode`, sodaß seine Instanzen in die Liste der Argumente eines Objekts vom Typ `AExpr` eingehängt werden können (siehe Abbildung 3.6).

`AExpr` enthält neben der Liste der Teilausdrücke nur noch *lokale* Informationen, d.h. Teilausdrücke befinden sich *ausschließlich* in der enthaltenen Liste von Teilausdrücken. Dadurch sind Rekursionen durch den gesamten Ausdrucksbaum extrem einfach, da der Operator des jeweiligen Knotens keine Rolle für die Rekursion spielt. So stellt der Code-Ausschnitt in Abbildung 3.7 bereits die

[12] Da in der PET-Klassenbibliothek praktisch keine virtuellen Funktionen benutzt werden, muß jede der über 100 Klassen jeweils separat behandelt werden, um alle Teilausdrücke durchlaufen und bearbeiten zu können.

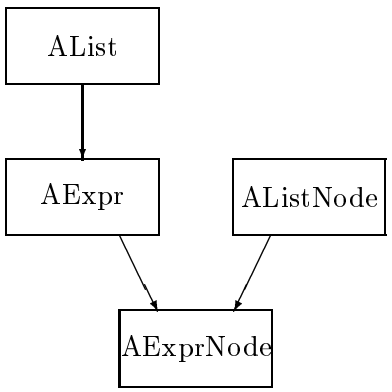


Abbildung 3.6: Die Vererbungshierarchie von `AExpr` und `AExprNode`

vollständige Implementierung der *Substitution von Variablen*[13] durch beliebige Ausdrücke dar.

```
void AExpr::substVar(const AObjectRef* pVarRef, const AExpr& expr) {
    for (AExprNode* pNode = first() ; pNode ; pNode = next(pNode))
        pNode->substVar(pVarRef, expr);
    if ( ((nType == VARATOM_DEF) || (nType == ANYVARATOM_DEF))
    &&   (pRef == pVarRef) )
        *this = expr;
    subst_local();
}
```

Abbildung 3.7: Die Implementation von `void AExpr::substVar(...)`

Man beachte hier auch die einfache Zuweisung an den aktuellen Ausdruck (`*this = expr`). Der Inhalt des aktuellen Objekts wird dabei einschließlich seiner Teilausdrücken zerstört und an der selben Adresse durch eine Kopie von `expr` ersetzt. Dieser Zuweisungsoperator ist auch stabil gegen die Zuweisung von eigenen Teilausdrücken. Dadurch kann nach der Erkennung der Konstellation in Abbildung 3.5 (zwei ¬-Operatoren in Folge) die Entfernung der beiden Operatoren durch die Anweisung „`*this = *first()->first();`“ vollständig behandelt werden.[14]

Die wichtigsten *lokalen Informationen* in `AExpr`-Objekten sind:

- Der Typ des Rückgabewerts des Ausdrucks (`AExpr::pSynType`)
- Der Typ des Operators (`AExpr::nType`); dieser wird durch einen Wert aus einem Aufzählungstyp dargestellt. Werte sind zum Beispiel:

[13] genauer: Objekt-Referenzen auf Variablen, s.u.

[14] Der ¬-Operator ist einstellig, die in den ¬-Objekten enthaltenen Listen enthalten deshalb immer genau einen Knoten vom Typ `AExprNode`, nämlich den negierten Ausdruck. `first()->first()` referenziert daher im obigen Beispiel den Teilausdruck, der = als Top-Operator hat.

  - `ORD_CONST` für konstante Werte vom Typ `INTEGER` oder von Teilbereichs- und Aufzählungstypen (auch `BOOLEAN`).
  - `UNARY_OP`, `BINARY_OP` und `NARY_OP` für ein- und zweistellige Operatoren bzw. für Operatoren mit variabler Stelligkeit.
  - `FUNC_CALL` für Aufrufe an benutzerdefinierte Funktionen
  - usw.

- Abhängig vom Wert in `nType` *spezielle Parameter* zum jeweiligen Operator in einer `union`-Struktur. So wird hier z.B. der Wert einer Konstanten oder die ID des referenzierten Objekts bei Funktionen oder Variablen angegeben.

Soviel an dieser Stelle zur Behandlung von Ausdrücken. Es handelt sich dabei um den komplexesten Teil des Programms, und in der gebotenen Kürze kann natürlich nur ein kleiner Ausschnitt beleuchtet werden. Im Folgenden werden nun noch die Themen *Scoping*, Zuordnung von Ausdrücken zu *DNF-Aussagen* und *Widerspruchserkennung* behandelt.

Scoping von Objektreferenzen

Bei der Analyse ist es nicht immer korrekt, daß Bezüge auf die gleiche *Variablendefinition* auch tatsächlich die gleiche *Variableninstanz* referenzieren. Zum Beispiel wird in Abbildung 3.8 von den beiden ANY-quantifizierten Transitionen die selbe Definition von i genutzt. Bei der Analyse ergibt sich daraus die Interferenzbedingung $(i = a) \wedge (i \neq a)$, wobei mit i zweimal die identisch selbe Variablendefinition referenziert wurde. Diese Bedingung ist unerfüllbar, obwohl in Wirklichkeit für jeden zulässigen Wert von a Interferenzen zwischen Instanzen der beiden Transitionen auftreten. Die Ursache für diese Fehlinterpretation ist, daß in Wirklichkeit die beiden Auftritte von i in der Formel keine Abhängigkeit voneinander haben: Wird die verschachtelte Transition expandiert, so erhält man zwei Transitionen, in denen zwei völlig unabhängige Laufvariablen in den ANY-Klauseln definiert werden.

Um nun die Referenzen auf die verschiedenen Instanzen von i in der Formel unterscheiden zu können, wird als Identifikation der Variableninstanz nicht die Adresse ihrer Definition, sondern eine künstlich geschaffene *Zwischenebene* genutzt. Dazu dienen Instanzen der Klasse **AObjectRef**, die statt der Variablen referenziert werden (siehe Abbildung 3.9). Diese enthalten dann erst einen Verweis auf die Variablendefinitionen. Die entsprechende Formeldarstellung lautet dann $(@1 = @3) \wedge (@2 \neq @3)$, wobei `@1` und `@2` Instanzen von i sind und `@3` eine Instanz von a ist. Dieser Ausdruck spiegelt die tatsächlich referenzierten Variableninstanzen korrekt wider.

Doch wie wird nun erkannt, daß in den beiden Transitionen *verschiedene* Instanzen von i, jedoch die *gleichen* Instanzen von a referenziert werden? Dazu wird beim Analysieren einer Transition in `ATransition::ATransition(...)`[15] eine

[15] Im Modul `analyse`.

```
TYPE SubRange = 1 .. 10;
VAR  a : SubRange;

TRANS
   ANY i: SubRange
   PROVIDED i = a
      BEGIN
      END;

   PROVIDED i ≠ a
      BEGIN
      END;
```

Abbildung 3.8: Zwei verschiedene Instanzen einer (ANY-) Variablen

Instanz der Klasse `AScope` auf dem Stack angelegt. Über diese kann mit Hilfe der Member-Funktion `AScope::addRef(...)` die Deklaration von ANY-Variablen in einer ANY-Klausel bekannt gemacht werden. Wird vor der Termination der Instanz von `AScope` (automatisch beim Ende der Analyse der Transition) eine Referenz auf diese Variable angefordert, so werden *transitionslokale* Referenzen geliefert. Werden Referenzen auf Objekte angefordert, die in dieser Umgebung nicht über `AScope::addRef(...)` angemeldet wurden, so werden diese aus einem Pool von global gültigen Objektreferenzen befriedigt (bzw. es werden bei Bedarf dort neue angelegt).

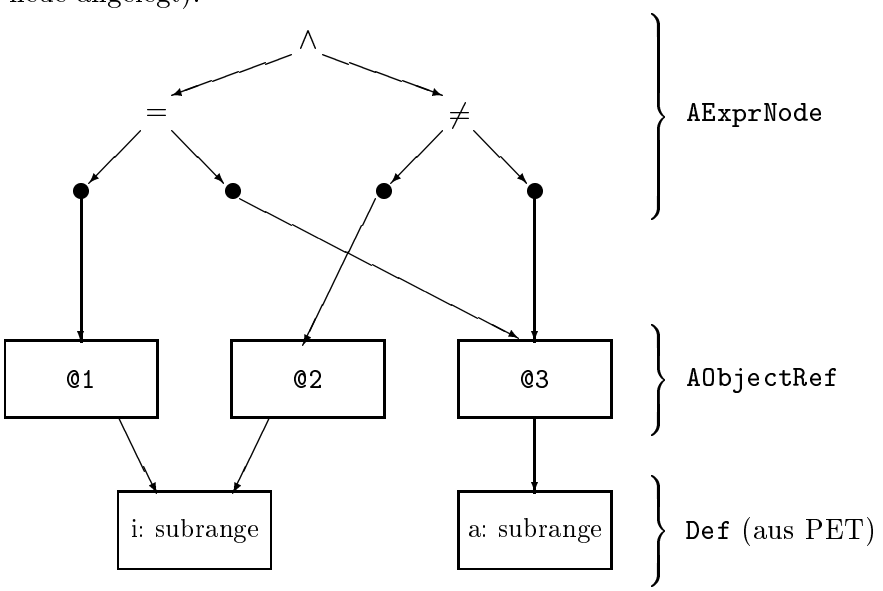


Abbildung 3.9: Indirekter Bezug zu Variablendefinitionen über `AObjectRef`

Das Anfordern einer Objektreferenz erfolgt über `getRef(...)`. Man übergibt der Funktion einen Zeiger auf eine Objektdefinition (z.B. auf die von a) und erhält einen Zeiger auf eine Instanz von `AObjectRef` (z.B. auf @3). Zwei Ob-

jektinstanzen sind genau dann gleich, wenn sie durch die selbe Instanz von `AObjectRef` vertreten werden.

Diese lokale Bindung von Objektreferenzen durch Instanziierung von `AScope` auf dem Stack wird neben den Transitionen auch bei `EXIST`-Ausdrücken und bei der Expansion von Funktionsaufrufen (für die formalen Parameter) benutzt.

Dieser Mechanismus wird auch dazu genutzt, die verschiedenen Aufrufe von **indeterministischen Funktionen** unterscheiden zu können (siehe 2.4.3 und 2.4.4). Dazu wird für jedes Auftreten von Referenzen auf solche Funktionen jeweils eine neue Objektreferenz generiert, auch wenn bereits in der gültigen Umgebung eine Referenz existiert. Dadurch werden die einzelnen Funktionsaufrufe unterscheidbar, auch wenn sie sich innerhalb eines Ausdrucks befinden (z.B. $f(\ldots)$ in $f() \neq f()$).

Zuordnung von Ausdrücken zu DNF-Atomen

Wie bereits in Abschnitt 3.2.2 erläutert, werden die *aussagenlogisch atomaren Ausdrücke* in den disjunktiven Normalformen durch Zahlenwerte repräsentiert. Natürlich muß es eine bijektive Abbildung auf die zugehörigen Aussagen[16] geben, um von einem DNF-Atom (Instanz von `AAtom`) zu seinem Ausdruck zu kommen, bzw. um zu einem Ausdruck eindeutig ein Atom zu erhalten. Insbesondere sollen äquivalente Ausdrücke auch zum selbe Atom gehören.

Zu diesem Zweck wird die Klasse `ABoolAtomAssocList` benutzt, die genau diese Zuordnung realisiert. In vielen Funktionen, die auf die atomaren Ausdrücke einer DNF zugreifen, muß daher eine solche Assoziationsliste übergeben werden.

Widerspruchserkennung in DNFs

Die Erkennung der Widersprüchlichkeit einer DNF wurde bereits in Abschnitt 2.4.5 auf die Analyse ihrer einzelnen Konjunktionen zurückgeführt. Es genügt also, zu einer gegebenen Menge von Aussagen festzustellen, ob diese *zugleich*[17] *erfüllbar* sind. Dies wird von der Klasse `AContradictionTester` realisiert.

Dazu wird einer Instanz dieser Klasse mit einer Folge von Aufrufen der Funktion `void ContradictionTester::add(const AExpr& expr, BOOL bNotNeg)` die Menge der zu prüfenden Aussagen (negiert oder nicht negiert) bekannt gemacht, um dann mit der Funktion `BOOL AContradictionTester::isContradictory()` die eigentliche Untersuchung zu starten.

Die grundsätzliche Vorgehensweise bei dieser Widerspruchserkennung wurde in Abschnitt 2.4.5 erläutert. Die dazu notwendige Verwaltung der Äquivalenzklassen von Ausdrücken wird durch die Klassen `ACtEqClass` (eine Äquivalenzklasse) und `ACtEqClassSet` (eine Menge von Äquivalenzklassen) realisiert.

[16] genauer auf die zugehörige Äquivalenzklasse von Ausdrücken
[17] also unter der gleichen Belegung

Die Beziehungen *zwischen* den Äquivalenzklassen werden durch die Klassen `ACtRelationPair` und `ACtRelationSet` dargestellt.

Über die Funktion `AContradictionTester::dump(...)` kann schließlich auch eine Begründung für die Widersprüchlichkeit oder Nicht-Widersprüchlichkeit der Konjunktion ausgegeben werden.

3.2.4 Das Modul *analyse*

Das Modul *analyse* führt aufbauend auf den vorherigen Modulen die eigentliche Analyse durch. Dazu werden nacheinander die folgenden Schritte ausgeführt:

- Kommandozeilen-Argumente interpretieren
- Objektdatei der Spezifikation einlesen
- Analyse durchführen
- Analyse-Report ausgeben
- Nicht ausführbare Transitionen entfernen
- Modifizierte Objektdatei der Spezifikation anlegen

Das Einlesen und Schreiben des Objektformats wird dabei von Funktionen der PET-Klassenbibliothek übernommen. Man erhält nach dem Lesen einen Zeiger auf eine Instanz der Klasse `SpecDef`.[18] Diese beschreibt die Spezifikation selbst und stellt damit die Wurzel der Modulhierarchie dar.

Die Analyse selbst wird von den Klassen `AModuleNode` und `ATransition` gesteuert. Dazu wird in der Funktion `void analyse(PSpecDef p)` eine Instanz von AModuleNode angelegt, wobei als Argument für den Konstruktor der oben genannte `SpecDef`-Zeiger übergeben wird. Die Instanz erzeugt daraufhin zu allen Untermodulen des übergebenen Moduls rekursiv wiederum Instanzen von `AModuleNode`[19] und sammelt diese in der Liste `AModuleNode::ChildModList`, um danach zu jeder Transition des Moduls eine Instanz der Klasse `ATransition` anzulegen. Diese werden wiederum in einer Liste `AModuleNode::TransList` gesammelt. Dabei werden auch evtl. vorhandene Zusicherungen zum Modulrumpf erfaßt und der entsprechenden `AModuleNode`-Instanz zugeordnet.

Es entsteht ein **Baum von Modulrümpfen**, wobei jedem Modul eine **Liste von Transitionen** zugeordnet ist (siehe Abbildung 3.10). Dabei wird ebenfalls noch *im* Konstruktor von `AModuleNode` die komplette lokale Analyse des Moduls durchgeführt, sodaß nach der Instanziierung des o.g. Wurzelknotens nur noch

[18] Eine der Klassen aus der PET-Klassenbibliothek. Wie bereits erwähnt sind im Gegensatz dazu alle *im Analysewerkzeug* definierten Klassen daran zu erkennen, daß ihr Name mit einem „A“ beginnt.

[19] Die Modulrümpfe unterhalb von `SpecDef` haben den Typ `BodyDef`. Es gibt daher einen entsprechenden zweiten Konstruktor `AModuleNode::AModuleNode(AModuleNode* pParent, PBodyDef pBd)`.

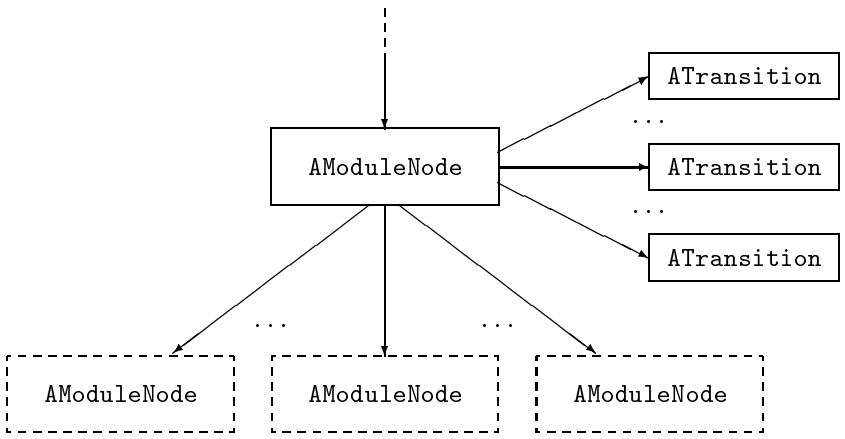


Abbildung 3.10: Hierarchie von `AModuleNode`-Instanzen

der Report ausgegeben bzw. die Entfernung der nicht ausführbaren Transitionen durchgeführt werden muß.

Zur Abbildung der Transitionsklauseln existieren die Klassen `AFrom`, `ATo`, `AWhen`, `ADelay`, `AProvided`, `AAny` und `APriority`. Diese kapseln den Zugriff auf die Details der Klauseln weitgehend ein. So gibt es jeweils Funktionen, die entscheiden, ob zwei typgleiche Klauseln zueinander *passen* und wie die dazu notwendige Bedingungen (meist in Form einer *disjunktive Normalform*) lauten.

Die Instanzen der Klasse `ATransition` enthalten zur Beschreibung der Transitionsklauseln jeweils eine Instanz von *jeder* der oben genannten Klassen, gleichgültig, ob die zugehörige Transition überhaupt diese Klausel besitzt oder nicht. Gibt es die entsprechende Klausel nicht, so zeigt die Klasseninstanz das entsprechende Defaultverhalten.

Bei der Prüfung, ob es zwischen zwei Transitionen Indeterminismus geben kann (durch `AModuleNode::CheckIndeterm(...)`) oder ob eine Transition von einer anderen überlappt wird (durch `ATransition::CheckOverlap(...)`) müssen daher lediglich zu jeder der sieben möglichen Klauseln die entsprechenden Objekte der beiden Transitionen „*befragt*“ werden, ob es Einwände gegen eine mögliche Interferenz gibt und welche Bedingungen die Interferenzsituation erfüllen muß. Die Konjunktion all dieser Bedingungen liefert dann gegebenenfalls die **Überlappungs-** oder **Indeterminismusbedingung**.

Eine Sonderrolle spielt hier lediglich die Klasse `AWhen`, da durch diese unter Umständen in den übrigen Bedingungen Variablensubstitutionen nötig werden (siehe Kapitel 2.3.3). Deshalb wird der Test zwischen den WHEN-Klassen als letztes durchgeführt und die bis dahin gebildete Interferenzbedingung zur Bearbeitung mit übergeben.

Zur Ermittlung aller möglichen Interferenzen werden also alle Transitionen eines Moduls jeweils paarweise in der oben geschilderten Weise anhand ihrer Klauseln verglichen und so auf Indeterminismus und Überlappungen geprüft. Die sich ergebenden Bedingungen werden dabei auch mit den Modul- und Transitionszusicherungen konjunktiv verknüpft und mit `AContradictionTester` auf

Widersprüche untersucht.

Die Instanzen von `ATransition` beinhalten jeweils eine Liste, in der zu jeder Interferenz ein Knoten vom Typ `AOverlapNode` eingehängt wird. In diesem wird die Art der Interferenz, die Bedingung für ihr Auftreten und eine Referenz auf die beteiligte zweite Transition festgehalten. Zudem wird zu jeder Transition eine *effektive PROVIDED-Bedingung* gebildet, indem ausgehend von ihrer expliziten PROVIDED-Bedingung alle Überlappungsbedingungen ausgeschlossen werden.[20] Ergibt sich dabei ein Widerspruch in der effektiven PROVIDED-Klausel, so ist die Transition nicht ausführbar und kann entfernt werden.

Die *Entfernung der nicht ausführbaren Transitionen* wird durch die Funktion `AModuleNode::removeDeadTransitions(...)` rekursiv für alle Module realisiert. Dabei werden auch die evtl. auf die PROVIDED-Klauseln solcher Transitionen aufbauenden OTHERWISE-Klauseln expandiert.

3.3 Erweiterungsmöglicheiten

Wie bereits an einigen Stellen angedeutet, gibt es noch vielfältige Möglichkeiten für Verbesserungen und Erweiterungen unterschiedlichen Umfangs:

- Erweiterungen des Termersetzungssystems, der Funktionsexpansion und der Widerspruchserkennung anhand von speziellen Problemstellungen (siehe Kapitel 2.4.3 bis 2.4.5).
- Differenziertere Behandlung von Delay-Klauseln (siehe Kapitel 2.3.4).
- Behandlung von mehreren unterschiedlichen Prioritäten bei den Transitionen eines Features. Dabei müssen auch Überlappungen *innerhalb* eines Features behandelt werden (siehe Kapitel 2.3.1).
- Analyse von EXIST-Ausdrücken. Diese werden zur Zeit nur auf Ausdrucksgleichheit geprüft.
- Handlichere Darstellung der DNFs als vollständig minimierte Terme. Eventuell sollte es zur besseren Lesbarkeit möglich sein, von Hand bestimmte Gruppierungsziele vorzugeben.
- Entfernung der nicht ausführbaren Transitionen direkt aus dem Quelltext. Im Gegensatz zur PDRESTORE-Lösung bleiben so die Kommentar- und Quelltextformatierungen erhalten. Besteht die Quelle aus mehreren Dateien, so kann dies aufwendig sein (siehe Kapitel 2.5).
- Erweiterte Möglichkeiten zur Formulierung von Zusicherungen (siehe Kapitel 2.3.7). Eventuell sollten diese abhängig von Hauptzuständen formuliert werden können oder auch WHEN-Bedingungen angegeben werden

[20] durch Konjunktion mit der negierten Form der Überlappungsbedingung (siehe Kapitel 2.3.5)

können. Unter Umständen wären auch temporallogische Operatoren wie „*es ist potentiell, daß ...*“ („*EF ...*“, siehe [Got94]) sinnvoll (aber schwer in die augenblickliche Weiterverarbeitung integrierbar).

- Expansion von EXIST-Ausdrücken und von Transitionen mit ANY-Klauseln bei kleinen (vollständig spezifizierten) Ordinalbereichen.
- Analyse der Transitionsblöcke. Dabei sind verschiedene Abstufungen denkbar. Eine der einfachsten Formen könnte darin bestehen, zu jeder Transition die Mengen der referenzierten und der (potentiell) veränderten Variablen sowie der abgegebenen Interaktionen aufzustellen. Mit Hilfe dieser Mengen wäre es einfach festzustellen, ob die Ausführungsreihenfolge zweier Transitionen (zwischen denen evtl. Indeterminismus besteht) überhaupt von Belang ist.[21]
- Globale Analyse der gesamten Spezifikation statt modullokaler Analyse.

Gerade die beiden letzten Punkte, die Analyse der Transitionsblöcke und die globale Analyse, werfen vielfältige prinzipielle Probleme auf. An dieser Stelle werden zur Ermittlung aussagekräftiger Analyseergebnisse (aufgrund der mangelnden Berechenbarkeit) wohl erhebliche Einschränkungen des zulässigen Spezifikationsstils notwendig sein. Auf jeden Fall besteht in diesem Gebiet noch erheblicher Forschungsbedarf.

3.4 Einsatz des Analysewerkzeugs

In diesem Abschnitt wird die Übersetzung und der praktischen Einsatz des Analyseprogramms kurz erläutert.

3.4.1 Übersetzung

Das Programm ist in C++ implementiert und benutzt Templates. Zur Übersetzung ist daher ein entsprechender Compiler notwendig. Bei der Entwicklung wurden aktuelle Versionen von g++ *(UNIX-Version des GNU-C++ V2.5.8)* und CSet/2 *(IBM OS/2 C++-Compiler V2.01)* benutzt. Eine MAKE-Datei für den GNU-Compiler ist den Quellen beigefügt. Ergebnis der Übersetzung ist dann eine ausführbare Datei `pdanalyse`.

Die Grundlage des Programms bildet die PET-Klassenbibliothek des PET-DINGO-Toolkits. Aufgrund vielfältiger Fehler in der Speicherverwaltung[22] der vorliegenden Versionen empfiehlt es sich dringend, als Sofortmaßnahme die

[21] In Estelle kann Nebenläufigkeit (insbesondere innerhalb eines Moduls) nur durch Nichtdeterminismus bei der Ausführungsreihenfolge spezifiziert werden.

[22] Es werden z.B. häufig Destruktoren mehrfach aufgerufen, da u.a. Objekte als Wert (statt als Referenz) übergeben werden, jedoch *copy-Konstruktoren* durchgehend fehlen und dadurch binäre Kopien der Objekte entstehen, die oft auch Zeiger auf (nicht duplizierte) abhängige Objekte enthalten.

Speicherfreigabe für PET-Objekte ganz zu unterdrücken. Dies ist am einfachsten möglich, indem man in der Funktion `void freeDef()` in der Datei `def.cxx` sämtliche Anweisungen bis auf `pd = 0` am Ende durch die Präprozessor-Anweisung `#if 0 ... #endif` deaktiviert. Dadurch wird verhindert, daß der von PET-Objekten belegte Speicher freigegeben wird. Da sowohl das Analysewerkzeug als auch PET und DINGO selbst nach relativ kurzer Zeit wieder terminieren, führt dieses Verfahren zu keinem merklich höheren Speicherbedarf, verhindert jedoch vielfältige Zugriffsfehler.

Langfristig sollten die häufig leicht zu behebenden Fehler von PET (nicht nur in der Speicherverwaltung) dennoch beseitigt werden.

3.4.2 Aufruf

```
>> pdanalyse: Analyse-Tool fuer Feature-Interactions in Estelle-Spezifikationen

>> Aufruf:   pdanalyse {-Option[+|-]} [-mModule] Obj_Infile [Obj_Outfile]

>> Option (Wert):
>>   short   (-) Nur Kurzfassung des Protokolls ausgeben
>>   oids    (-) Nur Variablennamen (-) oder exakte Objekt-ID (+) ausgeben
>>   petrest (-) PROVIDED-Ausdruecke auch mit PET-Funktionen ausgeben
>>   aexpr   (-) PROVIDED-Ausdruecke auch mit AEXPR-Funktionen ausgeben
>>   feature (+) Feature-Zugehoerigkeit beim Transitionsvergleich beachten
>>   effdnf  (+) Effektive erweiterte PROVIDED-Klausel ausgeben
>>   over    (+) Ueberlappungen melden
>>   indet   (+) Indeterminismus melden
>>   wunsover(-) auch (bzgl. WHEN) "unsichere" Ueberlappungen melden
>>   dunsover(-) auch (bzgl. DELAY) "unsichere" Ueberlappungen melden
>>   dxindet (-) auch (bzgl. DELAY) erweiterten Indeterminismus melden
>>   igniq   (-) "INDIVIDUAL QUEUE"-Attribut ignorieren
>>   indetf  (-) gibt es indeterministische Funktionen
>>   all         Alle obigen Optionen ein- (+) oder ausschalten (-)
```

Abbildung 3.11: Ausgabe des Programms bei Aufruf ohne Argumente

Wird das Analysewerkzeug ohne Parameter aufgerufen, so wird eine Kurzanleitung ausgegeben (siehe Abbildung 3.11). Ansonsten ist mindestens ein Argument anzugeben, nämlich der Dateiname der Binärdatei, die von PET erzeugt wurde (`Obj_Infile`).[23] Diese Datei wird nur gelesen und in keinem Fall verändert. Das Analyseprotokoll wird immer in den Standard-Ausgabestrom (`stdout`) ausgegeben und kann gegebenenfalls in eine Datei umgeleitet werden. Warnungen und Kontrollmeldungen werden in den Standard-Fehlerstrom (`stderr`) ausgegeben. Um zum Beispiel eine Estelle-Spezifikation in der Datei `test.stl` zu bearbeiten, sind folgende Aufrufe notwendig:

- `pet -o test.obj test.stl`
- `pdanalyse test.obj > test.rep`

[23] Zum Datenfluß siehe Abbildung 1.1

Die Übersetzung der Estelle-Spezifikation durch PET muß dabei fehlerfrei verlaufen sein, da es sonst unter bestimmten Umständen zum Absturz des Analysewerkzeugs kommen kann. Danach befindet sich in der Datei `test.obj` das von PET erzeugte Objektformat der Spezifikation und in `test.rep` der Analysereport in Textform. Kontrollmeldungen und Warnungen werden auf das Terminal ausgegeben.

Über den optionalen zweiten Parameter `Obj_Outfile` kann der Name einer Datei angegeben werden, in die die überarbeitete Version des Objektformats abgespeichert werden soll. Existiert bereits eine Datei diesen Namens, so wird sie überschrieben. Das Ergebnis entspricht einer Datei, wie sie von PET mit Hilfe der Option `-o` erzeugt werden kann, wenn in der Eingabespezifikation alle Transitionen korrekt[24] entfernt wurden, die im Protokoll als nicht ausführbar vermerkt sind. Wurden keine nicht ausführbaren Transitionen erkannt, so enthält die Datei dementsprechend eine Kopie der ursprünglichen Binärdatei. Wird dieses Argument nicht angegeben, so wird keine Binär-Ausgabedatei erzeugt.

Die übrigen Optionen (mit Ausnahme von `-m...`) sind „Schalter", d.h. die zugehörige Programmoption kann mit „`-`*Schlüsselwort*`+`" ein- und mit „`-`*Schlüsselwort*`-`" ausgeschaltet werden.[25] Die aktuelle (Default-) Belegung der Optionen wird in der Syntax-Übersicht angezeigt. Folgende Schalter werden z.Zt. unterstützt (die erste Variante ist jeweils Default):

- **short**
  Umfang des Protokolls einschränken?
  -: Komplettes Protokoll ausgeben
  +: Nur Kurzfassung (Übersicht) des Protokolls ausgeben

- **oids**
  Qualifizierung von Objekt-IDs (z.B. Variablen) einschränken?
  -: Nur Objektnamen nennen
  +: Name, Klasse und ID zu jeder Objektreferenz nennen

- **petrest**
  Transitionsklauseln zusätzlich mit Darstellungsfunktionen von PDRESTORE ausgeben?
  -: Nur Darstellungsfunktionen des Analysetools benutzen
  +: Zusätzlich Darstellung durch PDRESTORE-Funktionen ausgeben[26]

- **aexpr**
  PROVIDED-Bedingung mit Darstellungsfunktionen von `AExpr` *ausgeben?*
  -: Nur DNF- (und evtl PDRESTORE-) Darstellung ausgeben
  +: Zusätzlich Darstellung durch `AExpr`-Funktionen ausgeben

[24] Wie in Kapitel 2.5 dargestellt, müßten im Quelltext einige zusätzliche Änderungen vorgenommen werden.

[25] Folgt weder „+" noch „−", so entspricht dies dem „+"

[26] Kann aufgrund von Fehlern in PET gelegentlich zu Abstürzen führen!

- **feature**
  Feature-Zugehörigkeit beim Transitionsvergleich beachten?
  +: Nur Interaktionen zwischen Transitionen verschiedener Features testen
  −: Auch Interaktionen zwischen Transitionen gleicher Features testen

- **effdnf**
  Effektive (durch Überlappungen eingeschränkte) PROVIDED-Klausel ausgeben?
  +: Spezifizierte und effektive PROVIDED-Klausel ausgeben
  −: Nur spezifizierte PROVIDED-Klausel ausgeben

- **over**
  Erkannte (mögliche) Überlappungen melden?
  +: Meldungen zu Überlappungen ausgeben
  −: Keine Meldungen zu Überlappungen ausgeben

- **indet**
  Erkannten (möglichen) Indeterminismus melden?
  +: Meldungen zu Indeterminismus ausgeben
  −: Keine Meldungen zu Indeterminismus ausgeben

- **wunsover**
  bzgl. WHEN-Bedingungen unsichere Überlappungen melden?
  −: Keine WHEN-unsicheren Überlappungen melden
  +: auch WHEN-unsichere Überlappungen (gekennzeichnet) melden

- **dunsover**
  bzgl. DELAY-Bedingungen unsichere Überlappungen melden?
  −: Keine DELAY-unsicheren Überlappungen melden
  +: auch DELAY-unsichere Überlappungen (gekennzeichnet) melden

- **dxindet**
  bzgl. DELAY-Bedingungen erweiterten Indeterminismus melden?
  −: Auch bei DELAY nur Indeterminismus bei gleichen Prioritäten
  +: Alle Möglichkeiten von Indeterminismus bei DELAY-Klauseln melden

- **igniq**
  „INDIVIDUAL QUEUE“-Attribut ignorieren?
  −: Queue-Attributierung beachten
  +: Alle Queues als „COMMON QUEUE“ annehmen

- **indetf**
  Gibt es Funktionen mit indeterministischem Ergebnis?
  −: Alle in Klauseln referenzierten Funktionen sind deterministisch
  +: Ergebnisgleichheit bei mehrfachen Funktionsaufrufen unsicher

- **all**
  Sammelschalter: Alle Optionen einstellen.
  +: Alle Schalter auf „+“
  −: Alle Schalter auf „−“

Die letzte Option ist „`-m...`“, wobei für die Punkte der Name eines Modulrumpfes eingesetzt werden muß. Durch Angabe dieses Schalters kann das Protokoll auf ein Modul beschränkt werden. Dies bietet sich besonders bei langen Spezifikationen an, wenn Änderungen an einem einzelnen Modul untersucht werden sollen. Wird diese Option nicht angegeben, so werden alle Module im Protokoll berücksichtigt. Beispiel: „`-mServiceProviderBody`“.

3.4.3 Meldungen und Warnungen

Zur leichteren Handhabung des Analysewerkzeugs werden über den Standard-Fehlerstrom und damit normalerweise auf das Terminal begleitende Meldungen ausgegeben. Diese gliedern sich in Statusmeldungen, Fehlermeldungen und Warnungen. Alle Meldungen werden durch entsprechende Präfixe gekennzeichnet, um sie gegebenenfalls von Protokollmeldungen unterscheiden zu können[27].

Statusmeldungen

Statusmeldungen beschreiben den Fortschritt des Analysevorgangs. Sie sind am führenden „`>>`“ zu erkennen. Folgende Meldungen werden ausgegeben:

- „`>> Lesen der Eingabedatei` *Name*“
  Die Objektdatei wird eingelesen. Erfolgt unmittelbar nach dieser Meldung ein Absturz, so handelt es sich bei der angegebenen Datei wahrscheinlich nicht um eine korrekte PET-Objektdatei.[28]
- „`>> Analyse der Spezifikation` *Name*“
  Das Einlesen war erfolgreich, die Analyse wird gestartet.
- „`>> Ausgabe des Reports`“
  Der Report wird (in den Standard-Ausgabestrom) ausgegeben.
- „`>> Entfernung nicht ausfuehrbarer Transitionen`“
  Falls eine Zieldatei angegeben wurde, so werden nun die als nicht ausführbar erkannten Transitionen entfernt und weitere dazu notwendige Änderungen ausgeführt.
- „`>> Schreiben der Binaerdatei` *Name*“
  Falls eine Zieldatei angegeben wurde, so wird nun die optimierte Version der Objektdatei dorthin geschrieben.
- „`>> fertig`“
  Programm erfolgreich beendet.

[27] Es empfiehlt sich jedoch, (evtl. durch geeignete Umleitungen) das Protokoll von den übrigen Meldungen zu trennen.

[28] Leider werden Fehler beim Einlesen der Quelle durch die PET-Klassenbibliothek von dieser meist nur mit einem Absturz „behandelt“.

Fehlermeldungen

Es gibt nur eine (fatale) Fehlermeldung:

- „`!! FEHLER beim Lesen der Eingabedatei`“
  Die Eingabedatei ist fehlerhaft oder nicht vorhanden.

Warnungen

Warnungen beschreiben Auffälligkeiten, die bei der Analyse registriert wurden. Sie sind am führenden „`!! WARNUNG:`“ zu erkennen. Folgende Warnungen werden ausgegeben:

- „`!! WARNUNG: Nicht expandierbare Zusicherungsfunktion` *Name*“
  Zusicherungsfunktionen sind nutzlos, wenn sie nicht derart strukturiert sind, daß sie vom Analysewerkzeug zu einem geschlossenen Ausdruck expandiert werden können (siehe Kapitel 2.4.4).

- „`!! WARNUNG: Funktion` *Name* `ist keine Zusicherungsfunktion`“
  Der Name der Funktion enthält den String „assert“, es ist jedoch keiner der vorgeschriebenen Namen für Zusicherungsfunktionen. Evtl. liegt ein Schreibfehler vor, der die Zusicherung auch für das Analysewerkzeug zu einem Kommentar gemacht hat. Soll die Funktion keine Zusicherung sein, so kann die Meldung ignoriert werden.

- „`!! WARNUNG: Widerspruechliche PROVIDED-Klausel in Transition` *Name*“
  Die Provided-Klausel einer Transition selbst wurde bereits als unerfüllbar erkannt.

- „`!! WARNUNG: Zusicherungsfunktion in der INIT-Transition` *Name* `ignoriert`“
  In einer Initialisierungstransition wurde eine transitionslokale Zusicherung gefunden. Diese hat keinen Effekt.

- „`!! WARNUNG: Die Zusicherungen zu Transition` *Name* `sind unerfuellbar`“
  „`!! WARNUNG: Die Zusicherungen zu Modul` *Name* `sind unerfuellbar`“
  Es wurde eine Zusicherung gefunden, die selbst bereits unerfüllbar ist. Damit werden alle betroffenen Transitionen ohne weitere Prüfung als nicht ausführbar erkannt.

- „`!! WARNUNG: Die Zusicherungen zu Transition` *Name* `sind nutzlos (immer erfuellt)`“
  „`!! WARNUNG: Die Zusicherungen im Modulkoerper` *Name* `sind nutzlos (immer erfuellt)`“

Es wurde eine Zusicherung gefunden, die immer erfüllt ist. Diese ist nutzlos, da sie den (explizit bezeichneten) erreichbaren Zustandsraum nicht einschränkt.

- „`!! Unbekannte Option -`*Name* `ignoriert`“
  Es wurde eine unbekannte Option angegeben. Sie wird ignoriert.
- „`!! unerwartetes Argument` *Name* `ignoriert`“
  Es wurde eine unbekanntes Argument angegeben. Es wird ignoriert.
- „`!! WARNUNG: Modul` *Name* `nicht gefunden`“
  Es wurde kein Modul gefunden, das den in der Option „`-m...`“ angegebenen Namen hat. Es wird lediglich die Protokollübersicht ausgegeben.

3.4.4 Struktur des Analyseprotokolls

Zuletzt noch ein kurzer Überblick über das Format des Analyseprotokolls. Der Umfang der Protokollausgaben hängt dabei jeweils stark von den jeweiligen Optionen ab (s.o.).

In einem ersten Abschnitt werden die Aufrufparameter des Analyselaufs festgehalten:

```
**********************
*** Analyse-Report ***
**********************

* Eingabe-Datei: "test.bin"
* Optionen: -all+ -short- -igniq- -indetf-
```

Danach wird eine Übersicht über die Spezifikationsstruktur ausgegeben. Dabei werden die Transitionen zu jedem Modulrumpf aufgelistet und ggfs. als nicht ausführbar markiert. Dabei wird Transitionen ohne Namen eine automatisch erzeugte Identifikation „`anonymous(#)`“ (mit Modulweise fortlaufender Numerierung) zugeordnet.

```
*******************
* Zusammenfassung *
*******************

SPECIFICATION TestSpec
    TRANSITION anonymous(1)
    TRANSITION anonymous(2)
    TRANSITION MyTrans1 -- nicht ausfuehrbar --
    TRANSITION MyTrans2
    TRANSITION MyTrans3

BODY TestSpec.testBody
```

```
TRANSITION InitTrans
TRANSITION InTrans -- nicht ausfuehrbar --
TRANSITION OutTrans -- nicht ausfuehrbar --
```

Im dritten Abschnitt werden die Details der Analyse angegeben. Die Grundstruktur entspricht der der Zusammenfassung, jedoch werden zu den Modulrümpfen und den Transitionen Zusatzinformationen gegeben.

Dies sind zu den Modulrümpfen in erster Linie die *aussagenlogisch atomaren Aussagen*, die dann in den DNF-Formen der boolschen Schalt- und Interferenzbedingungen genutzt werden. Um die DNFs später kompakter darstellen zu können, werden diese atomaren Aussagen mit einer Kennung assoziiert, die dann später als Platzhalter für den Ausdruck benutzt wird. Eine Assoziationen lautet dann zum Beispiel „`%20 <--> (a = 1)`“. Im späteren Einsatz werden DNFs dann z.B. als „`((%20 AND %17) OR (%35))`“ formuliert.

Daneben werden noch diejenigen Hauptzustände aufgezählt, die als (potentiell) erreichbar erkannt wurden. Damit ergibt sich folgender Protokollabschnitt:

```
***********
* Details *
***********

SPECIFICATION TestSpec
    AUSSAGEN-ASSOZIATIONEN:
        %20 <--> (a{VAR@36} = 1)
        %21 <--> (b{VAR@37} = 2)
        %22 <--> (a{VAR@36} = 2)
        %23 <--> (c{VAR@38} = 3)
        %24 <--> (a{VAR@36} = 0)
        %25 <--> (a{VAR@36} = 123)
        %26 <--> (b{VAR@37} > 10)
        %27 <--> (b{VAR@37} > 11)
    (POTENTIELL) ERREICHBARE HAUPTZUSTAENDE: s1, s2, s3, s4
```

Im obigen Beispiel wurde auf die Option `-oids+` hin zu allen Objekten[29] eine eindeutige Objekt-ID angegeben. Diese wird unmittelbar hinter dem Objektnamen angegeben: „*Name*`{`*Typ*`@`*ID*`}`“. Dabei wird für *Typ* einer folgenden Werte ausgegeben:

`ANY_CONST`	z.B. „`CONST c = ANY INTEGER;`“
`ANY_VAR`	Variable aus `ANY`-Transitionsklausel
`VAR`	gewöhnliche Variable
`PREDEF`	Vordefinierte Funktionen wie z.B. `SUCC()`
`FUNC`	benutzerdefinierte Funktion
`FUNCPAR`	funktionaler Parameter

[29] Objekte: Variablen, Funktionen, ANY-Konstanten und ANY-(Klausel)-Variablen

ID ist eine Zahl, die eine eindeutige Instanzidentifizierung zuläßt (siehe Abschnitt 3.2.3). Zwei Instanzen sind dabei genau dann gleich, wenn ihre IDs gleich sind.

Im Anschluß an die Hauptzustandsliste und die Assoziationen des Moduls werden die Transitionen zusammen mit den ermittelten Eigenschaften (Transitionsklauseln, Interferenzen, effektive Schaltbedingung) dargestellt:

```
TRANSITION MyTrans1 -- nicht ausfuehrbar --
    FROM  s1
    TO  s2
    PRIORITY 5 == TestPrio
    ANY  i{ANY_VAR@35}: 0 .. TestPrio
    PROVIDED
      als PET-Restore:
         ((a = 1) and (b = 2))
      als AExpr-Ausdruck:
         ((a{VAR@36} = 1) AND (b{VAR@37} = 2))
      als DNF:
         ((%20 AND %21))
      als DNF (effektiv):
         FALSE (unerreichbar)
    DELAY  (-1)
    UNSICHER (DELAY): WIRD UEBERLAPPT DURCH MyTrans3 WENN
      ((%20 AND %21 AND %23))
    WIRD UEBERLAPPT DURCH MyTrans2 WENN
      ((%20 AND %21 AND NOT %24))
```

Der im obigen Beispiel gezeigte DELAY-Wert -1 deutet an, daß der Wert des DELAY-Ausdrucks nicht als konstanter Wert bekannt ist. In der PROVIDED-Klausel werden (bis zu) drei Darstellungsmethoden genutzt: Die Klausel wird über PET-Restore-Funktionen (aus PET-Klassenbibliothek), über AExpr-Funktionen (aus `aexpr.cxx`) und als DNF ausgegeben. Zusätzlich wird die *effektive* PROVIDED-Bedingung als DNF ausgegeben. Diese wird wenn nötig nach Hauptzustandsgruppen untergliedert:

```
als DNF (effektiv):
  FROM s3
      ((%25 AND NOT %26 AND NOT %27))
  FROM s4
      ((%25 AND NOT %27))
  FROM s2
      ((%25))
```

WHEN-Klauseln werden optional ebenfalls in den beiden Varianten PET-Restore und AExpr dargestellt:[30]

```
WHEN  ip_in[j; (i + 1)].msg
      ip_in[j{ANY_VAR@4}, (1 + i{ANY_VAR@3})].msg
```

[30] Unverträglich mit obigem Beispiel (DELAY!), deshalb hier separat.

Nach den Transitionsklauseln werden Meldungen über sichere und unsichere Überlappungen dieser Transition und die dazu notwendigen PROVIDED-Bedingungen angezeigt. Unter Umständen sind noch zusätzliche Bedingungen aus den WHEN-, FROM- und DELAY-Klauseln der beteiligten Transitionen zu beachten. Diese kann man jedoch leicht aus den Transitionsklauseln entnehmen. Bei der Auswertung der Überlappungsbedingungen zu effektiven PROVIDED-Klauseln werden diese Nebenbedingungen natürlich automatisch berücksichtigt.

Nachdem dann alle Transitionen des Moduls dargestellt wurden, folgt ggfs. danach das nächste Modul.

Kapitel 4

Zusammenfassung und Ausblick

Es wurde die Entwicklung eines Werkzeugs dargestellt, mit dessen Hilfe die statische Erkennung von Feature-Interaktionen anhand von möglichem Indeterminismus sowie die Entfernung von nicht ausführbaren Transitionen unterstützt wird. Da diese Probleme in dieser Form grundsätzlich unlösbar sind, wurden einige Einschränkungen nötig, die in erster Linie darin bestanden, die Transitionsblöcke aus der Untersuchung auszuschließen.

Zudem wurden zwei Strategien durchgehend verfolgt: Eine *offensive Meldestrategie* („melde alle Interferenzen, solange sie nicht sicher ausgeschlossen werden können“) und eine *defensive Optimierungsstrategie* („entferne nur solche Transitionen, die sicher niemals ausgeführt werden können“). Die Beachtung dieser Strategien und die Einführung des Begriffs der *effektiven Schaltbedingung* ermöglichten es, die Problematik auf die Frage nach der (sicheren) Widerlegbarkeit existenzquantifizierter Aussagen zurückzuführen. Dazu wurde zunächst gezeigt, wie sich alle notwendigen Informationen aus dem paarweisen Vergleich von Transitionen anhand der Transitionsklauseln gewinnen lassen. Die dabei gewonnenen Bedingungen für Interferenzen zwischen Transitionen bzw. für die effektiven Schaltbedingungen wurden im Anschluß in drei Stufen auf Erfüllbarkeit untersucht:

1. Bearbeitung und Vereinfachung mit Hilfe eines *Termersetzungssystems*, einschließlich Expansion von Funktionsaufrufen,
2. aussagenlogische Analyse und Bildung einer DNF,
3. prädikatenlogische Widerspruchsanalyse der einzelnen Konjunktionen der DNF.

Durch diese Vorgehensweise ist es gelungen, das komplexe Problem überschaubar zu zerlegen und zu implementieren. Die oben gemachten Einschränkungen und die grundsätzlich unvollständige Erkennung widersprüchlicher Ausdrücke

stellen dabei die eigentlichen Grenzen des Verfahrens dar. Um diese umgehen zu können wurde die Möglichkeit geschaffen, über die *Zusicherungen* dem Analysewerkzeug explizite Zusatzinformationen zur Verfügung zu stellen. Dadurch kann in vielen Fällen die Qualität der Analyse weiter verbessert werden, sodaß mehr nichtausführbare Transitionen bzw. mehr unerfüllbare Interferenzbedingungen automatisch als solche erkannt werden können. Zudem wurde gezeigt, wie diese Zusicherungen in der Testphase einer Spezifikation überprüft und dabei sogar zur Unterstützung der Tests eingesetzt werden können. Es wäre sicherlich nützlich, dieses Konzept weiter auszubauen.

Als ein weiteres Nebenergebnis wurde anhand der durch die Anwendung von *indeterministischen Funktionen* verursachten Probleme und Anomalien (siehe Kapitel 2.4.3) ein offensichtlicher Fehler im Estelle-Standard [ISO89] aufgedeckt.

Es haben sich im übrigen vielfältige *Erweiterungsmöglichkeiten* (siehe Kapitel 3.3) abgezeichnet, die von kleinen Anpassungen des Werkzeugs an spezielle Analyseprobleme (z.B. Erweiterungen des Termersetzungssystems oder der Widerspruchserkennung) bis hin zu weitaus komplexeren Ansätzen (z.B. globale Analyse unter Berücksichtigung der Semantik der Transitionsblöcke) reichen. Dabei bleibt im Bereich derartiger komplexer Analysen noch ein erhebliches Maß an Forschungsarbeit zu leisten.

Abschließend kann man festhalten, daß Estelle mit seiner starken Strukturierung relativ gut für derartige Analysen zugänglich ist. Es wurde gezeigt, wie auch ohne Berücksichtigung der Transitionsblöcke bereits ein beachtliches Maß an Informationen gewonnen werden kann. Als einer der Nachteile von Estelle hat sich dessen relativ komplexer Sprachumfang erwiesen. Eine etwas sparsamere Sprachdefinition und eine Beschränkung auf einfache und klare Kernkonzepte hätten sicherlich die Entwicklung des Analysewerkzeugs erheblich vereinfacht. Bezieht man verstärkt die Transitionsblöcke mit in die Analyse ein, so wird sich diese Tendenz sicherlich noch weitaus deutlicher abzeichnen. In jedem Fall bleibt als letzte Grenze jedoch die mangelnde Berechenbarkeit der gesamten Problemstellung einer vollständigen Analyse bei erweiterten endlichen Automaten. Eine Lösung kann hier nur darin bestehen, durch verstärkten Einsatz von geeigneten Spezifikationsmethoden die Analysierbarkeit von vornherein zu verbessern, anstatt beliebige Spezifikationen allgemein analysieren zu wollen.

Anhang A

Reduktionsordnungen

In diesem Kapitel wird die Definition der in Abschnitt 2.4.3 benutzten *rekursiven Pfadordnung* (RPO) angegeben. Zuvor werden noch einige zugrundeliegende Begriffe definiert. Dabei wird auf Beweise und gelegentlich auch auf eine *formal exakte* Definition verzichtet, da dazu wesentlich weiter ausgeholt werden müßte. Die informelle Form der Definitionen sollte jedoch zum Verständnis der Anwendung einer RPO genügen. Eine ausführliche formale Darstellung findet man in [Ave95].

Zunächst einige Definitionen:

- Eine Ordnung $>$ ist **Noethersch** *gdw*
  es gibt keine unendlich absteigende Kette $X_1 > X_2 > X_3 > \ldots$
- **Term(F,V)** ist die Menge aller Terme zur Signatur $sig = (S, F, \tau)$ und der *Variablenmenge* V. F ist dabei eine Menge von *Funktionssymbolen.*
- zu $t \in Term(F, V)$ gibt $\mathbf{Var(t)} \subseteq V$ die Menge der in t vorkommenden Variablen an.
- Eine **Regel** ist ein Paar (l, r) mit $l, r \in Term(F, V)$ und $Var(l) \supseteq Var(r)$.
  Schreibweise: $l \to r$
- Ein **Regelsystem** ist eine Menge von Regeln.
- Das Regelsystem R definiert die **(Reduktions-) Relation** $\Longrightarrow_R$ auf Term(F,V) durch
  $t_1 \Longrightarrow_R t_2$ gdw t_2 geht aus t_1 durch Anwendung einer Regel aus R hervor[1]
- Ein Regelsystem heißt **Noethersch**, wenn die zugehörige Relation $\Longrightarrow_R$ Noethersch ist.

[1] An dieser Stelle wird auf die umfassende Definition von $\Longrightarrow_R$ verzichtet.

Sei im folgenden $>$ eine Partialordnung auf $Term(F,V)$:

- Eine **Substitution** σ ist eine Funktion $\sigma : V \longrightarrow Term(F,V)$
  mit $\sigma(x) \equiv x$ für fast alle $x \in V$.
  σ wird fortgesetzt zu $\sigma : Term(F,V) \longrightarrow Term(F,V)$ durch
  $\sigma(f(t_1,\cdots,t_n)) = f(\sigma(t_1),\cdots,\sigma(t_n))$

- $>$ ist **stabil** *gdw*
  $>$ ist **verträglich mit Substitutionen** *gdw*
  $(s > t) \Longrightarrow (\sigma(s) > \sigma(t))$ für alle Substitutionen σ und $s,t \in Term(F,V)$

- $>$ ist **monoton**
  gdw
  $>$ ist **verträglich mit der Termstruktur**
  gdw
  $(s_i > t_i) \Longrightarrow (f(s_1,\cdots,s_n) > f(t_1,\cdots,t_n)$
  für alle $s_i, t_i, f(s_1,\cdots,s_n), f(t_1,\cdots,t_n) \in Term(F,V)$ und $i = 1 \ldots n$

- Eine **Reduktionsordnung** $>$ ist eine Noethersche, stabile und monotone Partialordnung auf Term(F,V).

Satz:
Ein Regelsystem R ist genau dann Noethersch, wenn es eine Reduktionsordnung $>$ gibt mit

$$\forall (l \to r) \in R : \; l > r$$

Die Definition der Reduktionsordnung stellt dabei sicher, daß mit jeder *Regelanwendung* der ganze Term verkleinert wird. Dies ermöglicht es, die Überprüfung auf die *Regeln* zu beschränken, anstatt alle Instanzen prüfen zu müssen.

Definition $\gg$:
Sei $>$ eine Partialordnung zur Menge M. Die **Multimengenordnung** $\gg$ ist auf Mult(M) definiert durch

$$A \gg B \qquad \text{gdw.} \qquad \forall y \in (B - A) : \; (\exists x \in (A - B) : \; x > y)$$

Definition $\underset{\sim}{\gg}$:
Sei $\underset{\sim}{>}$ eine Quasiordnung auf einer Menge M und $\approx \; = \; \underset{\sim}{>} \cap \underset{\sim}{<}$ die zugrundeliegende Äquivalenzrelation. Dann ist $\underset{\sim}{\gg}$ auf Mult(M) definiert durch

$$\begin{aligned} A \underset{\sim}{\gg} B \text{ gdw. } \exists X, Y \in Mult(M),\; x_i, y_i \in M,\; n \in N : \; & x_i \approx y_i \text{ f.a. } i = 1 \ldots n \wedge \\ & A = X \cup \{x_1,\cdots,x_n\} \wedge \\ & B = Y \cup \{y_1,\cdots,y_n\} \wedge \\ & X \gg Y \end{aligned}$$

Definition $\sim_{perm}$:
Sei $\underset{\sim}{>}$ eine Quasiordnung auf F und $\approx = \underset{\sim}{>} \cap \underset{\sim}{<}$ die zugrundeliegende Äquivalenzrelation. Die zugehörige **Permutationskongruenz** $\sim_{perm}$ ist auf $Term(F,V)$ rekursiv definiert durch

$f(s_1,\ldots,s_n) \sim_{perm} g(t_1,\ldots,t_m)$
gdw
$f \approx g \wedge n = m \wedge$ es existiert Permutation π mit $(s_i \sim_{perm} t_{\pi(i)}$ für $i = 1 \ldots n)$

Als nächstes wird nun das eigentliche Ziel der Definitionen angegeben, die RPO zu einer vorgegebenen Präzedenz. Die RPO ist eine *Reduktionsordnung* und liefert damit eine hinreichende Terminationsbedingung für die Anwendung des Regelsystems aus Kapitel 2.4.3.

Definition $>_{rpo}$:
Sei $\underset{\sim}{>}$ eine Quasiordnung auf F und $\approx = \underset{\sim}{>} \cap \underset{\sim}{<}$ die zugrundeliegende Äquivalenzrelation. Die Ordnung $\underset{\sim}{>}$ wird dabei auch als **Präzedenz** bezeichnet. Die zugehörige **Rekursive Pfadordnung** (**RPO**) $>_{rpo}$ ist auf $Term(F,V)$ rekursiv definiert durch

1.) $s \equiv f(s_1,\cdots,s_n) >_{rpo} g(t_1,\cdots,t_m) \equiv t$ gdw

	$s_i \underset{\sim rpo}{>} t$	für ein i	(α)
oder	$(f > g) \wedge (s >_{rpo} t_j)$	für alle j	(β)
oder	$(f \approx g) \wedge (\{s_1,\cdots,s_n\} \underset{\sim rpo}{\gg} \{t_1,\cdots,t_m\})$		(γ)

2.) $t >_{rpo} x$ für alle $x \in Var(t)$ mit $x \not\equiv t$

$\underset{\sim rpo}{>}$ sei dabei die Quasiordnung zu $>_{rpo}$ und der Äquivalenzrelation $\sim_{perm}$ (beide jeweils zur Präzedenz $\underset{\sim}{>}$)

Abbildungsverzeichnis

1.1 Das Analysewerkzeug im Datenfluß von PET-DINGO 8

2.1 Modul-Baumstruktur einer Estelle-Spezifikation 12
2.2 Überlappungen und Indeterminismus zwischen Transitionen . . . 13
2.3 Transitionen mit vergleichbaren (`t1` mit `t2`) und unvergleichbaren Prioritäten (`t1`/`t2` mit `t3`) 17
2.4 Spezialisierte (Feature-) Namen für Prioritätskonstanten 18
2.5 Zustandsübergangsgraph zu einer Transitionsmenge 19
2.6 Möglichkeit von Interferenzen mit WHEN-Klauseln 22
2.7 IP-Indexgleichheit bei Überlappungen 24
2.8 Indeterminismus durch optionale Schaltbarkeit 24
2.9 Indeterminismus durch unbekannte Zeitskalierung 25
2.10 Keine Überlappungsmöglichkeit durch asynchronen Timer-Ablauf 26
2.11 Interferenzbedingungen aus PROVIDED-Klauseln 27
2.12 Vollständige Überlappung von `t3` 28
2.13 Nicht expandierbare ANY-Transition 30
2.14 Analysierbare (`t2`/`t1`) und nicht analysierbare (`t3`/`t1`) Überlappungen bei ANY-Klauseln 30
2.15 Bindung von ANY-Variablen durch WHEN-Klauseln 32
2.16 Beispiel für eine Zusicherungsfunktion in einer Transition 36
2.17 Gemischte Zustands-/Bedingungs-Überlappung 37
2.18 Erweiterung zur gemischten Zustands-/Bedingungs-Überlappung 38
2.19 Probleme bei der Erkennung einer Ausdrucksäquivalenz 43
2.20 Definition einer indeterministischen Funktion 44
2.21 OTHERWISE-Anomalie bei indeterministischen Bedingungen . . 46
2.22 Expandierbare Funktion mit einfacher Struktur 51
2.23 Definition eines varianten Records 52

2.24 Totale Erweiterung eines partiellen Ausdrucks 52
2.25 Transitivität von Überlappungen (für konkrete Zustände) 58
2.26 Transitionsgruppen und „`OTHERWISE`“ 59

3.1 Modulstruktur des Analysewerkzeugs 63
3.2 Struktur der DNF 65
3.3 Vererbungshierarchie in `adnf.cxx` 66
3.4 Nicht-Primimplikant, der nicht automatisch minimiert wird . . . 67
3.5 Beibehaltung der Adressen bei Substitutionen 68
3.6 Die Vererbungshierarchie von `AExpr` und `AExprNode` 69
3.7 Die Implementation von `void AExpr::substVar(...)` 69
3.8 Zwei verschiedene Instanzen einer (ANY-) Variablen 71
3.9 Indirekter Bezug zu Variablendefinitionen über `AObjectRef` . . . 71
3.10 Hierarchie von `AModuleNode`-Instanzen 74
3.11 Ausgabe des Programms bei Aufruf ohne Argumente 77

Literaturverzeichnis

[Ave95] J. Avenhaus *Reduktionssysteme.* Springer-Verlag (1995)

[BaTö90] F. Bause, W. Tölle *C++ für Programmierer.* Vieweg-Verlag (1990)

[Bör87] E. Börger *Berechenbarkeit, Komplexität, Logik.* Vieweg-Verlag (1987)

[Bow+89] T. F. Bowen et al. *The feature interaction problem in telecommunication systems.* In „Seventh IEEE International Conference on Software Engineering for Telecommunication Systems“ (July 1989)

[BrGo94a] J. Bredereke, R. Gotzhein *A Case Study on Specification, Detection and Resolution of IN Feature Interactions with Estelle.* Interner Bericht Nr. 245/94, Fachbereich Informatik, Universität Kaiserslautern (Mai 1994)

[BrGo94b] J. Bredereke, R. Gotzhein *Specification, Detection and Resolution of IN Feature Interactions with Estelle.* In „Seventh International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols — FORTE '94, Proceedings“, IFIP TC6/WG6.1, Berne, Switzerland (October 1994)

[Cam+94] E. J. Cameron et al. *A Feature Interaction Benchmark in IN and Beyond* In „Feature Interactions in Telecommunications Systems“, IOS Press (1994)

[Got94] R. Gotzhein *Spezifikation von Kommunikationssystemen* Skript zur Vorlesung, Fachbereich Informatik, Universität Kaiserslautern (1994)

[ISO89] ISO/TC 97/SC 21, ISO 9074 *Estelle – A Formal Description Technique Based on an Extended State Transition Model.* (1989)

[ITU93] ITU-T *Q12xx-Series Intelligent Network Recommendations* (1993)

[PlBr90] P. J. Plauger, J. Brodie *Referenzhandbuch Standard C.* Vieweg-Verlag, Braunschweig (1990)

[Put88] E. v. Puttkamer *Digitale Logik.* Skript zur Vorlesung, Fachbereich Informatik, Universität Kaiserslautern (1988).

[Sie90] D. Siefkes *Formalisieren und Beweisen.* Vieweg-Verlag, Braunschweig (1990)

[SiSt91a] R. Sijelmassi, B. Strausser *The Distributed Implementation Generator: an overview and user guide.* Technical Report NCSL/SNA-91/3, National Institute of Standards and Technology, Gaithersburg, USA (January 1991)

[SiSt91b] R. Sijelmassi, B. Strausser *The Portable Estelle Translator: an overview and user guide.* Technical Report NCSL/SNA-91/3, National Institute of Standards and Technology, Gaithersburg, USA (January 1991)

[Str86] B. Stroustrup *The C++ Programming Language.* Addison-Wesley Publishing Company (1986)

Universität Kaiserslautern
Fachbereich Informatik
AG Rechnernetze

Prof. Dr. R. Gotzhein

Kaiserslautern, den 25. April 1995

Diplomarbeit Joachim Thees, Matr. Nr. 229246
Betreuer: Jan Bredereke

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig und nur mit den angegebenen Quellen und Hilfsmitteln verfaßt zu haben.

Joachim Thees