Integration von Zusicherungen in Estelle

Sascha Bötzel

25. Juni 1996

Zusammenfassung

Estelle ist eine formale Beschreibungstechnik (FDT) zur Spezifikation von verteilten, informationsverarbeitenden Systemen.
In dieser Arbeit werden Konzepte vorgestellt, mit deren Hilfe es möglich wird, zur Laufzeit Informationen über den Hauptzustand eines Moduls und über dessen Interaktionspunkte zu erhalten, die über das hinausgehen, was bisher mit Hilfe der `FROM` und `WHEN` Klauseln zu erreichen war. Diese Erweiterungen von ESTELLE sind kompatibel zu dem bisherigen ESTELLE–Standard. Mit Hilfe dieser Informationen wird das von Thees und Bredereke [The95] vorgeschlagene Prinzip von Zusicherungen erweitert und es wird gezeigt, wie auch eine Überprüfung von Zusicherungen zur Laufzeit der Spezifikation möglich ist. Weiter wird ein Konzept angegeben, mit dem temporallogische Ausdrücke in ESTELLE umgesetzt werden können. Auch diese Ausdrücke werden in einem weiteren Schritt in das Konzept der Zusicherungen eingegliedert.
In einem Beispiel werden die vorgestellten Konzepte in ein ESTELLE–Modul integriert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Kontext

Estelle ist eine formale Beschreibungstechnik (FDT) zur Spezifikation von verteilten, informationsverarbeitenden Systemen. Mit einer FDT ist es möglich, das wesentliche eines Systems abstrakter zu beschreiben, als dies mit konventionellen Programmiersprachen möglich wäre.
Trotzdem ist es möglich, aus ESTELLE–Spezifikationen z.B. mit Hilfe der Werkzeuge PET und DINGO [Sij91] automatisch eine Implementierung zu generieren. Weiterhin erlaubt es die formal definierte Semantik der FDT, bestimmte Eigenschaften einer Spezifikation mathematisch nachzuweisen.
Da Estelle eine operationale Sprache ist, ist sie zwar gut implementierbar, aber es lassen sich manche wichtigen Eigenschaften eines Systems nur implizit ausdrücken, so daß man bei der automatischen Analyse von Spezifikationen schwer auf sie zugreifen kann. Daher wäre es hilfreich, wenn man Invarianten oder Vorbedingungen explizit spezifizieren könnte. Joachim Thees stellt in [TheBr95] eine Methode vor, Zusicherungen über ESTELLE–Zustände in eine Spezifikation zu integrieren. In dieser Arbeit wurden vornehmlich statische Zusicherungen betrachtet, d.h. Zusicherungen, die vor der Laufzeit des Programmes benutzt wurden, und Methoden, um nicht ausführbare Transitionen zu erkennen und auszufiltern. Es wurden zwar auch Möglichkeiten aufgezeigt, Aussagen über Eigenschaften der ESTELLE–Spezifikation zu machen, und diese zur Laufzeit der Spezifikation zu überprüfen, aber es konnten bisher keine Aussagen über Hauptzustände beziehungsweise Interaktionspunkte und Warteschlangen gemacht werden. Mit Hilfe dieser Erweiterungen des Zusicherungsprinzips kann die Zuverlässigkeit eines Systems weiter gesteigert werden, da zwei voneinander unabhängige Spezifikationen gleichzeitig erfüllt werden müssen, die operationale ESTELLE–Spezifikation und die eigenschaftsorientierten Zusicherungen.

1.2 Aufgabenstellung

ESTELLE (beschrieben in [ISO89]) basiert auf dem Konzept der erweiterten endlichen Automaten. Dabei werden Kontextvariablen eingeführt, mit deren Hilfe Zustände dieses endlichen Automaten repräsentiert werden können. Dadurch vermeidet man den explosionsartigen Anstieg der Hauptzustände, der die Spezifikation mit Hilfe endlicher Automaten bisher behinderte. Zusicherungen werden bisher [The95, 2] nur über Kontextvariablen gemacht. D.h. über Variablen, die modulglobal definiert sind und mit deren Hilfe be-

stimmte Eigenschaften einer Spezifikation beschrieben werden können, ohne dafür eigene Hauptzustände einführen zu müssen. Diese Zusicherungen werden durchgeführt, indem boolesche Funktionen definiert werden, in denen Intervalle festgelegt werden, in deren Schranken sich die Werte dieser Kontextvariablen bewegen. Diese Funktionen werden während der Laufzeit der Implementation niemals aufgerufen. Damit sie dennoch von einem Tool zur Analyse von *Feature–Interaktionen*[1] erkannt und benutzt werden konnten, werden für diese Funktionen bestimmte Namen vergeben (s. [The95, 2]).
In dieser Arbeit werden Konzepte herausgearbeitet, mit deren Hilfe man Zusicherungen nicht nur über Kontextvariablen, sondern über den gesamten Zustandsraum von ESTELLE, mit Ausnahme der Modul- und Kommunikationsstrukturen, machen kann. Diese Arbeit gliedert sich im wesentlichen in drei Teile:

- Es wird ein Konzept entwickelt, mit dessen Hilfe zur Laufzeit Informationen über den Hauptzustand der aktuellen Modulinstanz und über ihre Interaktionspunkte für verfeinerte Spezifikationen und Analyse–Werkzeuge erreichbar gemacht werden. Ferner sollen sowohl statische als auch dynamische Zusicherungen über Hauptzustände und Interaktionspunkte möglich sein, ohne daß die Spezifikation nicht mehr den normalen ESTELLE–Syntaxregeln entspricht.

- Weiterhin wird ein Konzept entwickelt, wie man nicht nur prädikatenlogische Formeln als Zusicherungen spezifizieren kann, sondern auch Aussagen über den gesamten Ausführungsablauf. Dazu sollen Konzepte zur Implementierung einfacher temporallogischer Operatoren angegeben werden. Es ist jeweils abzuschätzen, inwieweit es möglich ist, diese Operatoren zur Laufzeit zu Testzwecken auf ihre Einhaltung zu überprüfen.

- In einem Beispiel wird die Bedeutung der Zusicherungen verdeutlicht.

2 Zusicherungen in ESTELLE bisher

Das Konzept der *Zusicherungen* in ESTELLE wurde erstmals in [The95] erwähnt. In dieser Arbeit wurde das Problem der Feature-Interaction untersucht, d.h. den Kollisionen, die entstehen, wenn mehrere, unabhängig von einander entwickelte Protokollmaschinen zu einem Protokoll zusammengefügt

---

[1] Unter Feature Interaktionen versteht man sich überschneidende Funktionalitäten bei verschiedenen Modulen eines Dienstes.
So kann es beispielsweise vorkommen, daß bei einer Telefonanlage, sowohl Gespräche umgeleitet werden sollen, als auch bestimmte Anrufer gesperrt sind.

werden. In diesem Zusammenhang erschien es sinnvoll, zusätzlich zur normalen ESTELLE–Spezifikation weitere Angaben machen zu können, die normalerweise implizit in jeder Spezifikation enthalten sind, die aber nicht ohne weiteres von Analyse–Werkzeugen gefunden werden können. Deshalb wurde das Konzept der Zusicherungen entwickelt. Dabei stellte sich die Frage,

```
    FUNCTION  BodyAssert : Boolean;
    { Modulglobale Zusicherungen                                           }
     BEGIN
       BodyAssert :=
           (assertEventually AND assertHenceforth AND
                  (sendMsg = lastSendMsg))
       AND
           (assertEventually AND assertHenceforth AND
                  (assertMainState = assertonlyReceiving))
       AND
           (assertHenceforth AND assertEventually AND
                  (sendCredit = 0))
       AND
           (sendCredit >= 0);
     END;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

     FUNCTION TransAssert : Boolean;
     { Zusicherungen, die erfuellt sind, wenn diese Trans. schaltet }
      BEGIN
        TransAssert :=
             (assertMainState = assertsendingAndReceiving)
        AND
             (sendMsg <> lastSendMsg)
        AND
             (sendCredit >= 1);
      END;
```

Bild 1: Die unterschiedlichen Funktionen zur Realisierung von Zusicherungen in ESTELLE

wie man Zusicherungen in ESTELLE–Spezifikationen unterbringen kann, ohne die Syntax von ESTELLE zu verändern. Eine Erweiterung des bisherigen Standards muß garantieren, daß die Spezifikationen weiterhin mit „konservativen“ ESTELLE–Werkzeugen bearbeitet werden können. In dieser Arbeit wird auf das in [The95] vorgeschlagene Konzept der Booleschen Funktionen zurückgegriffen, d.h. die Zusicherungen (oder *Assertions*) werden in Funktionen abgelegt, die immer den Wert `TRUE` zurückgeben, und die innerhalb der Spezifikation unter normalen Umständen nicht aufgerufen werden. Werden diese Funktionen dennoch einmal innerhalb der ESTELLE–Spezifikation benutzt, so verursacht dies keinen Schaden.

Man unterscheidet dabei zwischen Assertions, die global für alle Transitionen eines Moduls gelten, und anderen, die nur für eine bestimmte Transition gültig sind. In Bild 1 wird ein Beispiel für diese unterschiedlichen Zusicherungsfunktionen gegeben. Bild 2 gibt ein Beispiel dafür, wie eine nur für eine Transition gültige Zusicherung realisiert werden kann. Die Definition global gültiger Assertions geschieht analog im Kopf des jeweiligen Moduls.

In der Arbeit von Joachim Thees werden Zusicherungen benutzt, um eine statische Analyse der Schaltbarkeit von Transitionen bei Feature–Interaktionen zu erleichtern.

Es stellt sich die Frage, ob man das Konzept der Zusicherungen benutzen kann, um zur Laufzeit sicherzustellen, daß die Spezifikation nicht gegen die vorgegebenen Zusicherungen verstößt? Ist es also möglich, einen Weg zu finden, beim Schalten einer Transition zu überprüfen, ob bestimmte Eigenschaften bezüglich des Hauptzustands oder einiger Warteschlangen erfüllt sind? Dazu scheint es sinnvoll, die Zusicherungen zunächst auf weitere essentielle Bestandtteile von ESTELLE auszudehnen:

- Hauptzustände
- Interaktionspunkte und mit ihnen gekoppelte Warteschlangen

Bei diesen Konzepten von ESTELLE erscheint eine Einführung von Assertions sinnvoll, da dadurch das Verhalten der Spezifikation weiter verfeinert werden kann (d.h. Eigenschaften der Spezifikation, die bisher implizit im Kode versteckt waren, werden nun explizit in den Zusicherungen beschrieben). Falls es zu Widersprüchen zwischen Laufzeitverhalten und Zusicherungen kommt, ist es außerdem möglich, diese Widersprüche mit einem entsprechenden Werkzeug zur Laufzeit automatisch zu entdecken und protokollieren zu lassen.

Da nach erfolgreichen Tests der gesamte Zusicherungscode entfernt werden kann[2], kommt es durch die Verwendung von Zusicherungen zu keinerlei Einbußen im Laufzeitverhalten, wenn die endgültige Implementierung aus der Spezifikation generiert wird.

---

[2] Beispielsweise über eine Compiler–Option

```
{ Gib der Protokollmaschine nacheinander Uebertragungswuensche fuer }
{ die Nutzdaten, sofern Sendekredit fuer die lokale                 }
{ Flusskontrolle vorhanden ist.                                     }
TRANS
   FROM sendingAndReceiving
     TO SAME
       PROVIDED (sendMsg <> lastSendMsg) AND (sendCredit > 0)
         FUNCTION TransAssert : Boolean;
         { Zusicherungen die erfuellt sind, wenn diese Trans. schaltet }
          BEGIN
            TransAssert :=
                 (assertMainState = assertsendingAndReceiving)
            AND
                 (sendMsg <> lastSendMsg)
            AND
                 (sendCredit >= 1);
          END;
NAME userSending:
         BEGIN
           OUTPUT userToPm.T_dataReq(sendMsg);
           sendMsg := SUCC(sendMsg);
           sendCredit := sendCredit - 1
         END;
```

Bild 2: Beispiel zur Verwendung einer Transitions–Zusicherung

3 Erweiterungen der Zusicherungen in Estelle

3.1 Überblick über Estelle

Estelle ist ein Formale Beschreibungstechnik zum Entwurf von Kommunikationssystemen. Eine Estelle–Spezifikation beschreibt das Verhalten von (verteilten) hierarchisch-angeordneten Automaten. Diese Automaten kommunizieren über Nachrichten, die sie über bidirektionale *Kanäle* zwischen ihren *Interaktionspunkten* übertragen. Nachrichten werden in *Warteschlangen* solange zwischengespeichert, bis sie von dem entsprechenden Interaktionspunkt gelesen werden. Bei diesen Warteschlangen unterscheidet man zwischen *common queues*, Warteschlangen, die von allen Interaktionspunkten des Moduls gemeinsam benutzt werden, und *individual queues*, die nur

von einem Interaktionspunkt gelesen werden dürfen. Die *Transitionen* dieser Automaten werden in einer abgewandelten PASCAL–Syntax [ISO82] beschrieben. In ESTELLE ist es möglich, synchrone und asynchrone Parallelität zwischen den Automaten eines Systems zu modellieren.
Eine genaue Beschreibung von ESTELLE findet sich in [ISO89], ein grober Überblick in [Tur93].

Die für diese Arbeit relevanten Konzepte von ESTELLE sind

- **Hauptzustände.** Hauptzustände repräsentieren die Zustände des erweiterten endlichen Automaten, der mit Hilfe von ESTELLE beschrieben wird. Die möglichen Hauptzustände werden innerhalb einer ESTELLE–Spezifikation durch die `State`–Klausel definiert. Jeder Automat wird mit Hilfe des `Initialize`–Ausdrucks auf einen Anfangszustand gesetzt. Die Änderung des Zustands eines Automaten kann nur durch *Transitionen* geschehen, indem man `From ... To`–Klauseln verwendet. In Bild 3 findet sich ein Beispiel hierzu.

  Es gibt in ESTELLE–Spezifikationen, bis auf die `From ... To`– Klauseln, für denjenigen der die Spezifikation schreibt, keine Möglichkeit nachzuvollziehen, in welchem Hauptzustand sich das System gerade befindet. Gerade bei der Verwendung von *Statesets*, Mengen von Zuständen, die unter bestimmten Umständen gleiche Transitionsübergänge nach sich ziehen, wäre dies aber wünschenswert, um beispielsweise Aussagen über die Grenzwerte von Variablen in einem der möglichen Zustände machen zu können, die u.U. extrem von den Grenzwerten abweichen, die die Variable in einem der anderen Zustände dieses Statesets besitzt.

- **Interaktionspunkte** (*IP*), die als „Kupplungsstücke“ zwischen den Kanälen, auf denen die Nachrichten übertragen werden, und den Protokollinstanzen dienen. Über jeden dieser Interaktionspunkte kann auf eine Warteschlange zugegriffen werden, in der die übertragenen Nachrichten aufbewahrt werden. Bei diesen Queues unterscheidet man zwischen

  - **individual queues**, bei denen jedem Interaktionspunkt genau eine Schlange zugeordnet wird, in der nur Nachrichten für diesen IP stehen.
  - **common queues**, bei denen alle Interaktionspunkte auf die gleiche Queue zugreifen. D.h. Nachrichten, die in dieser Schlange gespeichert werden, sind nicht nur an einen bestimmten Interaktionspunkt addressiert. Das bedeutet, daß ein IP, der aus dieser

```
    STATE
      estab,
      waitAck;

    STATESET
      either = [estab,waitAck];

    INITIALIZE
      TO estab
NAME PM_init:
        BEGIN
          userSendCredit := creditLimit;
          sendSeqNum     := firstSeqNum;
          receiveSeqNum  := firstSeqNum;
        END;

     TRANS
       FROM estab
         TO waitAck
          WHEN pmToUser.T_dataReq(sdu)
NAME PM_sending:
            BEGIN
              msg.seqNum  := sendSeqNum;
              msg.pduId   := dataRI;
              msg.netData := sdu;
              OUTPUT pmToNet.N_dataReq(msg);
              userSendCredit := userSendCredit + 1;
            END;
```

Bild 3: Beispiel zur Definition und Änderung von Hauptzuständen in ESTELLE

Schlange lesen will, dies erst dann tun kann, wenn die für ihn bestimmte Nachricht an erster Stelle in der Queue befindet.

Definiert man *Module* innerhalb von ESTELLE, so wird dabei zwischen dem *Modulkopf* und dem *Modulrumpf*, oder auch *Body*, unterschieden. Der Modulkopf legt die syntaktische Schnittstelle des Moduls mit seiner Umgebung fest, während die Modulrümpfe das Verhalten eines Moduls beschreiben. Es ist erlaubt, zu einem Modulkopf mehrere verschiedene

```
MODULE protocolMachineType SYSTEMPROCESS;
  IP pmToUser: transmitService(provider) INDIVIDUAL QUEUE;
     pmToNet:  networkService(user) INDIVIDUAL QUEUE;
  END;
```

Bild 4: Beispiel für die Definition zweier Interaktionspunkte mit dem Attribut „Individual Queue“

> Modulrümpfe zu definieren. Die Interaktionspunkte werden innerhalb der Modulköpfe von ESTELLE definiert. Die einzige Ausnahme von dieser Regel stellen *interne Interaktionspunkte* dar, d.h. Interaktionspunkte über deren Kanäle Vater- und Sohnmodul miteinander verbunden werden. Ein Beispiel für die Definition von nicht internen Interaktionspunkten wird in Bild 4 gegeben. Die Definition von Common Queues erfolgt analog.

Alle in den nächsten Kapiteln vorgestellten Konzepte beziehen sich auf nichtgeschachtelte Module. Eine Erweiterung dieser Konzepte auf geschachtelte Strukturen findet in Kapitel 3.4 statt.

3.2 Überprüfung des Hauptzustands

Ziel dieser Arbeit war es, nach Möglichkeit keine inkompatible Erweiterung der ESTELLE–Syntax vorzunehmen. Wie bereits in [The95] erwähnt, baut das Konzept der Zusicherungsfunktionen darauf auf, daß Funktionen benutzt werden, die im normalen Programmverlauf entweder nicht aufgerufen werden[3] oder keine Wirkung haben, da sie immer `TRUE` zurückliefern. Deshalb ist es möglich, innerhalb dieser Funktionen bestimmte Schlüsselwörter zu verwenden, die innerhalb der Spezifikation nur dann eine Bedeutung haben, wenn ein bestimmtes Werkzeug verwendet wird[4]. Werden nun außerhalb der Zusicherungsfunktion diese Schlüsselwörter verwendet, so sollte das Analyse–Werkzeug eine Warnung ausgeben, daß dieses Schlüsselwort außerhalb einer Assertion benutzt wird. Benutzt man jedoch das Werkzeug nicht, so muß jeder ESTELLE–Compiler die angegebene Spezifikation akzeptieren[5].

---

[3] Innerhalb der ESTELLE–Spezifikation erfolgt kein Aufruf dieser Funktionen, ein Test–Werkzeug aber benutzt sie sehr wohl.

[4] Die Entwicklung dieses Werkzeugs ist nicht Teil dieser Arbeit

[5] Alle in dieser Arbeit vorgestellten Konzepte wurden mit Hilfe der entsprechenden Kapitel des ESTELLE–Standards auf ihre Kompatibilität überprüft.

```
   assertMainState : packed ARRAY [1..30] OF CHAR;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

   assertMainStateType : (sendingAndReceiving,
                          onlyReceiving);
   assertMainState : assertMainStateType;
```

Bild 5: Zwei Mögliche Repräsentationen des Hauptzustandes

Um auf den Hauptzustand zugreifen zu können, kann man innerhalb der Zusicherungsfunktionen Variablen oder Typen definieren, die einen Vergleich auf den Hauptzustand, innerhalb der erweiterten Semantik, ermöglichen. Zur Repräsentation des Hauptzustands kommen z.B. *Strings* und *Aufzählungstypen* in Frage. Die beiden Möglichkeiten werden in Bild 5 aufgezeigt.
Welches dieser beiden Konzepte ist nun besser geeignet?

3.2.1 Strings

Verwendet man Strings, so braucht man einer Variable dieses Typs bei der Initialisierung keinen sinnvollen Wert zuzuweisen, d.h. auch der leere String genügt. Anschließend können beliebige Vergleiche mit dieser Variablen durchgeführt werden. Diese Vergleiche müssen nach den Regeln der Standard–Estelle–Semantik nicht immer Sinn ergeben, d.h. man weist dieser speziellen String–Variablen den leeren String zu und vergleicht sie direkt danach mit einem „gefüllten“ String. Ein Beispiel für die Anwendung dieser Idee wird in Bild 6 gegeben. Hier werden String–Vergleiche durchgeführt, obwohl zu Beginn dieser Funktion der String `assertMainState` mit einem leeren String initialisiert wird. Die darauf folgenden Vergleiche ergeben innerhalb des Estelle–Codes keinerlei Sinn, bzw. ergeben immer `FALSE`. Innerhalb der erweiterten Semantik wird jedoch `assertMainState` durch den aktuellen Hauptzustand ersetzt. So erhält man eine Möglichkeit zum Vergleich von Hauptzuständen, um anschließend weitere Angaben über Grenzwerte von Variablen und über Beziehungen von Variablne untereinander machen zu können.
Obwohl das Konzept der Strings auf den ersten Blick sehr vielversprechend aussieht, offenbart es bei näherer Betrachtung einige Mängel:

- Die Abfragen, die mit Hilfe dieses Konzeptes durchgeführt werden, ufern, wie schon im Beispiel gesehen, in `IF ... THEN ... ELSE` Kaskaden aus, was nicht zur Lesbarkeit des Kodes beiträgt.

```
FUNCTION BodyAssert : Boolean;
BEGIN
  VAR
   assertMainState : packed ARRAY [1..30] OF Char;

  assertMainState := '';
  IF assertMainState = 'sendingAndReceiving'
     THEN BEGIN ... END
  ELSE IF assertMainState = 'onlyReceiving'
     THEN BEGIN ... END
             .
             .
             .
END;
```

Bild 6: Beispiel für die Verwendung von Strings

- Der String `assertMainState` sollte aus verständlichen Gründen nicht verändert werden dürfen. Das kann man innerhalb der Estelle–Spezifikation allerdings nicht verhindern. Es ist zwar möglich, jede Veränderung, die an diesem String durchgeführt wird, in dem erweiterten Semantik zu ignorieren, dies würde aber dazu führen, daß sich Standard-Semantik und erweiterte Semantik in gravierenden Dingen unterscheiden.

- Innerhalb der erweiterten Semantik ist nicht implizit klar, welchen Wertebereich die String–Variable besitzt. Die Folge ist, daß innerhalb des Werkzeugs diese Überprüfung ständig „von Hand" zu erfolgen hätte. Daraus folgt ein erhöhter Aufwand bei der Implementierung dieses Konzeptes.

3.2.2 Aufzählungstypen

Eine andere Möglichkeit, Informationen über den Hauptzustand einer Estelle–Spezifikation zu bekommen, besteht darin, alle auftretenden Hauptzustände in einem Aufzählungstyp zusammenzufassen, und Vergleiche dann mit Hilfe einer Variablen dieses Typs durchzuführen. Versucht man auf die Hauptzustände einer Estelle–Spezifikation über einen Aufzählungstyp zuzugreifen, so steht man vor allem vor der Schwierigkeit, daß Erweiterungen der Hauptzustände zwangsläufig auch eine Erweiterung der in den Zusicherungen angegebenen Hauptzustände nach sich ziehen müssen. Dabei kann es leicht passieren, daß eine Änderung nicht konsequent durchgeführt wird, oder es könnte – was vielleicht noch schlimmer ist – zu Schreibfehlern innerhalb der Aufzählung kommen. Werden nun die Hauptzustände überprüft, so kann man aus den oben genannten Gründen fehlerhafte Informationen erhalten, und eine sinnvolle Analyse der Spezifikation ist nicht mehr möglich. Da aber in dem Programmcode[6] , der aus der ESTELLE–Spezifikation erzeugt wird, und auf dem das Analyse–Werkzeug aufsetzt, die Aufzählung aller Hauptzustände vorliegt, ist es möglich, zu überprüfen, ob die bei den Zusicherungen angegebenen Hauptzustände vollständig und korrekt sind. Aus Gründen der Syntax von ESTELLE können nicht alle in dieser Arbeit vorgestellten Konzepte während des ersten Compilerdurchlaufs überprüft werden. Deshalb ist es erforderlich, daß die Überprüfung der Vollständigkeit der Aufzählungstypen erfolgt, nachdem der gesamte Spezifikationstext von dem Werkzeug gelesen wurde. Der daraus resultierende Zeitverlust beim Übersetzen der Spezifikation sollte aber nicht ins Gewicht fallen. Diese Tatsache gleicht den Nachteil des Eingebens der Hauptzustände aus, da man sicher sein kann, daß zu dem Zeitpunkt, an dem die Implementation gestartet wird, die Hauptzustände entweder korrekt vorliegen, oder aber es zu einer Fehlermeldung/Warnung des Analyse–Werkzeugs kommt.

Weitere Vorteile bei der Verwendung von Aufzählungstypen liegen darin, daß man `CASE`–Konstrukte zur Überprüfung verwenden kann, was die Lesbarkeit der Zusicherungs–Funktionen erhöht (siehe dazu auch Bild 7). Weiterhin steht der Geltungsbereich dieses Aufzählungstyps innerhalb der Semantik fest, d.h. der Entwickler eines Werkzeugs braucht sich nicht um die Einhaltung von Scoping–Grenzen zu kümmern. Ein Nachteil dieser Variante gegenüber dem String–Konzept besteht darin, daß man sowohl für den Typ, als auch für die Variable, die man mit diesem Typ benutzt, Namen reservieren muß. Es wird festgelegt, daß diese Namen in der erweiterten Semantik nur innerhalb der Zusicherungs–Funktionen reserviert sind, d.h. außerhalb dieser Funktionen sind sie frei und ohne Seiteneffekte verfügbar. Diese Einschränkung des ESTELLE–Standards ist in unseren Augen aber akzeptabel, da eine wirkliche Einschränkung des ESTELLE–Standards nur dann vorliegt, wenn die Konzepte der erweiterten Semantik benutzt werden. Verzichtet man auf diese Erweiterungen, so hat man den vollen Sprachumfang von ESTELLE zur Verfügung. Das oben genannte Problem der Zuweisung an die Hauptzustandsvariable wird auch mit der Verwendung eines Aufzählungs-

---

[6] Zwar generieren nicht alle Werkzeuge, die mit ESTELLE arbeiten, wirklichen Code einer Programmiersprache, aber der Einsatz des bereits angesprochenen Analyse–Tools, um Zusicherungen dynamisch, also zur Laufzeit, zu überprüfen, scheint nur dann sinnvoll zu sein, wenn dieses Wekzeug mit Code arbeiten kann, der aus der Spezifikation gewonnen wurde.

```
FUNCTION BodyAssert () : Boolean;
  TYPE
   assertMainStateType = (assertS1,asertS2,asertS3,asertS4);
  VAR
   assertMainState : assertMainStateType;

 BEGIN
  CASE (assertMainState) OF
    assertS1:
       BEGIN ... END;
    assertS2:
       BEGIN ... END;
  END;
 END;
    ...
```

Bild 7: Beispiel für die Verwendung eines Aufzählungstyps

typs nicht ganz aus der Welt geschafft, zumindest aber wird durch dieses Konzept verhindert, daß der Variable Werte zugewiesen werden, die nicht in der Aufzählung vorhanden sind. D.h. falsch geschriebene Hauptzustände werden durch die Verwendung eines Aufzählungstypen verhindert.
Ein weiteres Problem besteht darin, daß es zulässig ist, Spezifikationen zu schreiben, die keinen Hauptzustand besitzen. Dann würde es zu einem leeren Aufzählungstyp kommen, der aber laut ESTELLE–Standard nicht erlaubt ist. In diesem Fall macht es aber auch keinen Sinn, Zusicherungen über bestimmte Hauptzustände zu machen. Insofern braucht dieser Fall nicht betrachtet zu werden.

3.2.3 Definition der Erweiterten Semantik

Da beide Varianten ihre Vor- und Nachteile besitzen, ist es nicht leicht, zu sagen, bei welcher sich weniger Schwierigkeiten bei einer formalen Definition und ihrer Anwendung ergeben. Da aber die Aufzählungstypen in unseren Augen wesentlich bequemer zu handhaben sind – sowohl in der ESTELLE–Spezifikation, als auch bei der Semantik–Definition (s. hierzu auch Anhang B), fiel die Entscheidung zu Ihren Gunsten. Damit ergeben sich also die in Tabelle 1 angegebenen Spracherweiterungen[7] des ESTELLE–Standards.

---

[7] In diesem Zusammenhang kann man nur insofern von einer Spracherweiterung reden, als reservierte Worte eingeführt werden, die nur dann eine semantisch andere Bedeutung

Erweiterung	Semantik
`FUNCTION BodyAssert : Boolean;`	Zusicherungsfunktion, die Zusicherungen für das gesamte Modul angibt. Das Ergebnis, das von Ihr zurückgeliefert wird, ergibt sich aus der logischen Und–Verknüpfung der im Rumpf angegebenen Zusicherungen.
`FUNCTION TransAssert : Boolean;`	Zusicherungsfunktion, die Zusicherungen für eine Transition angibt. Das Ergebnis, das von ihr zurückgeliefert wird, ergibt sich aus der logischen Und–Verknüpfung der im Rumpf angegebenen Zusicherungen.
`assertMainStateType = (...);`	Aufzählungstyp, der die Hauptzustände des Moduls beinhaltet.
`assertMainState`	Variable vom Typ `assertMainStateType`. Obwohl sie niemals initialisiert wird, enthält sie zur Laufzeit den aktuellen Hauptzustand.

Tabelle 1: Erweiterung des ESTELLE–Standards, um Zugriff auf den Hauptzustand zu erhalten.

Daraus ergeben sich folgende Regeln der erweiterten kontextsensitiven Syntax und der erweiterten Semantik:

- Es muß am Ende des Übersetzungsvorgangs überprüft werden, ob die in `assertMainStateType` angegebenen Hauptzustände den Hauptzuständen des jeweiligen Moduls entsprechen. Ist dies nicht der Fall, muß eine Warnung ausgegeben werden, die den aufgetretenen Fehler beschreibt. Jede weitere Definition dieses Typs sollte ignoriert werden und die Ausgabe einer Warnung zur Folge haben. Wird dieser Typ außerhalb einer Zusicherungsfunktion angetroffen, so sollte eine Warnung ausgegeben werden, die den Benutzer darauf hinweist, daß er ein im Prinzip reserviertes Wort an einer Stelle gebraucht, an der die erweiterte Semantik nicht sinnvoll damit umgehen kann.

---

erhalten, wenn das angesprochene Analyse–Werkzeug benutzt wird, und diese Ausdrücke innerhalb der Zusicherungsfunktionen `BodyAssert` oder `TransAssert` benutzt werden. Tauchen diese Worte außerhalb der Assertions auf, so bleibt ihre semantische Bedeutung unverändert. Man erhält also eine vollwertige Erweiterung der bisherigen ESTELLE–Semantik, die aber kompatibel zur traditionellen Semantik bleibt.

- Auf jedes Auftreten der Variable `assertMainState` außerhalb von Zusicherungsfunktionen muß mit einer Warnung reagiert werden, die den Benutzer darauf hinweist, daß er ein im Prinzip reserviertes Wort an einer Stelle gebraucht, an der das Tool nicht sinnvoll damit umgehen kann.

- Jede Zuweisung an diese Variable innerhalb der Estelle– Spezifikation stellt einen Fehler dar, da der Hauptzustand einer Spezifikation nur mit Hilfe von `FROM...TO`–Klauseln geändert werden darf. Deshalb muß an dieser Stelle ebenfalls eine Warnung mit entsprechendem Inhalt an den Benutzer ausgegeben werden. Bei Auftreten der Variable `assertMainState` innerhalb von Zusicherungen muß innerhalb des erzeugten Codes diese Variable durch den jeweiligen Hauptzustand der Spezifikation ersetzt werden. Da dieser z.B. innerhalb von PET seinerseits durch eine Variable repräsentiert wird, stellt dies kein großes Problem dar.

- Aus Gründen, auf die erst im nächsten Kapitel eingegangen werden kann, ist es erforderlich, dem Namen jedes Hauptzustands das Literal `assert` voranzustellen. Dies ist bei Hauptzuständen zwar nicht unbedingt notwendig, aus Konsistenzgründen aber empfehlenswert.

3.3 Interaktionspunkte und Warteschlangen

Wie bereits erwähnt, dienen die Interaktionspunkte in Estelle dazu, die einzelnen Module mit den entsprechenden Kanälen zu koppeln. Die auf diesen Kanälen übertragenen Nachrichten werden in Warteschlangen, den sogenannten *queues*, gespeichert, bis sie entgegengenommen werden. Nachrichten werden aus diesen Warteschlangen mithilfe der `When`–Klauseln entfernt. Diese bewirkt, daß die Nachricht aus der Schlange gelöscht wird, und man anschließend innerhalb der Spezifikation auf diese Nachricht bzw. ihre Parameter zugreifen kann. In Bild 8 wird dieses Verfahren verdeutlicht.

Innerhalb einer Estelle–Spezifikation besteht keine Möglichkeit, auf die Nachrichten zuzugreifen, die gerade an den entsprechenden Interaktionspunkten anliegen, ohne die Nachricht gleichzeitig aus der Warteschlange zu entfernen. Trotzdem kann es, gerade in Zusicherungen, sinnvoll sein, etwas über die Länge der momentanen Warteschlange zu wissen, oder über die Art der Nachricht, die am Anfang der Schlange steht. Beispielsweise ist es bei der Verwendung einer *common queue* durchaus sinnvoll zuzusichern, daß niemals mehr als drei Nachrichten gleichzeitig in der Schlange vorhanden sind. Geht es dann später darum, die Spezifikation mit Hilfe eines Werkzeugs in eine Im-

```
     TRANS
       FROM estab
         TO waitAck
           WHEN pmToUser.T_dataReq(sdu)
           (* Wenn am Interaktionspunkt eine Nachricht vom Typ
              T_dataReq anliegt, so nimm sie aus der Warteschlange
              und lege ihre Parameter in der Variablen sdu ab.
              Nur dann fahre mit der Bearbeitung fort.
              Sonst suche nach einer anderen feuerbaren Transition.
           *)
NAME PM_sending:
            BEGIN
              msg.seqNum  := sendSeqNum;
              msg.pduId   := dataRI;
              msg.netData := sdu;
              OUTPUT pmToNet.N_dataReq(msg);
              userSendCredit := userSendCredit + 1;
            END;
```

Bild 8: Beispiel zur Verwendung der `When`–Klausel

plementation umzusetzen, so kann der Puffer dieser Queue passend gewählt werden, so daß keine Resourcen verschwendet werden.

3.3.1 Informationen über Interaktionspunkte

Um an die gewünschten Informationen der jeweiligen Interaktionspunkte zu kommen, werden weitere reservierte Wörter[8] eingeführt.

Variablendefinition

Um überhaupt auf Interaktionspunkte zugreifen zu können, werden zwei weitere Aufzählungstypen eingeführt.
Der erste enthält die Namen aller Interaktionpunkte sowie zusätzlich das Konstrukt `assertAllIPs`. Dieser Aufzählungstyp soll es möglich machen, auf die Namen der Interaktionspunkte zugreifen zu können. Das Literal `assertAllIPs` dient als abkürzende Schreibweise für Zusicherungen, die modulweit

---

[8] Zur Erinnerung: Diese Wörter haben nur in den Zusicherungen eine spezielle Bedeutung. Trotzdem wird versucht, die Anzahl der reservierten Ausdrücke möglichst klein zu halten.

für alle Interaktionspunkte gelten sollen.
Der andere enthält alle möglichen auftretenden Nachrichtentypen. Dieser Typ wird später benötigt, um Aussagen über die in der Warteschlange an erster Stelle stehende Nachricht machen zu können.
Die Informationen, die über einen Interaktionspunkt zur Verfügung stehen sollen, sollen in einem `RECORD` abgespeichert werden. Dabei sind folgende Informationen von besonderem Interesse:

- Befindet sich an erster Stelle innerhalb der Queue, an der dieser IP angeschlossen ist, eine Nachricht, die an den IP adressiert ist?
  Diese Information wird von der Booleschen Variablen `IsMsgInQueue` zur Verfügung gestellt.
- Wieviele Nachrichten stehen in der Queue, unabhängig davon, ob sie an diesen bestimmten IP adressiert sind?
  Die Anzahl der Nachrichten wird mit Hilfe der Variable `MsgCount` gezählt.
- Welchen Typ hat die erste Nachricht, die in der Queue steht?
- Welche Werte haben die Parameter der ersten Nachricht in der Queue, so sie parametrisiert ist?

Da die gewünschten Informationen, bis auf den Namen des Interaktionspunktes, erst zur Laufzeit verfügbar sein werden, muß ein Mechanismus gefunden werden, der die Informationen in das entsprechende Konstrukt einträgt. Dies könnte geschehen, sobald der Name des Interaktionspunktes in den `RECORD` eingetragen wird. D.h. sobald in dem entsprechenden `RECORD` das Feld des IP–Namens über eine explizite Zuweisung gefüllt wird, füllt das Werkzeug automatisch alle anderen Felder aus, ohne daß für diese Aktionen Anweisungen im Spezifikationstext auftauchen. Dies scheint aber eine unsaubere Lösung zu sein, da dann der Spezifikationstext nicht mit den tatsächlich ausgeführten Anweisungen übereinstimmen würde. Diese Variante dürfte eine Spezifikation also eher „verwirrender“ als „lesbarer“ machen.

Die Funktion assertFillIPRecord

Deshalb wird zunächst[9] eine systemglobale Funktion eingeführt, die als Parameter den Namen des Interaktionspunktes übergeben bekommt, und einen

---

[9] Das Konzept dieser Funktion wird in Kapitel 3.3.2 noch insofern geringfügig verändert, daß statt einer normalen Funktion eine primitive Funktion benutzt wird. An der erweiterten Semantik der Funktion `assertFillIPRecord` ändert sich jedoch nichts.

```
TYPE
   assertIPNamesType =
             (assertpmTuUser,    { In diesem Aufzaehlungstyp sind }
              assertpmToNet,     { alle in diesem Modul vorkom-   }
              assertAllIPs);     { menden IPs enthalten.          }
   assertMsgType     =
             (assertT_dataReq,   { Hier werden alle in der Spezi- }
              assertT_dataInd,   { fikation vorkommenden Nach-    }
              assertT_creditInd, { richtentypen abgelegt.         }
              assertN_dataReq,
              assertN_dataInd);
   assertIPInfoType  =
              RECORD
                IPName :         assertIPNamesType;
                                 { Name des IP                    }
                IsMsgInQueue :   Boolean;
                                 { Befindet sich eine Nachricht   }
                                 { fuer den entsprechenden IP in  }
                                 { der Queue?                     }
                MsgCount :       Integer;
                                 { Wieviele Nachrichten stehen in }
                                 { der Queue?                     }
                CASE TypeOfFirstMsg : assertMsgType OF
                                 { Typ der ersten Nachricht in    }
                                 { der Queue.                     }
                  T_dataReq,                      { Hier folgen   }
                  T_dataInd:   (sdu: sduType);    { die moeglichen}
                  T_creditInd: (credit: Integer); { Parameter     }
                  N_dataReq,
                  N_dataInd:   (pdu: pduType);

                END;
             END;
VAR
   assertIPInfo : assertIPInfoType;
```

Bild 9: Variablendefinitionen für Zusicherungen über Interaktionspunkte

`RECORD` zurückgibt, in dem die entsprechenden Informationen enthalten sind. Da die Daten beim Erstellen der Spezifikation nicht vorliegen, werden alle

Felder des `RECORDS` innerhalb der Estelle–Spezifikation auf neutrale Werte gesetzt. D.h. auf beliebige Werte, die schon bei der Erstellung der Spezifikation vorliegen. Diese Zuweisungen dienen nur als Platzhalter für die Zuweisungen, die durch das Analyse–Werkzeug zur Laufzeit erfolgen sollen. Erst unter Zuhilfenahme der erweiteren Semantik bekommt man die korrekten Werte[10]. In Bild 9 wird verdeutlicht, wie die Definition der entsprechenden Variablen und Typen aussieht, die EBNF findet sich in Anhang B. Bei der Definition des `RECORDS` muß beachtet werden, daß alle Komponentennamen nur einaml verwendet werden dürfen. deshalb werden alle Nachrichtentypen, die dieselbe Komponente als Parameter beinhalten durch Kommata getrennt aufgeführt. Im Rumpf der Funktion wird der Name des Interaktionspunktes in das entsprechende Feld eingetragen und alle anderen Felder auf neutrale Werte (s.o.) gesetzt. Dies geschieht, um eine Spezifikation zu erzeugen, die der Standard–Estelle–Syntax entspricht. Wird das Analyse–Tool verwendet, so werden von diesem Werkzeug, an Stelle der angegebenen Werte, die für den jeweiligen Interaktionspunkt gültigen Werte in die Felder eingetragen. In Bild 10

```
     FUNCTION assertFillIPRecord (assertRecordName: assertIPNamesType):
              assertIPInfoType;
VAR assertIPRecord: assertIPInfoType;

   BEGIN
     assertIPRecord.
         IPName := assertRecordName;        { Die hier eingetragenen }
     assertIPRecord.                        { Werte haben semantisch }
         IsMsgInQueue := FALSE;             { keine Auswirkung. Sie  }
     assertIPRecord.                        { werden zur Laufzeit    }
         MsgCount := 0;                     { durch das Analyse-Tool }
     assertIPRecord.                        { durch die realen Werte }
         TypeOfFirstMsg := assertT_dataReq; { ersetzt.               }
     assertIPRecord.
         sdu := '@';
     assertFillIPRecord := assertIPRecord;
END;
```

Bild 10: Aufbau der Funktion `assertFillIPRecord`

wird der Aufbau dieser Funktion angegeben.

---

[10] Diese Syntax ist weitaus besser lesbar als die zuvor behandelte.

Nachdem es nun möglich ist, Informationen über Interaktionspunkte zu erhalten, muß nun noch eine Möglichkeit gefunden werden, Zusicherungen über diese Interaktionspunkte zu machen.

3.3.2 Zusicherungen über Interaktionspunkte

Will man die Funktion `assertFillIPRecord` innerhalb einer der beiden Assertions verwenden, so kommt es zu Problemen mit dem Scoping innerhalb von ESTELLE–Spezifikationen. ESTELLE überschreibt bereits definierte Namen mit den jeweils modulweit gültigen. Dies geschieht auch, wenn die bereits vergebenen Identifier zu einem völlig anderen Typ gehören. Das führt dazu, daß beispielsweise spezifikationsglobale Aufzählungstypen, die die Namen von Interaktionspunkten enthalten, innerhalb eines Moduls mit den jeweiligen Interaktionspunkten überschrieben werden. So kommt es dazu, daß innerhalb der ESTELLE–Semantik beim Aufruf der Funktion `assertFillIPRecord` der übergebenen Parameter als IP identifiziert wird und nicht als Teil eines Aufzählungstyps gilt. Deshalb müssen, was die Deklaration von Typen angeht, folgende Vereinbarungen getroffen werden:

- Die für die Zusicherungen benötigten Typ–Definitionen müssen außerhalb der eigentlichen `Body`–Spezifikation erfolgen, da sonst andere Module nicht auf diese globalen Typen zugreifen können. Daraus ergibt sich bis jetzt die in Bild 3.3.2 wiedergegebene Erweiterung der bisherigen ESTELLE–EBNF. Dabei gelten die im ESTELLE–Standard festgelegten Constraints bezüglich der Benutzung von Identifiern.
- Alle erforderlichen Variablen–Deklarationen müssen in jedem Modul lokal durchgeführt werden.
- Jedes Element der Aufzählungstypen muß mit dem Präfix `assert` versehen werden, um das oben beschriebene Überschreiben der Namen zu verhindern. Aus Konsistenzgründen sollte dieser Präfix den Elementen aller Aufzählungstypen vorangestellt werden. Bis auf die zusätzliche Schreibarbeit bringt dieses Vorgehen keine weiteren Nachteile mit sich.

Bereits zu Beginn dieses Kapitels wurde gesagt, daß die Funktion `assertFillIPRecord` eine modulglobale Funktion ist, d.h. sie muß in jedem Modul, in dem die entsprechenden Zusicherungen benutzt werden sollen, erneut definiert werden. Bei Spezifikationen mit einer großen Anzahl von Modulen ist dies nicht nur ausgesprochen lästig, es bläht auch den Spezifikationstext unnötig auf.
Daher ist es in unseren Augen besser, auf die Verwendung von *primitive functions* zurückzugreifen.

```
declarations-part = {declarations}.

declarations = CONSTNAT-DEFINITION-PART
               | TYPE-DEFINITION-PART
               | assert-type-definitions
               | channel-definition
               | module-header-definition
               | module-body-definition
               | interaction-point-definition
               | module-variable-declaration-part
               | VARIABLE-DECLARATION-PART
               | state-definition-part
               | assert-function-declaration-part
               | PROCEDURE-AND-FUNCTION-DECLARATION-PART.

assert-type-definitions = assertIPNamesType
                          | assertMsgType
                          | assertIPInfoType.
```

Bild 11: Erweiterung der Estelle–EBNF

primitive functions

```
   FUNCTION assertFillIPRecord (assertRecordName: assertIPNamesType):
            assertIPInfoType;
    primitive;
```

Bild 12: Beispiel zur Deklaration einer primitive-function

Bei diesen Funktionen handelt es sich um Konstrukte von Estelle, bei denen zwar der Funktionskopf angegeben wird, der Rumpf der Funktion aber nicht in der Spezifikation erfolgt. Auch die Notation der Funktionssemantik ist von Estelle unabhängig. In Bild 12 ist ein Beispiel zur Deklaration dieser Funktionen angegeben. Der Vorteil bei der Verwendung von primitive Funktions besteht darin, daß diese Funktionen an alle Sohn–Module weiter vererbt werden. D.h. man spart den Schreibaufwand, der bei einer herkömmlichen Spezifikation anfallen würde. Ein Nachteil besteht jedoch darin, daß, wie bereits gesagt, kein Funktionsrumpf angegeben werden kann, die kontextabhängige Semantik also vollkommen verborgen bleibt. Dies läßt sich aber dadurch vermeiden, daß der Rumpf der Funktion als Kommentar an den Funktionsrumpf angehängt wird. Dabei muß allerdings beachtet werden, daß ESTELLE keine geschachtelten Kommentare versteht. Entweder man läßt also Kommentare innerhalb des Funktionsrumpfes weg, oder man verwendet die beiden unterschiedlichen Arten der Kommentierung: `(* *)` und {}.

Mit Hilfe dieser in Bild 13 gezeigten Methode steht nun der Rumpf der

```
      FUNCTION assertFillIPRecord (assertRecordName: assertIPNamesType):
               assertIPInfoType;
               primitive;
(*
 VAR assertIPRecord: assertIPInfoType;

    BEGIN
      assertIPRecord.
          IPName := assertRecordName;        { Die hier eingetragenen }
      assertIPRecord.                        { Werte haben semantisch }
          IsMsgInQueue := FALSE;             { keine Auswirkung. Sie  }
      assertIPRecord.                        { werden zur Laufzeit    }
          MsgCount := 0;                     { durch das Analyse-Tool }
      assertIPRecord.                        { durch die realen Werte }
          TypeOfFirstMsg := assertT_dataReq; { ersetzt.               }
      assertIPRecord.
          sdu := '@';
      assertFillIPRecord := assertIPRecord;
         END;
*)
```

Bild 13: Die Funktion `assertFillIPRecord` als primitive Funktion.

Funktion in der Spezifikation, die Funktion wird an alle Sohn–Module vererbt, und bei einer Verarbeitung durch ein Analyse–Werkzeug braucht bloß der auf die Schlüsselwörter „assertFillIPRecord“ und „primitive“ reagiert zu werden, um an die erweiterte Semantik zu gelangen. Die Angabe des oben beschriebenen Kommentars ist also nicht zwingend erforderlich, dient aber der besseren lesbarkeit der Spezifikation und wird deshalb als Stil–Regel gefordert. In unseren Augen ist dies die beste Lösung, da sie sowohl minimalen (Schreib)–Aufwand und maximale Wirkung vereint, und wird deshalb auch bei weiteren in dieser Arbeit auftauchenden Funktionen verwendet.

3.3.3 individual– und common–queues

Wie bereits in Kapitel 3.1 angesprochen, wird in Estelle zwischen *individual* und *common queues* unterschieden. Beim Zugriff auf die erste Nachricht, die an einem Interaktionspunkt anliegt, entstehen durch diese Unterscheidung allerdings keine Probleme.
Liegt bei einer individual queue keine Nachricht an, so ist der Wert von `assertIPRecord.isMsginQueue` `FALSE` und der Wert von `assertIPRecord.-MsgCount` gleich 0. Sonst zeigt die Boolesche Variable `TRUE` an, und in `assertIPRecord.MsgCount` befindet sich die Anzahl der in der Warteschlange stehenden Nachrichten. Bei einer leeren common queue würde das Ergebnis genauso aussehen, wie bei einer leeren individual queue. Sind jedoch in einer common queue Nachrichten vorhanden, die erste Nachricht jedoch nicht an den entsprechenden IP adressiert, so kommt es zu folgendem Verhalten:
Der Wert von `assertIPRecord.IsMsginQueue` ist `FALSE`, der Wert von `assertIPRecord.MsgCount` zeigt hingegen die Anzahl der Nachrichten an, die sich momentan in der Queue befinden. Mit diesem Verhalten lassen sich die Zustände von Warteschlangen, die für Zusicherungen interssant sind, eindeutig identifizieren.

3.4 Hierarchische Schachtelung

Wie bereits in Kapitel 3.3.2 bei den Interaktionspunkten beschrieben, werden in Estelle bereits definierte Identifier von einer nochmaligen Deklaration überschrieben. Dies führt zu Problemen, wenn man die bisher beschriebenen Konzepte auf geschachtelte Module ausdehnen will.
Ohne die Konzepte zu verändern, besteht eine Möglichkeit der Ausdehnung darin, bei der Definition der neu eingeführten Aufzählungstypen alle Identifier zu berücksichtigen, über die Aussagen gemacht werden sollen. Dabei ist es nicht wichtig, ob diese innerhalb eines Vater– oder eines Sohn–Moduls verwendet werden. Dieses Vorgehen führt dazu, daß redundante Information „mitgeschleppt“ werden muß. Dafür ist dieses Vorgehen leicht zu implementieren. Die bisher vorgestellten Konzepte lassen sich also auf geschachtelte Estelle–Strukturen ausdehnen, solange die Instanzen der einzelnen Module eindeutig sind. Da es aber oft vorkommen wird, daß für einen Modul–Kopf mehrere unterschiedliche Body–Definitionen angegeben werden, sind die bisher vorgestellten Lösungen für geschachtelte Spezifikationen nicht brauchbar. Um nun weiterhin Informationen über einen Interaktionspunkt in einem bestimmten Sohn–Modul erhalten zu können, braucht man als zusätzliche Identifizierungshilfe die Modulvariable, die auf diesen Sohn zeigt. Deshalb wir ein weiterer reservierter Aufzählungstyp eingeführt, der alle Modulvaria-

blen der Spezifikation enthält. Zusätzlich dazu wird das Literal `assertSelf` eingeführt, das benutzt werden soll, wenn auf das aktuelle Modul zugegriffen wird. Daraus ergibt sich dann die Aufzählung, wie sie in Bild 14 gezeigt wird. Um bisher Informationen über die Interaktionspunkte zu erhalten, wurde die Funktion `assertFillIPRecord` benutzt, die als Parameter den Namen des jeweiligen Informationspunktes übergeben bekam (s. Kapitel 3.3.1). Nun wird der Kopf dieser Funktion um einen weiteren Parameter, nämlich die Modulvariable erweitert, und in der kontextabhängigen Semantik werden die Informationen über den Interaktionspunkt eines beliebigen Moduls zugänglich. Am Modulrumpf selber braucht dabei nichts geändert zu werden. In Bild 15 wird die - nun endgültige - Form der Funktion `assertFillIPRecord` gezeigt. Damit das in diesem Kapitel vorgestellte Konzept aber überhaupt angewendet werden kann, muß zusätzlich gefordert werden, daß auf jede Modulinstanz genau eine Modulvariable zeigt, da sonst wieder Mehrdeutigkeiten entstehen, die innerhalb der erweiterten Semantik nicht aufgelöst werden können. Daraus ergibt sich die in Bild 3.4 festgelegte Erweiterung der ESTELLE–EBNF.

```
TYPE
   assertModVarType = (assertUserA, assertUserB, assertPM_A,
                       assertPM_B, assertNetwork, assertSelf);
```

Bild 14: Der Aufzählungstyp assertModVarType

3.5 Semantik der Zusicherungen

In Kapitel 2 wurde bereits erläutert, daß Zusicherungen in ESTELLE bisher nur statisch, d.h. vor der Laufzeit der eigentlichen Implementierung überprüft wurden. Das geschah, um Feature–Interaktionen zwischen den verschiedenen Komponenten eines Intelligent Networks auf die Spur zu kommen.

Mit der Einführung von Zusicherungen, die auch zur Laufzeit überprüft werden sollen, gewinnt das Konzept zwar einiges an Qualität, es treten aber auch Probleme auf, mit denen man sich bisher nicht auseinanderzusetzen hatte. Als Qualitätsgewinn läßt sich zum Beispiel die Abfrage von Hauptzuständen und Anzahl der Nachrichtenschalange an einem IP verbuchen, dafür mußten aber zahlreiche neue reservierte Wörter eingefügt werden, so daß der Sprachumfang von ESTELLE eingeschränkt wird.

- In Zusicherungsfunktionen werden die gemachten Zusicherungen logisch miteinander verknüpft, und die Funktion erhält als Ergebnis dann

```
      FUNCTION assertFillIPRecord (assertModuleName: assertModVarType,
                                   assertRecordName: assertIPNamesType):
               assertIPInfoType;
               primitive;
(*
 VAR assertIPRecord: assertIPInfoType;

    BEGIN
      assertIPRecord.
          IPName := assertRecordName;        { Die hier eingetragenen }
      assertIPRecord.                        { Werte haben semantisch }
          IsMsgInQueue := FALSE;             { keine Auswirkung. Sie  }
      assertIPRecord.                        { werden zur Laufzeit    }
          MsgCount := 0;                     { durch das Analyse-Tool }
      assertIPRecord.                        { durch die realen Werte }
          TypeOfFirstMsg := assertT_dataReq; { ersetzt.               }
      assertIPRecord.
          sdu := '@';
      assertFillIPRecord := assertIPRecord;
         END;
*)
```

Bild 15: Die Funktion `assertFillIPRecord` mit der Möglichkeit auf Interaktionspunkte anderer Module zugreifen zu können.

das Ergebnis dieser Verknüpfung. Da diese Funktionen zur Laufzeit nicht ausgeführt werden, ist das Ergebnis zur Laufzeit uninteressant. Wenn nun aber die `assert`–Funktionen doch zur Laufzeit aufgerufen werden, kommt es dazu, daß entweder `TRUE` oder `FALSE` zurückgeliefert wird, je nachdem, welchen Wert die Assertions gerade haben. Das ließe sich dadurch vermeiden, daß nach der Zuweisung der Zusicherungen eine erneute Zuweisung mit `TRUE` durchgeführt wird. Dieses Vorgehen führt unserer Meinung nach aber eher zur Verwirrung, als daß es zur weiteren Klärung der Spezifikation beitrüge. Um dieses Verhalten sauber zu unterbinden sollte ein Analyse–Werkzeug, das mit Zusicherungen arbeitet, den Benutzer auf die Probleme aufmerksam machen, die ein expliziter Aufruf von Zusicherungsfunktionen mit sich bringt.

```
declarations-part = {declarations}.

declarations = CONSTNAT-DEFINITION-PART
             | TYPE-DEFINITION-PART
             | assert-type-definitions
             | channel-definition
             | module-header-definition
             | module-body-definition
             | interaction-point-definition
             | module-variable-declaration-part
             | VARIABLE-DECLARATION-PART
             | state-definition-part
             | assert-function-declaration-part
             | PROCEDURE-AND-FUNCTION-DECLARATION-PART.

assert-type-definitions = assertIPNamesType
                        | assertMsgType
                        | assertIPInfoType
                        | assertModVarType.
```

Bild 16: Endgültige Form der Erweiterung der ESTELLE–EBNF

- Eine andere Frage ist, wann die Zusicherungen zur Laufzeit überprüft werden müssen.
  Der Transitionsauswahlprozess ist in 2 Phasen unterteilt. In der ersten Phase werden alle schaltbereiten Transitionen ausgewählt und in der Menge $AS(gid_p)$ abgelegt. In der zweiten Phase erfolgt das atomare Schalten einer ausgewählten Transition dieser Menge. Dieser Vorgang wird solange wiederholt, bis die Menge $AS(gid_p)$ leer ist. Dann beginnt erneut Phase 1. Die Menge $AS(gid_p)$ wird in ESTELLE nach folgenden Regeln gebildet:
  - Hat ein Vatermodul eine oder mehrere schaltbereite Transitionen, so besteht $AS(gid_p)$ nur aus diesen Transitionen.
  - Hat das Vatermodul keine schaltbaren Transitionen, und ist das Vatermodul *activity* oder *systemactivity* attributiert so besteht $AS(gid_p)$ aus einer nichtleeren Menge schaltbereiter Transitionen eines der Söhne. Auf welche Menge dabei zurückgegriffen wird, wird nichtdeterministisch ermittelt.

- Besitzt ein Vatermodul keine schaltbereite Transition, und ist es *process* oder *systemprocess* attributiert, so wird $AS(gid_p)$ aus der Vereinigung der Mengen der schaltbaren Transitionen der Söhnen gebildet.

Anschließend wird aus der Menge $AS(gid_p)$ nichtdeterministisch solange eine Transition schalten, wie die Menge $AS(gid_p)$ nicht leer ist. Anhand dieser Vorgaben erscheint die folgende Reihenfolge bei der Überprüfung von Zusicherungen sinnvoll:

- lokale Transitionen werden ausgeführt, unmittelbar bevor die ausgewählte Transition schaltet.
- modulglobale Transitionen werden überprüft, unmittelbar nachdem eine Transition geschaltet hat. Durch dieses Vorgehen wird sichergestellt, daß innerhalb einer Spezifikation, die nur aus der `initialize`–Transition besteht, die modulglobalen Zusicherungen wenigstens einmal - nach dem Schalten der `initialize`–Transition - überprüft werden. Laut [ISO89, 9.5] wird die `initialize`–Transition wie eine normale Transition behandelt.

3.6 Zusicherungen über die Variablen anderer Module

Bisher wurden nur Zusicherungen gemacht, die entweder Module direkt betrafen oder Aussagen über Sohnmodule machten. Analog zum Konzepts der exportierten Variablen von ESTELLE wäre es auch möglich, Aussagen über Variablen von Modulen zu machen, die nicht zu dem Subsystem gehören, in dem die Zusicherungen gemacht wurden. Dazu müßte man innerhalb der erweiterten Semantik festlegen, daß innerhalb von Zusicherungsfunktionen alle Variablen der anderen Module als exportiert betrachtet werden. Dann wäre ein Zugriff der Art `Modulname.Variablenname` möglich. Durch diese Semantik definiert man einen zweiten Geltungsbereich, der sich gravierend von dem der konservativen ESTELLE–Semantik unterscheidet. Da es aber durchaus sinnvoll sein kann, Zusicherungen über das Verhalten anderer Modulinstanzen zu machen, soll an dieser Stelle dieses Konzept nicht gänzlich verworfen werden. Aufgrund der in unseren Augen aber erheblichen Unsauberkeit dieses Konzepts wird auf dieses Konzept innerhalb dieser Arbeit nicht zurückgegriffen.

3.7 Aufwandsabschätzung

3.7.1 Aufwand der Implementierung

Ein Werkzeug, das die hier angegebenen Konzepte (sowohl statisch als auch dynamisch) verwirklicht und nutzbar macht, soll mit dem arbeiten können, was von einem ESTELLE–Compiler erzeugt wird. Da z.B. DINGO für jedes Modul ein Objekt erzeugt, das die Informationen enthält, die in dieser Arbeit benutzt werden, ist der Aufwand, der benötigt wird, um Zusicherungen zu ermöglichen, die bereits vor der Laufzeit der Spezifikation überprüft werden können, als gering einzuschätzen. Es muß lediglich eine Möglichkeit gefunden werden, bei der Deklaration der Aufzählungstypen sicherzustellen, daß alle korrekt und vollständig angegeben wurden.

3.7.2 Laufzeitverhalten

Läuft eine ESTELLE–Implementierung, und werden die darin vorhandenen Zusicherungen überprüft, so kommt es mit Sicherheit zu Änderungen im Laufzeitverhalten dieser Anwendung gegenüber einer Implementierung ohne Zusicherungen.
Da die hier vorgestellten Konzepte aber nur theoretisch untersucht wurden, kann keine gesicherte Aussage darüber erfolgen, wie sehr das Laufzeitverhalten beeinträchtigt wird. Mit großer Sicherheit kann aber gesagt werden, daß sich das Laufzeitverhalten bei der Verwendung von globalen Zusicherungen um einen Faktor vergrößert, der der zusätzlichen Verwendung einer komplexen `Provided`–Klausel entspricht. Eine gleichartige Verzögerung dürfte sich durch die Verwendung lokaler Zusicherungen ergeben.
Legt man die in Kapitel 3.5 gemachten Aussagen über die Überprüfungshäufigkeit der Zusicherungsfunktionen zugrunde, so erkennt man, daß aufgrund einer beliebigen Schachtelungstiefe der ESTELLE–Moduln keine gesicherte Aussage über die Anzahl der auszuwertenden Zusicherungsfunktionen gemacht werden können. Deshalb ist die einzige Aussage, die gemacht werden kann, diejenige, daß sich bei der Verwendung von Zusicherungen die Ausführung von ESTELLE–Spezifikationen um einen spürbaren Faktor verlangsamen wird.
Da aber, besonders bei zeitkritischen Anwendungen, alle Assertions nach der Testphase wieder entfernt werden können, ist nicht zu befürchten, daß ESTELLE als Spezifikationssprache durch diesen Performance–Verlust an Bedeutung verlieren wird.

4 Temporallogik in Zusicherungen

Bereits in Griechenland diskutierten Stoiker, Megarier und auch Aristoteles über die Zeit, indem sie Annahmen über Gegenwart, Vergangenheit und Zukunft machten. Diese Arbeiten gerieten in Vergessenheit, bis im Mittelalter Thomas von Aquin und einige arabische Mathematiker sich wieder mit diesem Thema beschäftigten. Seit den vierziger Jahren dieses Jahrhunderts begann dann das Interesse an temporaler Logik wieder zuzunehmen. Seit den ersten Arbeiten über Temporallogik von Amir Pnueli [Pnu77] ist dieser Zweig der formalen Logiken von besonderem Interesse für die Bereiche der Informatik, die sich mit nebenläufigen Systemen beschäftigen. Heute ist die Temporallogik ein bekannter Formalismus, der seine Verwendung größtenteils beim Entwurf offener, verteilter Systeme findet. Eine ausführliche Einführung, aus der auch diese Einleitung stammt, findet sich in [Got92].

4.1 Temporallogische Operatoren

Temporale Logik basiert auf Aussagenlogik. Betrachtet man Aussagenlogik von einem zeitlichen Standpunkt aus, so können mit dieser nur Aussagen über die Gegenwart gemacht werden.

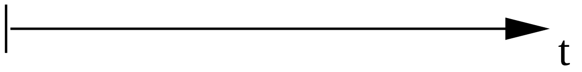
Um nun auch Aussagen über die Zukunft zu ermöglichen, führt die Temporallogik zwei weitere Operatoren ein. Sei im folgenden p ein Ausdruck. Dann gilt:

- $\Box\ p$ (gelesen „für immer p“) bedeutet, daß p jetzt gilt, und zu jedem beliebigen Zeitpunkt in der Zukunft.
- $\Diamond\ p$ (gelesen „schließlich p“) bedeutet, daß p jetzt oder zu einem beliebigen Zeitpunkt in der Zukunft gilt.

Mit Hilfe dieser Logik kann man natürlich Aussagen über all das machen, was auch mit Aussagenlogik möglich ist.

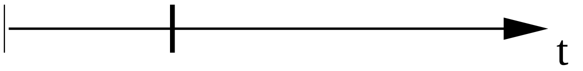
Die folgenden Beispiele wurden [Got92] entnommen.

Die Erde umkreist die Sonne.

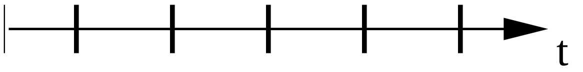


Mit dem „für immer“–Operator kann man Aussagen machen, die zeitlich invariant sind, d.h. sie gelten ab dem Zeitpunkt, zu dem die Aussage gemacht wird, für immer. Damit kommt eine zeitliche Komponente mit hinzu, die mit Aussagenlogik allein nicht ausdrückbar gewesen wäre.
□ (Der Mond umkreist die Erde.)

Durch den „schließlich“–Operator lassen sich Ereignisse beschreiben, die jetzt oder zu einem beliebigen Zeitpunkt in der Zukunft eintreten.
◇ (Der Mond verdeckt die Sonne vollständig.)



Kombiniert man die Operatoren, so kommt man z.B. zu Aussagen über sich wiederholende Ereignisse, wie bei der Kombination „für immer schließlich“.
□◇ (Es ist Vollmond.)



Dreht man die Operatoren um, so kann man Aussagen über Ereignisse in der Zukunft machen, die ab einem bestimmten Zeitpunkt für immer gelten.
◇□ (Die Erde kühlt ab.)

Außerdem kann man die temporallogischen Operatoren mit den Operatoren der Aussagenlogik (∨,∧,¬,⇒) verknüpfen, um so zu komplexeren Aussagen zu gelangen.
□ (Die Erde kreist um die Sonne.) ∧
□ (Der Mond umkreist die Erde.)
⇒ □ (Der Mond kreist um die Sonne.)

4.2 Umsetzung von Temporallogik in ESTELLE

Um die oben vorgestellten Operatoren in ESTELLE umzusetzen, muß man sich zuerst Gedanken darüber machen, wie die beiden Operatoren (im folgenden nach den englischen Bezeichnungen „henceforth“ und „eventually“ genannnt) umgesetzt werden können. Eine Möglichkeit besteht darin, Funktionen zu definieren, deren Namen dem jeweiligen Operator entsprechen. Die Rümpfe dieser Funktionen sind dabei leer, d.h. jede Funktion liefert bei ihrem

Aufruf nur `TRUE` zurück. Bei der Deklaration dieser Funktionen wird auf die in Kapitel 3.3.2 beschriebene Methode der Verwendung von primitive–functions zurückgegriffen (s. Bild 17). Mit Hilfe dieser beiden Funktionen ist es möglich

```
FUNCTION AssertHenceforth : Boolean;
  primitive;
(*
  BEGIN
    AssertHenceforth := TRUE;
  END;
*)
FUNCTION AssertEventually : Boolean;
  pimitive;
(*
  BEGIN
    AssertEventually := TRUE;
  END;
*)
```

Bild 17: Definition der temporallogischen Funktionen in ESTELLE

die vier möglichen[11] Kombinationen von temporallogischen Operatoren wiederzugeben, indem die entsprechenden Operatoren durch ein logisches „und“ miteinander verknüpft werden. Anschließend kann dann der Ausdruck, auf den sich die Operatoren beziehen ebenfalls mit Hilfe eines logischen „und“ angefügt werden. Daraus ergibt sich folgende EBNF für die Verwendung von temporallogischen Operatoren in ESTELLE: Dabei ist zu beachten, daß alle temporallogischen Ausdrücke zwingend geklammert werden müssen, da in der Pascal–EBNF (s. [ISO82]) vorgeschrieben wird, daß nicht-atomare aussagenlogische Ausdrücke geklammert werden müssen. Die in Bild 18 gezeigte EBNF für Temporallogik muß in die gültige ESTELLE–EBNF für logische Ausdrücke eingegliedert werden. Daraus ergibt sich dann folgende EBNF:

```
TL-expression = "(" TL-Operator "AND"
                 ( "(" Expression ")" | TL-Operator "AND" "(" Expression ")" )
                ")".
TL-Operator   = "assertHenceforth" | "assertEventually".
```

Bild 18: EBNF der temporallogischen Operatoren

---

[11] Warum es nur vier geben kann wird in 4.3 beschrieben

```
Boolean-expression = expression | TL-expression.
TL-expression = "(" TL-Operator "AND"
                  ( "(" Expression ")" | TL-Operator "AND" "(" Expression ")" )
                ")".
TL-Operator  = "assertHenceforth" | "assertEventually".
```

Bild 19: Erweiterung der ESTELLE–EBNF

4.3 Reduktion der TL–Operatoren innerhalb von ESTELLE

Mit dem bisher eingeführten Konzept der temporallogischen Operatoren ist es nun möglich Aussagen über das zeitliche Verhalten von ESTELLE–Spezifikationen zu machen. Dabei sollten aber folgende Reduktionsregeln[12] beachtet werden, um lange und unnötige Schachtelungen von Operatoren zu vermeiden:

- Gleiche Operatoren werden reduziert. Das bedeutet:
  $\Box\ \Box\ p$ wird zu $\Box\ p$ und
  $\Diamond\ \Diamond\ p$ wird zu $\Diamond\ p$.

- Steht innerhalb einer Aussage kein temporallogischer Operator, so kann man implizit ein $\Box$ davorschreiben.
  Da alle bisher (s. [The95]) gemachten Zusicherungen über die gesamte Laufzeit der Spezifikation gelten, sind alle Zusicherungen bereits mit einem impliziten Henceforth versehen. Daher erscheint es nicht sinnvoll dieses $\Box$ auch noch explizit vor Zusicherungen zu schreiben in denen keine temporallogischen Operatoren Verwendung finden. Werden jedoch temporallogische Operatoren benutzt, so kann $\Box$ aus konsistenzgründen nicht weggelassen werden.

- $\Box\ \Diamond\ \Box$ wird zu $\Diamond\ \Box$
  Informell ist dies Reduktion klar, da eine Aussage, die „für alle Zeit“ „ab einem beliebigen Zeitpunkt für immer“ wahr ist, genauso auch „ab einem beliebigen Zeitpunkt für immer“ wahr ist. Der erste Henceforth Operator ist also unnötig.

---

[12] Die Beweise für die hier angegebenen Reduktionsregeln werden innerhalb dieser Arbeit nicht nachvollzogen. In [HuCr78] findet sich für das modallogische System 4.3. der Beweis, daß man mit nur acht Modalitäten auskommt. Der Beweis für die hier verwendete Temporallogik wird weitgehend analog geführt.

- $\Diamond \, \Box \, \Diamond$ wird zu $\Box \, \Diamond$
  Auch hier ist die Reduktion informell einsichtig, da für eine Aussage, die „Ab einem beliebigen Zeitpunkt" „hin und wieder" wahr wird, auch gilt, daß sie nur „hin und wieder" wahr wird. In diesem Fall ist also der erste Eventually Operator überflüssig.

Mit Hilfe dieser Reduktionsregeln kann man alle gewünschten Aussagen in temporaler Logik mit nur vier verschiedenen Ausdrücken machen. Diese Tatsache erhöht vor allem die Lesbarkeit von in dieser Weise spezifiziertem Quellcode.

4.4 Anmerkungen zur Implementierung

Integriert man die zuvor vorgestellten TL–Operatoren in eine ESTELLE–Spezifikation, so werden einige Probleme von der Tatsache ausgelöst, daß jedes System – und sei es noch so stabil – nur eine endliche Zeit läuft. Daher kann es passieren, daß bestimmte Zusicherungen, die mit Hilfe temporaler Operatoren gemacht wurden, nicht erfüllt werden, ohne daß genau beurteilt werden kann, ob ein Fehler in der Spezifikation vorlag, oder ob die Laufzeit des Systems nur zu klein war. Ein Ausweg aus diesem Problem würde darin bestehen, die temporallogischen Ausdrücke auf ein bestimmtes, vorher festzulegendes, Intervall zu beschränken. In der Praxis kann es aber durchaus vorkommen, daß sich Systeme für eine nicht berechenbare Zeit in einem *idle-wait*–Zustand befinden. Trifft dies zu, so kommt neben dem Problem der endlichen Laufzeit eines Systems noch die Frage auf, wie die Intervalle für die temporallogischen Operatoren gewählt werden müssen. Diese Frage ist aber genauso unlösbar, wie das Problem, daß die Laufzeit der Systeme endlich ist. Deshalb scheint uns dieser Weg nicht gangbar.
Aus diesem Grund wird bei der Implementierung der temporallogischen Konzepte folgendes Vorgehen vorgeschlagen:

- Innerhalb des Analyse–Werkzeugs sollten alle Vorkommen von temporallogischen Operatoren auf ihre Reduzierbarkeit überprüft werden (s. Kapitel 4.3)

- Wurde innerhalb des Spezifikationstextes der eventually–Operator ($\Diamond$) – allein oder in einer Kombination mit henceforth – verwendet, so müssen die nachfolgenden Ausdrücke gesondert behandelt werden.
  Dann müssen zu jedem Zeitpunkt, zu dem eine Überprüfung der Zusicherungen stattfindet, für jeden dieser Ausdrücke spezifische Zähler hochgezählt werden, die angeben, ob der Ausdruck bei der letzten Überprüfung erfüllt war, oder nicht. Nach dem Ende des Analysevorgangs

(normalerweise bei Laufzeitende des Systems), muß dann eine Auswertung der Zähler erfolgen. Dabei ist es zweckmäßig nur auf die Ausdrücke einzugehen, bei denen nicht klar ist, ob die Spezifikation sich so verhalten hat, wie der Entwickler es beabsichtigte. Das bedeutet, bei einem Ausdruck der Art $\Diamond\ p$ genügt es festzuhalten, ob der nachfolgende Ausdruck einmal erfüllt worden ist.

Bei der Operatorkombination $\Diamond\ \Box$ muß der nachfolgende Ausdruck immer erfüllt bleiben, nachdem er einmal erfüllt worden ist. Die Zusicherung ist allerdings auch erfüllt, wenn der Ausdruck während der Laufzeit nicht erfüllt worden ist, da $\Diamond\ \Box\ p$ nur besagt, daß p ab einem beliebigen Zeitpunkt in der Zukunft immer wahr sein muß. Hier sollte das Analyse–Werkzeug mitteilen, ob und ab welchem Zeitpunkt p erfüllt wurde. Ab diesem Zeitpunkt kann die temporallogische Zusicherung wie eine prädikatenlogische behandelt werden.

Wird eine Aussage der Art $\Box\ \Diamond\ p$ gemacht, so sollte von einem Analyse Werkzeug ausgegeben werden, ob, und wie oft p erfüllt war, da es auch hier wieder nicht zwingend notwendig ist, daß p während der Laufzeit des Systems erfüllt war.

4.5 Laufzeitverhalten

Wie schon in Kapitel 3.7.2 läßt sich von theoretischer Seite her nur schwer beurteilen, inwiefern sich die Verwendung temporallogischer Operatoren auf die Laufzeit von ESTELLE–Spezifikationen auswirken wird. Mit Sicherheit läßt sich nur sagen, daß sich die Laufzeit erhöhen wird, da innerhalb des Analyse–Werkzeugs bei jeder Überprüfung von Zusicherungen, die mit Hilfe von Temporallogik gemacht werden, Zähler mitgeführt werden, die das Verhalten dieser Zusicherungen über die Zeit repräsentieren sollen, wie schon in Kapitel 4.4 beschrieben. Wie hoch dieser Aufwand ist, hängt aber sehr stark von der Implementierung dieses Werkzeugs ab, so daß sich darüber keine gesicherte Aussage machen läßt.

5 Fallstudie

Als Beispiel wurde in dieser Arbeit durchgehend auf eine Aufgabe aus dem Praktikum „Entwicklung von Kommunikationssystemen“ zurückgegriffen, das im Sommersemester 1995 an der Universität Kaiserslautern stattgefunden hat.

In dieser Aufgabe ging es darum, ein Alternating–Bit–Protokoll in ESTELLE zu spezifizieren. Der zum Teil um die vorgestellten Konzepte erweiterte kommentierte Quelltext findet sich in Anhang A. In disem Beispiel wurden nicht alle Transitionen und Module durchgehend mit Zusicherungen versehen, vielmehr ging es darum, praktische Beispiele für die Anwendung der erweiterten Syntax zu geben. Bei der Betrachtung des Beispiels fällt auf, daß die meisten der gemachten Zusicherungen – besonders die transitionslokalen – trivialer Natur sind, und sich auch direkt aus dem Spezifikationstext ergeben würden. Das liegt aber in erster Linie an der – ebenfalls – trivialen Natur des Beispiels. Eine sinnvolle globale Zusicherung findet sich z.B. im Modul `PMBody`, in dem zugesichert wird, daß im Kanal `transmitService`, der Benutzer und Protokollmaschine führt, niemals mehr als 3 Nachrichten zwischengespeichert werden. Herleiten läßt sich diese Aussage aus dem Mechanismus der Sendekreditvergabe, der die Anzahl der vom User gesendeten Nachrichten auf genau 3 beschränkt. Dieses Limit steht aber nicht explizit im Spezifikationstext und ließe sich wohl auch durch ein automatisches Werkzeug nicht finden.

6 Zusammenfassung

In dieser Arbeit werden Konzepte vorgestellt, mit deren Hilfe das von Joachim Thees [TheBr95] angedachte Prinzip der Zusicherungen in ESTELLE–Spezifikationen erweitert wird. Dabei wird besonders darauf geachtet, daß keine Änderungen am bestehenden ESTELLE–Standard vorgenommen werden. D.h. eine Spezifikation, die die hier vorgestellten Konzepte verwendet, entspricht weiterhin allen semantischen rEGELN des ESTELLE–Standards. Die besonderen Möglichkeiten, die sich aus der Verwendung dieser Konzepte ergeben, können erst mit Hilfe eines noch zu entwickelnden Analyse–Werkzeugs sinnvoll genutzt werden.

Mit Hilfe einiger besonderer Identifier (z.B. dem Identifier `assertMainState`, der den aktuellen Hauptzustand der Spezifikation beinhaltet), die eine kontextsensitive Semantik besitzen, ist es nun möglich, Informationen über den Hauptzustand und die Interaktionspunkte einer Spezifikation zur Laufzeit zu bekommen. Das stellt insofern eine Erweiterung des Konzepts von Joachim Thees dar, als dieser nur statische Aussagen machte (s. [The95]). Diese Informationen können dazu benutzt werden, um innerhalb der beiden Zusicherungsfunktionen `BodyAssert` und `TransAssert` Zusicherungen zu machen, die in dieser Art nicht unmittelbar aus dem Spezifikationstext ersichtlich sind. Dabei können bei den Interaktionspunkten z.B. sowohl Aussagen über die momentane Länge einer Warteschlange, über die Art der ersten Nachricht als auch darüber gemacht werden, ob die erste Nachricht in der Schlange wirklich für den entsprechenden Interaktionspunkt bestimmt ist.

Im zweiten Teil der Arbeit wurde eine Möglichkeit aufgezeigt, wie temporallogische Operatoren in den ESTELLE–Standard integriert werden können. Dies geschieht mit den beiden Funktionen `assertHenceforth` und `assertEventually`, die den logischen Operatoren $\Box$ und $\Diamond$ entsprechen. Weiterhin werden diese temporallogischen Aussagen in Zusicherungsfunktionen verwendet, um Zusicherungen zu machen, deren Gültigkeit erst zu einem, zum Zeitpunkt der Spezifikation noch unbestimmten, Zeitpunkt in der Zukunft beginnt.

7 Ausblick

Bei den hier vorgestellten Konzepten bleiben noch einige Bereiche ungeklärt, die aber den Rahmen dieser Arbeit sprengen würden. Deshalb werden sie in diesem Kapitel nur umrissen.

- **Verstöße gegen Zusicherungen**
  Bisher wurde noch nicht diskutiert, wie sich das Analyse–Werkzeug bei einem Verstoß gegen gemachte Zusicherungen zur Laufzeit verhalten soll. Es erscheint sinnvoll, in so einem Fall nicht direkt die Bearbeitung der Spezifikation abzubrechen, sondern vielmehr diese Vorfälle in einer Log–Datei mitzuprotokollieren. Denkbar wäre in diesem Zusammenhang auch eine Bewertung der Schwere der Verstöße. So könnte zum Beispiel eine falsche Nachricht an einem IP schwerer bewertet werden, als eine Variable, deren Wert außerhalb der zugesicherten Grenzwerte liegt. Diese Gewichtung kann aber nicht global vorgegeben werden, sondern der Anwender sollte in der Lage sein, seine eigene Gewichtung anzugeben. Prinzipiell muß bei der Betrachtung des Verhaltens des Analyse–Tools aber wohl unterschieden werden, ob es sich bei der betrachteten Spezifikation um einen Prototyp oder um eine genutzte Implementierung handelt, denn innerhalb einer genutzten Implementierung sollten keine Verstöße gegen gemachte Zusicherungen mehr vorkommen.

- **Zusicherungen über Timeouts**
  Prinzipiell sollte es möglich sein, auch Zusicherungen über die in ESTELLE benutzten `DELAY`–Klauseln zu machen. Da die Umsetzung solcher Zusicherungen den Rahmen dieser Arbeit sprengen würde, wurden sie nicht weiter berücksichtigt.

- **Aussagen über Verbindungsstrukturen** In dieser Arbeit wird keine Möglichkeit aufgezeigt, wie Aussagen über Verbindungsstrukturen innerhalb von ESTELLE gemacht werden können. Vom Prinzip her ließen sich Zusicherungen über diese Strukturen jedoch ähnlich realisieren, wie die Zusicherungen über die Interaktionspunkte einer Spezifikation. D.h., es wird eine Funktion, etwa `assertIsConnectedWith (Module : assertModVarType) : assertModVarSetType`, definiert, die dann die Menge der Verbindungen dieses Moduls zurückgibt.

- **Aussagen über Modulstrukturen** Analog zu den Aussagen überdie Verbindungsstrukturen sollte es auch möglich sein, Aussagen über die Anzahl von Sohnmodulen, oder über die Verwendung von Modulvaraiblen zu machen.
  In dieser Arbeit wurde gefordert, daß auf jedes Modul genau eine Modulvariable zeigt. Es wäre allerdings auch denkbar Spezifikationen zu schreiben, bei denen Modulvariablen erst im Laufe der Spezifikation oder gar nich instanziiert werden.

A Beispiel

```
SPECIFICATION AB_Protocol;

  { Praktikum Entwicklung von Kommunikationssystemen SoSe 95,
    Jan Bredereke, Reinhard Gotzhein
  }

  { Einfuehrung einer Flusskontrolle zwischen den Protokollmaschinen,
    Erweiterung um "Backpressure"-Flusskontrolle hin zum sendenden User.
    Einfuehrung eines unzuverlaessigen Mediums;
    daher wurden Massnahmen zur Behebung von Nachrichtenverlusten notwendig:
    das Alternating-Bit-Protokoll;
    Erweiterung auf eine symmetrische Uebertragung in beide Richtungen.
  }

  TIMESCALE SECONDS; { Zeiteinheit fuer die DELAY-Klauseln. }

  CONST
    retransmissionTime = 5;
    firstSeqNum = 0;

  TYPE
    netUserRangeType = 0..1;                    { Bereich der Indizes fuer   }
```

```
                                              { die Nutzer des Netzwerkes. }
  userKindType = (userA_kind, userB_kind);    { Parametertyp: Was fuer ein }
                                              { User? userA sendet 'A' bis }
                                              { 'F', userB sendet '1' bis  }
                                              { '6'.                       }
  sduType    = CHAR;                          { Service Data Unit.         }
  pduIdType  = (dataRI, ack);                 { Selektortyp fuer den       }
                                              { folgenden PDU-Typ.         }
  seqNumType = 0..1;                          { Sequenznummern-Typ.        }
  pduType    = RECORD                         { Protocol Data Unit:        }
                 seqNum: seqNumType;          { - Sequenznummer,           }
                 CASE pduId: pduIdType OF     { - Typ der PDU,             }
                   dataRI: (netData: sduType); {   * Nettodaten oder        }
                   ack:  ()                   {   * Bestaetigung.          }
               END;

{ Der Kanal, ueber den die Protokollmaschine ihrem User ihren Dienst
  zur Verfuegung stellt: }
CHANNEL transmitService(user, provider);
  BY user:     T_dataReq(sdu: sduType);      { Aufforderung zur Datenueber-  }
                                             { tragung.                      }
  BY provider: T_dataInd(sdu: sduType);      { Anzeige einer Datenuebertrag. }
               T_creditInd(credit: INTEGER); { Gewaehrung von Sendekredit    }
                                             { an den User gegenueber der PM }
                                             { fuer die lokale Fluss-        }
                                             { kontrolle.                    }

{ Der Kanal, ueber den das Netzwerk einer Protokollmaschine ihren Dienst
  zur Verfuegung stellt: }
CHANNEL networkService(user, provider);
  BY user:     N_dataReq(pdu: pduType);
  BY provider: N_dataInd(pdu: pduType);

{ Aeussere Schnittstelle fuer einen User: }
MODULE userType SYSTEMPROCESS(userKind: userKindType);
  IP userToPm: transmitService(user) INDIVIDUAL QUEUE;
  END;
```

```
{ Aeussere Schnittstelle fuer eine Protokollmaschine: }
MODULE protocolMachineType SYSTEMPROCESS;
  IP pmToUser: transmitService(provider) INDIVIDUAL QUEUE;
     pmToNet:  networkService(user) INDIVIDUAL QUEUE;
  END;

{ Aeussere Schnittstelle fuer das Netzwerk: }
MODULE networkType SYSTEMPROCESS;
  { Das Netzwerk hat hier nur zwei IPs. Damit kann die Adressierung implizit
    erfolgen: Was in den einen IP hineingetan wird, kommt am anderen
    wieder heraus. }
  IP netToPm: ARRAY[netUserRangeType] OF networkService(provider)
    INDIVIDUAL QUEUE;
  END;

{**********************************************************************}
{*                                                                    *}
{* Hier beginnen die fuer die Zusicherungen benoetigten Typ-Definitionen *}
{*                                                                    *}
{**********************************************************************}

TYPE

assertIPNamesType =
          (assertuserToPm,             { In diesem Aufzaehlungstyp sind }
           assertpmToUser,             { alle in diesem Modul vorkom-   }
           assertpmToNet,              { menden IPs enthalten.          }
           assertnetToPm,
           assertAllIPs);
assertModVarType =
          (assertUserA,                { Aufzaehlungstyp, der alle Mo-  }
           assertUserB,                { dulvariablen enthaelt. Diese   }
           assertPM_A,                 { werden fuer Zusicherungen bei  }
           assertPM_B,                 { geschachtelten Modulinstanzen  }
           assertNetwork,              { benoetigt.                     }
           assertSelf);
```

```
assertMsgType     =
          (assertT_dataReq,            { Hier werden alle in der Spezi- }
           assertT_dataInd,            { fikation vorkommenden Nach-    }
           assertT_creditInd,          { richtentypen abgelegt.         }
           assertN_dataReq,
           assertN_dataInd);
assertIPInfoType =
           RECORD
             IPName :                  assertIPNamesType;
                                       { Name des IP                    }
             IsMsgInQueue :            Boolean;
                                       { Befindet sich eine Nachricht   }
                                       { fuer den entsprechenden IP an  }
                                       { erster Stelle in der Queue?    }
             MsgCount :                Integer;
                                       { Wieviele Nachrichten stehen in }
                                       { der Queue?                     }
             CASE TypeOfFirstMsg : assertMsgType OF
                                       { Typ der ersten Nachricht in    }
                                       { der Queue.                     }
               assertT_dataReq,                      { Hier folgen      }
               assertT_dataInd:    (sdu: sduType);   { die moeglichen   }
               assertT_creditInd:  (credit: Integer); { Parameter.      }
               assertN_dataReq,
               assertN_dataInd:    (pdu: pduType);
             END;

{***********************************************************************}
{*                                                                     *}
{* Ende der fuer die Zusicherungen benoetigten Typ-Definitionen        *}
{*                                                                     *}
{***********************************************************************}
{-----------------------------------------------------------------------}
```

```
{**********************************************************************}
{*                                                                    *}
{*          Die fuer Zusicherungen benoetigten Funktionen             *}
{*                                                                    *}
{**********************************************************************}

FUNCTION assertHenceforth : Boolean;
  primitive;
(*
  BEGIN
    assertHenceforth := TRUE;
  END;
*)
FUNCTION assertEventually : Boolean;
  primitive;
(*
  BEGIN
    assertEventually := TRUE;
  END;
*)
FUNCTION assertFillIPRecord (assertModuleName: assertModVarType,
                             assertRecordName: assertIPNamesType):
              assertIPInfoType;
         primitive;
(*
  VAR assertIPRecord: assertIPInfoType;

     BEGIN
       assertIPRecord.
           IPName := assertRecordName;        { Die hier eingetragenen }
       assertIPRecord.                        { Werte haben semantisch }
           IsMsgInQueue := FALSE;             { keine Auswirkung. Sie  }
       assertIPRecord.                        { werden zur Laufzeit    }
           MsgCount := 0;                     { durch das Analyse-Tool }
       assertIPRecord.                        { durch die realen Werte }
           TypeOfFirstMsg := assertT_dataReq; { ersetzt.               }
       assertIPRecord.
           sdu := '@';
       assertFillIPRecord := assertIPRecord;
     END;
 *)
```

```
{--------------------------------------------------------------------------}
{ Die (generische) Realisierung eines Users: }
BODY userBody FOR userType;
  { Aufzaehlungstyp fuer Zusicherungen ueber die Hauptzustaende          }
  { des Moduls.                                                          }
  TYPE assertMainStateType = (assertsendingAndReceiving,
                              assertonlyReceiving);


  VAR sendMsg:     sduType;
      lastSendMsg: sduType;
      receivedMsg: sduType;
      sendCredit: INTEGER;

  { Variablen, die fuer die Zusicherungen benoetigt werden.             }

      assertMainState: assertMainStateType;
      assertIPInfo:    assertIPInfoType;


  FUNCTION  BodyAssert : Boolean;
  { Modulglobale Zusicherungen                                          }
   BEGIN
     BodyAssert :=
         (assertEventually AND assertHenceforth AND
                (sendMsg = lastSendMsg))
     AND
         (assertEventually AND assertHenceforth AND
                (assertMainState = assertonlyReceiving))
     AND
         (assertEventually AND assertHenceforth AND
                (sendCredit > 0))
     AND
         (assertHenceforth AND (sendCredit >= 0));
   END;

  STATE sendingAndReceiving,
        onlyReceiving;

  STATESET either = [sendingAndReceiving, onlyReceiving];
```

```
    INITIALIZE  { Initialisierung: }
      TO sendingAndReceiving { In den anfaenglichen Hauptzustand. }
NAME userInit:
BEGIN    { Und auch die Variablen initialisieren. }
          CASE userKind OF
  userA_kind:
    BEGIN
      sendMsg := 'A';
      lastSendMsg := 'F';
    END;
  userB_kind:
    BEGIN
      sendMsg := '1';
      lastSendMsg := '6';
    END;
  END;
  sendCredit := 0;      { Zuerst ist noch kein Sendekredit vorhanden. }
  receivedMsg := '@'    { (Es ist immer gut, wenn Variablen einen     }
END;                    { definierten Wert haben.)                    }

    { Gib der Protokollmaschine nacheinander Uebertragungswuensche fuer }
    { die Nutzdaten, sofern Sendekredit fuer die lokale                 }
    { Flusskontrolle vorhanden ist.                                     }
    TRANS
      FROM sendingAndReceiving
        TO SAME
          PROVIDED (sendMsg <> lastSendMsg) AND (sendCredit > 0)

          FUNCTION TransAssert : Boolean;
          { Zusicherungen, die erfuellt sind, wenn diese Trans. schaltet }
           BEGIN
             TransAssert :=
                  (assertMainState = assertsendingAndReceiving)
             AND
                  (sendMsg <> lastSendMsg)
             AND
                  (sendCredit >= 1);
           END;
```

```
NAME userSending:
            BEGIN
              OUTPUT userToPm.T_dataReq(sendMsg);
              sendMsg := SUCC(sendMsg);
              sendCredit := sendCredit - 1
            END;

    { Empfange einen Sendekredit von der Protokollmaschine fuer die lokale }
    { Flusskontrolle.                                                      }
    TRANS
      FROM either
        TO SAME
          WHEN userToPm.T_creditInd(credit)

          FUNCTION TransAssert : Boolean;
          { Zusicherungen, die erfuellt sind, wenn diese Trans. schaltet }
           BEGIN
            TransAssert :=
                 (credit > 0);
           END;
NAME userReceivingSendCredit:
            BEGIN
              sendCredit := sendCredit + credit
            END;

    { Gib der Protokollmaschine den letzten Uebertragungswunsch }
    { (sofern Sendekredit fuer die lokale  Flusskontrolle       }
    { vorhanden ist).                                           }
    TRANS
      FROM sendingAndReceiving
        TO onlyReceiving
          PROVIDED (sendMsg = lastSendMsg) AND (sendCredit > 0)

          FUNCTION TransAssert : Boolean;
          { Zusicherungen, die erfuellt sind, wenn diese Trans. schaltet }
           BEGIN
            TransAssert :=
                 (assertMainState = assertsendingAndReceiving)
            AND
                 (sendCredit >= 1)
            AND
```

```
                 (sendMsg = lastSendMsg);
           END;
NAME userSendingLast:
            BEGIN
              OUTPUT userToPm.T_dataReq(sendMsg);
              sendCredit := sendCredit - 1
            END;
    TRANS
      FROM either
        TO SAME
          WHEN userToPm.T_dataInd(sdu)
NAME userReceiving:
            BEGIN
              receivedMsg := sdu;
              {$C$ printf ("user meldet den Empfang einer Nachricht \n"); }
            END;
  END; { BODY userBody FOR userType }

  {--------------------------------------------------------------------------}

  BODY PM_body FOR protocolMachineType;
    CONST
      creditLimit = 3;

    { Aufzaehlungstyp fuer Zusicherungen ueber die Hauptzustaende           }
    { des Moduls                                                            }

    TYPE assertMainStateType = (assertestab,
                                assertwaitAck);
```

```
VAR
  msg: pduType;
  userSendCredit: INTEGER;
  sendSeqNum:    seqNumType;
  receiveSeqNum: seqNumType;

{ Variablen, die fuer die Zusicherungen benoetigt werden.               }

  assertMainState: assertMainStateType;
  assertIPInfo:    assertIPInfoType;


{ Modulglobale Zusicherungen                                            }
FUNCTION BodyAssert : Boolean;
BEGIN
 assertIPInfo := assertFillIPRecord (assertSelf, assertpmToUser);
 { Mit dieser Funktion werden zur Laufzeit die Informationen ueber den  }
 { IP pmToUser des momentan aktiven Moduls beschafft.                   }
 BodyAssert :=
     (userSendCredit <= creditLimit)
 AND
     (assertEventually AND assertHenceforth AND
            (userSendCredit > 0))
 AND
     (assertHenceforth AND assertEventually AND
            (userSendCredit <> 0))
 AND
     (assertIPInfo.MsgCount <= creditLimit);
END;

FUNCTION nextSeqNum(currSeqNum: seqNumType): seqNumType;
BEGIN
  nextSeqNum := SUCC(currSeqNum) MOD 2;
END;
```

```
    { Eine totale Funktion auf einer nur manchmal definierten Komponente
      eines varianten Records: }
    FUNCTION isAckWithCorrectSeqNum(pdu: pduType; seqNum: seqNumType):
      BOOLEAN;
    BEGIN
      IF pdu.pduId <> ack THEN
        isAckWithCorrectSeqNum := FALSE
      ELSE
        isAckWithCorrectSeqNum := (pdu.seqNum = seqNum)
    END;

    { Eine totale Funktion auf einer nur manchmal definierten Komponente
      eines varianten Records: }
    FUNCTION isAckWithIncorrectSeqNum(pdu: pduType; seqNum: seqNumType):
      BOOLEAN;
    BEGIN
      IF pdu.pduId <> ack THEN
        isAckWithIncorrectSeqNum := FALSE
      ELSE
        isAckWithIncorrectSeqNum := (pdu.seqNum <> seqNum)
    END;

    STATE
      estab,
      waitAck;

    STATESET
      either = [estab,waitAck];

    INITIALIZE
      TO estab
NAME PM_init:
        BEGIN
          userSendCredit := creditLimit;
          sendSeqNum    := firstSeqNum;
          receiveSeqNum := firstSeqNum;
        END;
```

```
    { Gib einen vorhandenen Sendekredit der lokalen Flusskontrolle an    }
    { den User weiter. Falls gleichzeitig auch eine andere Transition    }
    { schaltbereit ist, findet eine nichtdeterministische Auswahl statt. }
    TRANS
      FROM either
        TO SAME
          PROVIDED userSendCredit > 0

          FUNCTION TransAssert: Boolean;
          { Zusicherungen, die erfuellt sind, wenn diese Trans. schaltet }
          BEGIN
           TransAssert :=
                (userSendCredit > 0);
          END;
NAME PM_grantingSendCredit:
            BEGIN
              OUTPUT pmToUser.T_creditInd(userSendCredit);
              userSendCredit := 0
            END;

    TRANS
      FROM estab
        TO waitAck
          WHEN pmToUser.T_dataReq(sdu)
NAME PM_sending:
            BEGIN
              msg.seqNum  := sendSeqNum;
              msg.pduId   := dataRI;
              msg.netData := sdu;
              OUTPUT pmToNet.N_dataReq(msg);
              userSendCredit := userSendCredit + 1;
            END;

    TRANS
      FROM waitAck
        TO estab
          WHEN pmToNet.N_dataInd(pdu)
            PROVIDED isAckWithCorrectSeqNum(pdu, sendSeqNum)
```

```
NAME PM_ackReceiving:
              BEGIN
                sendSeqNum := nextSeqNum(sendSeqNum);
              END;

    { Ignoriere die Wiederholung einer bereits empfangenen Bestaetigung. }
    TRANS
      FROM waitAck
        TO SAME
          WHEN pmToNet.N_dataInd(pdu)
            PROVIDED isAckWithIncorrectSeqNum(pdu, sendSeqNum)
NAME PM_repeatedAck:
              BEGIN
              END;

    { Wiederhole die letzten Daten, wenn zu lange keine Bestaetigung kommt. }
    TRANS
      FROM waitAck
        TO SAME
          DELAY (retransmissionTime)
NAME PM_timeout:
            BEGIN
              OUTPUT pmToNet.N_dataReq(msg)
            END;

    { Bestaetige den Empfang einer PDU. Wenn sie noch nicht empfangen wurde, }
    { gib ihre Nutzdaten nach oben weiter.                                    }
    TRANS
      FROM either
        TO SAME
          WHEN pmToNet.N_dataInd(pdu)
            PROVIDED pdu.pduId = dataRI
              VAR tmpMsg: pduType;
NAME PM_receiving:
              BEGIN
                tmpMsg.seqNum := pdu.seqNum;
                tmpMsg.pduId := ack;
                OUTPUT pmToNet.N_dataReq(tmpMsg);
                IF pdu.seqNum = receiveSeqNum THEN
                BEGIN
                  OUTPUT pmToUser.T_dataInd(pdu.netData);
```

```
                receiveSeqNum := nextSeqNum(receiveSeqNum)
              END
            END;
  END; { BODY PM_body FOR protocolMachineType }

  {--------------------------------------------------------------------------}

  BODY networkBody FOR networkType;
    VAR  { Bedeutungen: Siehe INITIALIZE-Transition. }
      nextLoss: ARRAY[netUserRangeType] OF INTEGER;
      lossCtr:  ARRAY[netUserRangeType] OF INTEGER;

    FUNCTION partner(myIndex: netUserRangeType): netUserRangeType;
      BEGIN
        partner := (myIndex + 1) MOD 2
      END;

    FUNCTION simpleRandom(oldValue: INTEGER): INTEGER;
      BEGIN
         simpleRandom := (oldValue * 7 + 1) MOD 5
      END;

    STATE active;
    INITIALIZE
      TO active
NAME networkInit:
        BEGIN     { Zu Anfang gilt: }
          nextLoss[0] := 2; { Die 2. Nachricht von User A wird verloren gehen. }
          nextLoss[1] := 3; { Die 3. Nachricht von User B wird verloren gehen. }
          ALL idx:netUserRangeType DO lossCtr[idx] := 0
                            { Noch wurde keine Nachricht verloren. }
        END;
```

```
    TRANS
      ANY idx:netUserRangeType DO
      FROM active
        TO SAME
          WHEN netToPm[idx].N_dataReq(pdu)
            PROVIDED nextLoss[idx] <> lossCtr[idx]
NAME networkDataTransfer:
              BEGIN
        OUTPUT netToPm[partner(idx)].N_dataInd(pdu);
                lossCtr[idx] := SUCC(lossCtr[idx]);
              END;
            PROVIDED nextLoss[idx] = lossCtr[idx]
NAME networkDataLoss:
              BEGIN
                lossCtr[idx] := 0;
                nextLoss[idx] := simpleRandom(nextLoss[idx])
              END;
  END; { BODY networkBody FOR networkType }

  {------------------------------------------------------------------------------}
  MODVAR
    userA: userType;
    userB: userType;
    PM_A: protocolMachineType;
    PM_B: protocolMachineType;
    network: networkType;

  INITIALIZE
NAME aufgabe2Init:
    BEGIN
      INIT userA WITH userBody(userA_kind);
      INIT userB WITH userBody(userB_kind);
      INIT PM_A WITH PM_body;
      INIT PM_B WITH PM_body;
      INIT network WITH networkBody;
      CONNECT userA.userToPm TO PM_A.pmToUser;
      CONNECT userB.userToPm TO PM_B.pmToUser;
      CONNECT PM_A.pmToNet TO network.netToPm[0];
      CONNECT PM_B.pmToNet TO network.netToPm[1];
    END;
END.
```

B Überblick über die Erweiterungen des ESTELLE–Standards

In diesem Kapitel werden noch einmal alle neu eingeführten ESTELLE–Operatoren gesammelt aufgeführt und es wird ihre kontextsensitive Semantik definiert.
In diesem Kapitel wird die normale EBNF–Syntax verwendet. Zusätzlich werden zwei weitere Symbole eingeführt:

- Ψ – Die mit diesem Zeichen verbundenen Ausdrücke müssen alle verwendet werden, wobei aber die Reihenfolge beliebig ist.
- $\circ$ – Repräsentiert die Konkatenation eines Literals mit einem EBNF–Token.

Dieses Kapitel soll in erster Linie als Referenz bei der Verwendung von Zusicherungen dienen. Dabei haben die einzelnen Abschnitte folgenden Aufbau:

- **Semantik**
  Informelle Definition der kontextabhängigen Semantik dieser ESTELLE–Erweiterung.
- **Erste Erwähnung**
  Seitennummer, auf der diese Erweiterung zum ersten Mal eingeführt wird.
- **Definition** Beschreibung der Stelle, an der die Erweiterung benutzt werden muß. Dabei werden bei übergeordneten komplexen Typen auch die EBNF und jeweils entsprechende Ausschnitte aus der ESTELLE EBNF angegeben.

assertAllIPs

- **Semantik**
  Literal, das innerhalb von Zusicherungen über Interaktionspunkte dazu dient, Aussagen über alle Interaktionspunkte eines Moduls zu machen. Benutzt wird `assertAllIPs` als Parameter bei der Funktion `assertFillIPRecord` (s.d.).
- **Erste Erwähnung**
  Auf Seite 17.

- **Definition**
  Dieses Literal wird innerhalb der Definition des `assertIPNamesType`–Aufzählungstypen eingeführt (s.d.).

assertIPInfoType

- **Semantik**
  Dieser Typ enthält in einem varianten `RECORD` alle für die vorgestellten Konzepte relevanten Information über einen Interaktionspunkt.

- **Erste Erwähnung**
  Auf Seite 19.

- **Definition**
  Die Definition erfolgt zusammen mit den übrigen spezifikationsglobalen Typen (s. Bild 3.4).

```
assertIPInfoType =     ``RECORD`` ConstPart; VarPart ``END;``.
ConstPart =            ``IPName: assertIPNamesType`` Ψ
                       ``IsMsgInQueue: Boolean`` Ψ
                       ``MsgCount: Integer``.
VarPart =              ``CASE TypeOfFirstMsg: assertMsgType OF``
                       msg-identifier ``;`` Ψ {msg-identifier “; “}.
msg-identifier =       IDENTIFIER {'','' IDENTIFIER} '':''
                       +{interaction-definition}.
```

assertIPNamesType

- **Semantik**
  Dieser Aufzählungstyp enthält die Namen aller in dieser Spezifikation vorkommenden Interaktionspunkte. Aus Gründen der Wertebreichsverwaltung in Estelle muß allen Interaktionspunktnamen das Wort `assert` vorangestellt werden. Da die Namen der Interaktionspunkte in einem Aufzählungstyp verwaltet werden, kommt hinzu, daß alle Interaktionspunkte der Spezifikation verschieden benannt sein müssen.

- **Erste Erwähnung**
  Auf Seite 19.

- **Definition**
  Die Definition erfolgt zusammen mit den übrigen spezifikationsglobalen Typen (s. Bild 3.4).

```
assertIPNamesType =     ‘‘(‘‘ IPName [‘‘,‘‘ IPName]‘‘)‘‘.
IPName =                ‘‘assert‘‘ ○ ip-identifier.
```

assertMainState

- **Semantik**
  `assertMainState` ist eine Variable vom Typ `assertMainStateType`. In der erweiterten Semantik enthält sie den Hauptzustand des jeweiligen Moduls.

- **Erste Erwähnung**
  Auf Seite 11.

- **Definition**
  Die Deklaration erfolgt im Variablen-Deklarationsteil des jeweiligen Moduls.

assertMainStateType

- **Semantik**
  Dieser Aufzählungstyp enthält die Namen aller in dieser Spezifikation vorkommenden Hauptzustände. Aus Gründen der Wertebreichsverwaltung in ESTELLE muß allen Hauptzustandsnamen das Wort `assert` vorangestellt werden.

- **Erste Erwähnung**
  Auf Seite 11.

- **Definition**
  Die Definition erfolgt zusammen mit den übrigen spezifikationsglobalen Typen (s. Bild 3.4).

```
assertIPNamesType =     ‘‘(‘‘ MainStateName [‘‘,‘‘ MainStateName]‘‘)‘‘.
MainStateName =         ‘‘assert‘‘ ○ state-identifier.
```

assertMsgType

- **Semantik**
  Dieser Aufzählungstyp enthält alle in der Spezifikation vorkommenden Nachrichtennamen. Aus Gründen der Konsistenz wird jedem Namen das Wort `assert` vorangestellt. Wie auch bei dem Aufzählungstyp `assertIPNamesType` ist es erforderlich, daß alle in einer Spezifikation vorkommenden Nachrichtentypen unterschiedliche Namen haben.
- **Erste Erwähnung**
  Auf Seite 19.
- **Definition**
  Die Definition erfolgt zusammen mit den übrigen spezifikationsglobalen Typen (s. Bild 3.4).

```
assertMsgType =         ‘‘(‘‘ Msg [‘‘,‘‘Msg]‘‘)‘‘.
Msg =                   ‘‘assert‘‘ ○ message-identifier.
```

assertModVarType

- **Semantik**
  Dieser Aufzählungstyp enthält alle in der Spezifikation vorkommenden Modulvariablen. Aus Gründen der Konsistenz wird jedem Namen das Wort `assert` vorangestellt.
- **ErsteErwähnung**
  Auf Seite 25.
- **Definition**
  Die Definition erfolgt zusammen mit den übrigen spezifikationsglobalen Typen (s. Bild 3.4).

```
assertModVarType =      ‘‘(‘‘ ModVar [‘‘,‘‘ModVar]‘‘)‘‘.
ModVar =                ‘‘assert‘‘ ○ module-identifier.
```

assertSelf

- **Semantik**
  Literal, das beim Aufruf der Funktion `assertFillIPRecord` (s.d.) dazu dient, Information über die momentan gültige Modulinstanz zu bekommen. Es könnte stattdessen auch die jeweilige Instanz–Variable benutzt werden, doch die Verwendung dieses Literals macht die Spezifikation in unseren Augen lesbarer.

- **Erste Erwähnung**
  Auf Seite 25.

- **Definition**
  Dieses Literal wird bei der Definition des Aufzählungstyps `assertModVarType` eingeführt (s.d.).

FUNCTION assertEventually : Boolean;

- **Semantik**
  Eine primitive–Funktion, die immer `TRUE` zurückliefert. In der erweiteren Semantik repräsentiert sie den temporallogischen Operator $\diamond$. Verknüpfungen mit `assertHenceforth` erfolgen mittels des `AND` Operators. Dabei entspricht die Klammerung der aus der Aussagenlogik bekannten.

- **Erste Erwähnung**
  Auf Seite 32.

- **Definition**
  Die Definition dieser Funktion erfolgt innerhalb des Deklarationsteils der Spezifikation (s. Bild 3.4). Da es sich um eine primitive Funktion handelt, kann sie in allen Modulen dieser Spezifikation benutzt werden. Der in Kommentaren angegebene Funktionsrumpf kann auch weggelassen werden, er dient lediglich dazu, den Spezifikationstext lesaberer zu machen.

```
TL-expression =
      ‘‘(‘‘ TL-Operator ‘‘AND‘‘
      (‘‘(‘‘ Expression ‘‘)‘‘|‘‘AND‘‘ TL-Operator ‘‘AND‘‘ Expression) ‘‘)‘‘.
TL-Operator =
      ‘‘assertHenceforth‘‘ | ‘‘assertEventually‘‘.
```

FUNCTION assertFillIPRecord

- **Semantik**
  Diese Funktion füllt innerhalb der erweiterten Semantik eine Variable vom Typ `assertIPNamesType` mit den Werten des entsprechenden Interaktionspunktes. Aufgerufen wird diese Funktion mit dem Namen des Moduls, auf das sich die Information beziehen soll und dem Namen des entsprechenden IPs.

- **Erste Erwähnung**
  Auf Seite 20.

- **Definition**
  Die Definition dieser Funktion erfolgt innerhalb des Deklarationsteils der Spezifikation (s. Bild 3.4). Da es sich um eine primitive Funktion handelt, kann sie in allen Modulen dieser Spezifikation benutzt werden. Der in Kommentaren angegebene Funktionsrumpf kann auch weggelassen werden, er dient lediglich dazu, den Spezifikationstext lesaberer zu machen. Die genaue Zuweisungsfolge innerhalb der Funktion findet sich in Bild 10.

FUNCTION assertHenceforth : Boolean;

- **Semantik**
  Eine Funktion, die immer `TRUE` zurückliefert. In der erweiteren Semantik repräsentiert sie den temporallogischen Operator $\Box$. Verknüpfungen mit `assertEventually` erfolgen mittels des `AND` Operators. Die Klammerung entspricht dabei der aus der Aussagenlogik bekannten.

- **Erste Erwähnung**
  Auf Seite 32.

- **Definition**
  Die Definition dieser Funktion erfolgt zu Beginn der Spezifikation. Da es sich um eine primitive Funktion handelt, kann sie in allen Modulen

dieser Spezifikation benutzt werden. Der in Kommentaren angegebene Funktionsrumpf kann auch weggelassen werden, er dient lediglich dazu, den Spezifikationstext lesaberer zu machen.

```
TL-expression =
      ‘‘(‘‘ TL-Operator ‘‘AND‘‘
      (‘‘(‘‘ Expression ‘‘)‘‘|‘‘AND‘‘ TL-Operator ‘‘AND‘‘ Expression) ‘‘)‘‘.
TL-Operator =
      ‘‘assertHenceforth‘‘ | ‘‘assertEventually‘‘.
```

FUNCTION BodyAssert : Boolean;

- **Semantik**
  Die Funktion in der alle für alle Transitionen dieses Moduls gültigen Zusicherungen angegeben werden. Der Rückgabewert dieser Funktion ergibt sich aus der Konjunktion der einzelnen Zusicherungen.
- **Erste Erwähnung**
  Auf Seite 5.
- **Definition**
  Die Definition dieser Funktion erfolgt nach der `INITIALIZE`–Transition des jeweiligen Moduls. Dadurch ergeben sich folgende EBNFs:

```
body-definition =        declaration-part
                         initialization-part
                         BodyAssert
                         transition-declaration-part.

BodyAssert =             Header; Body.
Header =                 ‘‘FUNCTION BodyAssert: Boolean‘‘.
Body =                   ‘‘BEGIN BodyAssert:= ‘‘ Boolean-expression
                         {‘‘AND‘‘ Boolean-Expression} ‘‘END;‘‘.
Boolean-Expression =     expression | TL-expression.
```

FUNCTION TransAssert : Boolean;

- **Semantik**
  Die Funktion in der alle für diese Transition gültigen Zusicherungen angegeben werden. Der Rückgabewert dieser Funktion ergibt sich aus der Konjunktion der einzelnen Zusicherungen.

- **Erste Erwähnung**
  Auf Seite 5.

- **Definition**
  Die Definition dieser Funktion erfolgt vor der `NAME`–Klausel der jeweiligen Transition. Dadurch ergeben sich folgende EBNFs:

```
transition-block =      CONSTANT-DEFINITION-PART
                        TYPE-DEFINITION-PART
                        VARIABLE-DEFINITION-PART
                        PROCEDURE-AND-FUNCTION-DECLARATION-PART
                        TransAssert
                        [transition-name] STATEMENT-PART.

TransAssert =           Header; Body.
Header =                ‘‘FUNCTION TransAssert: Boolean‘‘.
Body =                  ‘‘BEGIN TransAssert:= ‘‘ Boolean-expression
                        {‘‘AND‘‘ Boolean-Expression} ‘‘END;‘‘.
Boolean-Expression =    expression | TL-expression.
```

Literatur

[TheBr95] Thees, J. und Bredereke, J. Ein Werkzeug zur Analyse von Feature–Interaktionen in IN. In: Gotzhein, R. und Bredereke, J. (Hrsg)., „5. GI/ITG–Fachgespräch 'Formale Beschreibungstechniken für verteilte Systeme' “, S. 199–208, Universität Kaiserslautern, Fachbereich Informatik (22. – 23. Juni 1995).

[The95] Thees, J. Ein Werkzeug zur Analyse von Feature–Interaktionen in ESTELLE–Spezifikationen. Diplomarbeit, Universität Kaiserslautern, FB Informatik, 1995.

[ISO89] Canadian Standards Association; Final Text of DIS9074. Information Processing Systems – Open Systems Interconnection – Estelle: A Formal Desciption Technique Based on an Extended State Transition Model; ISSN 0317-5669 (1989)

[Tur93] Kenneth Turner (Ed.) – Using Formal Description Techniques – An Introduction to Estelle, LOTOS and SDL. Wiley 1993

[Sij91] R. Sijelmassi and B. Strausser The Portable Estelle Translator: an overview and user guide. Tec