Institut für Automatisierungstechnik
der
Otto-von-Guericke-Universität Magdeburg

Implementationsunterstützende Werkzeuge für Kommunikationsprotokolle im Echtzeitbereich

Abschlußbericht zum Landesprojekt
FKZ: 1457A/0083

Projektleiter: Dr.-Ing. Peter Eichelbaum

Bearbeiter: Luis Urbina
Thorsten Szczepanski
Thomas Hadlich

März 1997

Inhaltsverzeichnis

1. Einführung in die Thematik 1

1.1. Allgemeines 1
1.2. Vorstellung des entwickelten Ansatzes 1
1.3. Aufbau des Berichtes 2

2. Vorstellung der verwendeten Beschreibungssprachen 4

2.1. Formale Beschreibungssprache Estelle 4
2.2. ANSI C 7

3. Transformationsregeln 10

3.1. Templates 10
3.2. p2c Transformationsregeln 13

4. Realisierung des Tools 23

4.1. Übersicht 23
4.2. PETDINGO 27
4.3. p2c (Generator) 29
4.3.1. Komponentenfiles 29
4.3.2. Ausgabe 30
4.3.3. Generierung 35

5. Testung 41

6. Weiterführende Arbeiten 45

7. Zusammenfassung 48

Literaturverzeichnis 49

Verzeichnis der Abkürzungen

ANSI	- American National Standard Institute
CCITT	- International Consultative Committee on Telegraphy an Telephony
DPM	- Dual Port Memory
EDT	- Estelle Development Tool
FDT	- formal description techniques
FIFO	- First In First Out
ISO	- International Organization for Stanardization
LOTOS	- Language of Temporal Ordering Specification
PETDINGO	- Portable Estelle Translator Distributed Implementation Generator
SDL	- Specification an Description Language

1. Einführung in die Thematik

1.1. Allgemeines

Die Dezentralisierung in der Automatisierungstechnik führt zu einer immer stärkeren Verwendung von Kommunikationssystemen auf diesem Gebiet.
Dabei steht sowohl der Echzeitaspekt, als auch die möglichst einfache Vernetzung verschiedener Komponenten im Vordergrund. Dazu stehen dem Anwender eine Vielzahl standardisierter Kommunikationsprotokolle zur Verfügung.
Neben dem informalen Teil enthalten diese Standards formale Beschreibungen des Protokolls (Zustandsgraph, Sequenz-Diagramme, Zustands-Tabellen). Um eine rechnergestützte softwaretechnische Umsetzung des Protokolls zu erreichen, ist eine Überführung der informalen Teile in eine formale Beschreibung unumgänglich. In der Telekommunikation werden formale Beschreibungstechniken (formal description techniques - FDT´s) für die Spezifikation von Protokollen und Diensten schon seit längerem genutzt.
Eingesetzt werden dabei hauptsächlich die standardisierten formalen Beschreibungssprachen LOTOS, Estelle und SDL [Hog89].
Für den Anwender solcher formaler Beschreibungstechniken ist es am Wichtigsten aus einer formalen Beschreibung möglichst einfach und schnell ein Programm zu erhalten, welches auf seinem konkret eingesetzten Rechner ablauffähig ist [Tur93], [Hog92].
Hier setzt das Projekt mit seiner Entwicklung an. Dabei ist das Ziel die automatische Implementierung, d.h., der Vorgang der Implementierung nach der formalen Beschreibung eines Kommunikationsprotokolls bis zum Code eines konkreten Zielprozessorsystems, soll so weit, wie möglich automatisch ablaufen. Voraussetzung für diese Vorgehensweise ist das Vorliegen einer vollständigen formalen Spezifikation eines Kommunikationsprotokolls [Eic93].
In diesem Bericht wird der Weg einer solchen automatischen Implementierung aufgezeigt. Hierzu wurde ein entsprechendes Werkzeug entwickelt, das sich streng an die Forderungen der Echtzeitkommunikation hält, einen möglichst effizienten Code zu erzeugen. Mit dem Einsatz formaler Beschreibungstechniken im Software-Entwurfsprozeß werden die Softwareentwickler in die Lage versetzt, neben einer eindeutigen Beschreibung der Anforderungen schnell Softwareprototypen zu erstellen und zu testen.

1.2. Vorstellung des entwickelten Ansatzes

Bisher wurden im Rahmen der Implementierung von Kommunikationsprotokollen 2 Wege beschritten. Zum einen wurde direkt aus der formalen Spezifikation innerhalb einer Norm ein entsprechendes C - Programm für eine Mikrocontroller erstellt. Diese Programme ließen sich aber schlecht testen und korrigieren und waren mit hohem Programmieraufwand verbunden.
Durch die Anwendung der formalen Spezifikation konnte zwar bisher eine sehr gut strukturierte und prüfbare/änderbare Ausgangsnotation gefunden werden. Bisherige Tools lieferten aber nur einen C - Code für UNIX - Maschinen. Die Anpassung und Transferrierung auf konkrete Mikrocontroller war wiederum nur mit sehr hohem Programmieraufwand und genauesten Kenntnissen der Rückwirkungen möglich. Der entwickelte Ansatz will nun all diese zeitaufwendigen und mit Fehlermöglichkeiten behafteten Teilschritte minimieren und für den Anwender unsichtbar machen. Dazu waren 2 Ausgangspunkte zu betrachten. Entweder die Fassung der bisher von Hand

durchgeführten Arbeiten in Filterprogramme für die automatische Abarbeitung oder die Realisierung eines geschlossenen Compiler-Werkzeuges.
Der Weg über Filterprogramme wurde nicht gewählt, da der Anwender hier immer noch eine Vielzahl an Auswahlmöglichkeiten hätte berücksichtigen müssen. Die Untersuchungen führten zum Weg der Realisierung eines Compilers.
Ausgehend von der Zielstellung der Realisierung eines Compilers wurde untersucht, welche Tools zur Generierung einer Zwischenabbildung zur Verfügung stehen.
Dazu wurden folgende Public-Domain-Compiler untersucht EDT [EDT94], PETDINGO [PET91] und estl [NIS87].
Untersucht wurden folgende Punkte:

- Ist das Tool frei verfügbar?
- Ist die Entwicklungsumgebung erweiterbar bzw. kompatibel?
- Wird eine abstrakte Zwischensprache unterstützt, so daß auf diese aufgesetzt werden kann?
- Ist die Zwischensprache bekannt und gut strukturiert?
- Wenn eine Zwischensprache verwendet wird, bietet das Tool vordefinierte Funktionen, die die Entwicklung eines Generators, d.h., eines Tools, das die Zwischensprache in die Zielsprache übersetzt, vereinfachen?

In diesem Zusammenhang wurden die Quelldateien bzw. die Literatur zu den jeweiligen Tools analysiert bzw. durchgearbeitet.
EDT ist nicht geeignet, da es nicht als Public-Domain-Software verfügbar ist und damit die Nutzung der Ergebnisse des Projekts erschweren würde.
estl implementiert Estelle nicht vollständig und unterstützt außerdem nicht die Erzeugung einer abstrakten Zwischensprache.
Nur für das Tool PETDINGO konnten die zu untersuchenden Punkte positiv bewertet werden.
PETDINGO ist ein Compiler, der aus dem Translator PET und dem Generator DINGO besteht.
Dieser Portable Estelle Translator (PET) liefert als Zwischensignal innerhalb des PETDINGO-Compilers eine objektorientierte Schnittstelle, auf die aufgesetzt werden konnte. Er führt die Funktionen scannen, parsen und eine semantische Prüfung einer Estelle Spezifikation durch. Dadurch sind Fehlermöglichkeiten bei der Interpretierung der formalen Spezifikation nahezu ausgeschlossen.
Darauf aufbauend wurde ein Generator P2C entwickelt, der aus dem objektorientierten Zwischencode ANSI C - Code erzeugt. Die Kombination der Elemente Translator PET und Generator P2C ergibt den Compiler E2C. Gleichzeitig erfolgte eine Prüfung der Realisierbarkeit von Estelle-Konstrukten bezüglich der Realisierbarkeit auf Mikrocontrollern und es wurden demgemäße Einschränkungen bei der Notation in Estelle herausgearbeitet, d.h. die Einschränkungen können gleich bei der Spezifikation in Estelle berücksichtigt werden und müssen nicht erst im C - Code berücksichtigt werden, wo die Auswirkungen auf die Gesamtspezifikation schlecht abzusehen sind.

1.3. Aufbau des Berichtes

Der Bericht ist wie folgt gegliedert: Das nächste Kapitel gibt eine kurze Einführung in die verwendeten Beschreibungssprachen.

Kapitel 3. stellt die Transformationsregeln zur Realisierung des Compilers dar. Sie spezifizieren die Algorithmen der Umsetzung von der Estelle Notation in den ANSI C - Code und sind damit der Ausgangspunkt der Compilerentwicklung.

Das 4. Kapitel erläutert dann die praktische Realisierung des Tools. Hier werden die Bestandteile des Compilers erläutert und die Struktur des p2c (Generator) verdeutlicht. Als Gesamtlösung ergibt sich dabei der Compiler e2c.

Das anschließende 5. Kapitel zeigt eine kurze Testanwendung zur Verdeutlichung der Übersetzungsprinzipien.

Im 6. Kapitel werden Aussagen zur Übertragung der gewonnenen Erfahrungen auf andere formale Beschreibungssprachen getroffen.

Schließlich gibt das 7. Kapitel eine Zusammenfassung der vorgestellten Lösungsstrategie und gibt Ausblicke auf weitere Arbeiten auf diesem Gebiet.

2. Vorstellung der verwendeten Beschreibungssprachen

2.1. Formale Beschreibungssprache Estelle

Eine anspruchsvolle Aufgabe bei der Entwicklung von Systemen ist die exakte Dokumentation des Systemverhaltens. Dazu sind spezielle Beschreibungsmittel notwendig. Die natürliche Sprache ist für diesen Zweck meist ungeeignet, da umgangssprachliche Dokumentationen nicht immer eindeutig interpretierbar und schwer auf Korrektheit und innere Konsistenz überprüfbar sind.
Besonders bei verteilten Systemen, wie z.B. auf dem Gebiet der Kommunikations- und Prozeßleittechnik sind Probleme durch nicht eindeutige, nicht exakte Beschreibungen gravierend, da verteilte Systeme oft auch an verteilten Orten von verschiedenen Personen entwickelt werden. Aus diesem Grunde wurden von der internationalen Normungsorganisation CCITT und ISO spezielle Sprachen zur Beschreibung von verteilten Systemen geschaffen. Diese formalen Beschreibungssprachen sind Estelle, LOTOS und SDL [Hog89].

Das Konzept von Estelle

Estelle basiert auf dem Modell von erweiterten endlichen Automaten. Die Notation dieser Spezifikationssprache ist von ISO-Pascal abgeleitet. Besonders die Sprachelemente für Definition, Deklaration und Manipulation von Daten, sowie die Kontrollstrukturen sind mit der Programmiersprache Pascal identisch. Für die Kommunikationsmechanismen werden zusätzliche, an Pascal angelehnte Sprachkonstrukte eingeführt.
Die erweiterten endlichen Automaten werden in Estelle als Module bezeichnet und können über Kanäle miteinander kommunizieren. Zu diesem Zweck besitzen die Module Interaktionspunkte, denen ein FIFO-Puffer zugeordnet ist.
Die Kommunikation findet daher asynchron statt. Durch Attribute ist angebbar, ob mehrere Interaktionspunkte den gleichen Puffer benutzen sollen. Zu jedem Interaktionspunkt gehört eine Anzahl von Interaktionen, welche die Ein- und Ausgaben der beteiligten Module darstellen. In der Kanaldefinition wird u.a. festgelegt, welche Rolle jedes beteiligte Modul bei der Kommunikation einnimmt. Durch die Rollendefinition ist für die an der Kommunikation über einen Kommunikationspunkt beteiligten Module erkennbar, welche Interaktion empfangen oder gesendet wird.
Zustandsübergänge eines erweiterten endlichen Automaten werden in Estelle als Transitionen bezeichnet. Die from - Klausel gibt den Anfangszustand an, die to - Klausel den Folgezustand bei Ausführung der Transition. In einer when - Klausel wird diejenige Eingabe - Interaktion aufgeführt, die eine Transition auslösen kann. Ob die Transition tatsächlich ausgelöst wird, kann zusätzlich durch eine provided-Klausel bestimmt werden. Im Zusammenhang mit einer when - Klausel werden durch die provided - Klausel die Werte der Interaktions - Parameter ausgewertet, ansonsten die Werte der Kontext - Variablen des erweiterten endlichen Automaten. Das zeitliche Verhalten von Modulen, etwa die Verzögerung einer Transition, kann durch die Angabe einer Zeiteinheit mittels delay - Klausel spezifiziert werden.
Aktivitäten, wie eine Ausgabe oder die Manipulation von Variablen, die während der Ausführung einer Transition durchgeführt werden sollen, befinden sich im Transitionsblock des Moduls.
Die Module einer Estelle - Spezifikation sind hierarchisch angeordnet, es existieren daher Vater-Sohn-Beziehungen zwischen den Modulen. Dies ist ein wesentlicher Aspekt der Estelle Beschreibungssprache.

In Estelle ist zwischen Moduldefinition und Modulinstanz zu unterscheiden. In einer Moduldefinition erfolgt die Spezifikation des Moduls, vergleichbar mit der Definition eines Datentyps. Sie besteht aus einem Modulkopf, in dem die Schnittstelle beschrieben wird, und einem Modulrumpf, in dem die Beschreibung des eigentlichen Modulverhaltens erfolgt. Eine Modulinstanz entspricht der Deklaration einer Variablen dieses Typs. Dabei können, genau wie bei Variablen, von einer Moduldefinition mehrere Modulinstanzen existieren. Durch Attribute kann bestimmt werden, ob Sohnmodule sequentiell oder parallel abgearbeitet werden.
Die hierarchische Anordnung, also die Baumstruktur, muß bei der Interpretation der Spezifikation nicht statisch, sondern kann durch Erzeugen und Beenden von Modulinstanzen dynamisch sein.

Einschränkungen des Sprachumfanges von Estelle

Um eine Umsetzung einer Estelle Spezifikation auf konkrete Prozessorsysteme vornehmen zu können, machen sich Einschränkungen notwendig. Diese werden mit Hilfe von Spezifikationen [ifa96], den bei ihrer manuellen Umsetzung gemachten Erfahrungen und der Struktur des Compilers getroffen.

Spezifikation

Eine Estelle-Spezifikation kann höchsten als eine Spezifikation mit der Systemklasse systemactivity definiert werden. Wenn keine Systemklasse festgelegt wird, müssen alle eingeschlossenen Prozesse, sobald Prozesse spezifiziert werden, mit dem Attribut activity deklariert werden. Bei einer nichtleeren Transitionsliste in der Estelle-Spezifikation muß von der Semantik her die Systemklasse festgelegt werden. Die Standard-Warteschlangendisziplin muß als individual queue definiert werden.
Eine Zeitskala kann festgelegt werden, sie bleibt aber beim Generierungsprozeß unberücksichtigt, da alles, was die Zeit betrifft vom Laufzeitsystem realisiert wird.
Es wird eine Zeitskala von einer Sekunde angenommen.

Deklarationsteil

Es gibt eine Reihe von Typen und Konstanten, die standardmäßig im Laufzeitsystem definiert und implementiert sind. Diese Typen sind u.a., die standardisierten Pascal-Datentypen und solche, die oft in PROFIBUS-Protokoll-Spezifikationen verwendet werden. Um eine gültige Estelle-Spezifikation zu formulieren, müssen sie aber, wenn sie in der Spezifikation verwendet werden sollen, im Deklarationsteil der Spezifikation definiert werden. Die Definition und Implementierungen solcher Konstanten befinden sich z.B. in den Dateien pb2.h, memdpr.h und runtime.h . Die Realisierung einiger Typen ist in *dpr.h zu finden.

Kanaldefinition

Vom Laufzeitsystem ergibt sich eine Einschränkung für die Typen und die Parameter von Signalen innerhalb einer Kanaldefinition. Sie besitzen eine feste Struktur. Es kann höchstens ein Typ und ein Parameternamen verwendet werden. Der Typ ist DataBuffer und der Parametername data_ptr. In Estelle ist DataBuffer ein Feld bestehend aus 256 Elementen mit dem Datentyp Byte BYTE. Aufgrund der unterschiedlichen Schreibweise vom Pascal-Typ Byte und BYTE im Laufzeitsystem muß BYTE ebenfalls neu definiert werden.

```
DataBuffer=array[0.. 255] of BYTE;
```

Ein Signal signal-name in einer Kanaldefinition kann z.B. wie folgt aussehen:

```
signal-name (data_ptr : DataBuffer);
```

Interaktionspunkt

Bei der Deklaration von Interaktionspunkten ist zu beachten, daß wenn eine Warteschlangendisziplin angegeben wird, dann muß diese eine individual queue sein. Bei Nichtangabe einer Warteschlangendisziplin wird die Standard-Warteschlangendisziplin der Spezifikation benutzt. Deshalb empfiehlt es sich bei Interaktionspunktdeklarationen keine Angabe der Warteschlangendisziplin zu machen.

Modulkopf

Ein Modulkopf kann wahlweise mit einer Modulklasse versehen werden. Diese kann nur systemactivity oder activity sein. Eine Eingabe eines Export-Teils in einer Modulkopfdefinition ist nicht erlaubt. Demzufolge ist die Deklaration von zu exportierenden Variablen unzulässig.

Deklaration von Transitionen

Priority- und Delay-Klausel

Bei der Definition einer Transition sind Priority- und Delay-Klauseln nicht erlaubt, da alle Funktionen, die mit Zeiteinschränkungen zu tun haben, im Laufzeitsystem realisiert sind oder extra vom Spezifizierer in C zu implementieren zu sind.
Priority-Klauseln sind nicht erlaubt, da das Laufzeitsystem dies nicht unterstützt.

When- und Provided-Klausel

Da Signale höchstens einen Parameter haben dürfen, mit dem Informationen übertragen werden, nämlich data_ptr, muß der Spezifizierer dies bei jedem Zugriff auf die Signale bzw. den Signalparameter berücksichtigen. So werden Vergleiche zwischen den Werten von data_ptr und anderen Werten innerhalb einer Provided-Klausel in einer einfachen und stabilen Form gemacht.
Gleiches gilt für die When-Klausel.

Anweisungen

Init-Anweisung

Die Erzeugung von Modulen mittels der Init-Anweisung ist ausschließlich in dem Initialisierungsteil eines Moduls erlaubt.
Somit können Module nicht dynamisch erzeugt werden.

Release- und Terminierung von Modulen

Sie werden nicht unterstützt, da Module solange leben sollen, bis das gesamte System terminiert wird. Nach der Terminierung des Systems werden im Laufzeitsystem Funktionen aufgerufen, die u.a. die Freigabe von Speicherplatz bewirken.

Disconnect- und Detach-Operation

Diese Anweisungen werden aus dem im obigen Abschnitt genannten Grund nicht gebraucht.

Output-Anweisung

Realisiert mit der Runtime Library.

2.2. ANSI C

ANSI C stellt eine Weiterentwicklung der bekannten Programmiersprache C dar. Grundgedanke der Weiterentwicklung ist die Standardisierung der Funktionen, Festlegungen zum Compiler und der Erzeugung von ablauffähigen Programmen für unterschiedliche Plattformen. Dabei ist besonders die Möglichkeit der Erzeugung eines ablauffähigen Programmcodes für unterschiedliche Plattformen bis hin zum Mikrocontroller für die Lösung des Forschungsproblems relevant.

Die ANSI C Norm unterscheidet zwischen zwei Rechner - Umgebungen (data-processing-system environments) [Her89]:

- Übersetzungs - Umgebung (translation environment)
- Ausführungs - Umgebung (execution environment).

Deren Eigenschaften definieren und schreiben die Ergebnisse von konformen C - Programmen entsprechend der syntaktischen und semantischen Regeln für konforme Implementationen vor.

Übersetzungs - Umgebung

Programmstruktur

C - Programme müssen nicht als ein Ganzes zur selben Zeit übersetzt werden. Die getrennten Übersetzungs - Einheiten eines Programms können über 3 Wege miteinander kommunizieren:

- Aufruf von Funktionen mit externer Bindefähigkeit,
- Zugriff auf Objekte mit externer Bindefähigkeit,
- Manipulation von Daten - Dateien.

Solche Übersetzungs - Einheiten können getrennt übersetzt werden und dann später (mit Hilfe eines Linkers gebunden werden, um ein ausführbares Programm zu erhalten.

Übersetzungsphase

Bei der Übersetzung sind die folgenden Phasen in der vorgegebenen Reihenfolge durchzuführen.

1. Umwandlung aller physikalischer Zeichen in der Quelldatei in ihr entsprechendes Zeichen aus der Quell - Zeichenmenge.

2. Jedes Vorkommen eines Neue-Zeile-Zeichens mit einem unmittelbar voranstehenden Fortsetzungszeichen \ wird gelöscht, um die physikalischen Quellzeichen „zusammenzukleben“, um so die logischen Quellzeichen zu erhalten.

3. Die Quelldatei wird in Präprozessor-Token und Folgen von Zwischenraumzeichen zerlegt.

4. Präprozessor-Direktiven werden ausgeführt und Makroaufrufe werden ausgelöst.

5. Jedes Zeichen aus der Quell-Zeichenmenge und Escape-Sequenzen in Zeichen-Konstanten in d in String-Literale werden in Zeichen der Ausführungs-Zeichenmenge konvertiert.

6. Hintereinander angegebene String-Literale werden zusammengehängt.

7. Zwischenraum-Zeichen, welche Token trennen, sind nicht mehr signifikant. Präprozessor-Token werden in Token umgewandelt. Die resultierenden Token werden syntaktisch und semantisch analysiert und übersetzt.

8. Alle externen Objekt- und Funktions-Referenzen werden befriedigt. Bibliothekskomponenten sollten dazu gebunden werden, um externe Referenzen zu Funktionen und Objekten, welche nicht in der momentanen Übersetzung definiert sind, zu befriedigen. Das aus diesen Phasen resultierende Ergebnis ist dann ein sogenanntes program image, welches Informationen beinhaltet, die notwendig für seine Ausführung in der Ausführungs-Umgebung ist.

Ausführungsumgebung

Es werden 2 Ausführungs-Umgebungen definiert:

- alleinstehend (freestanding environment)
- mit Betriebssystem-Unterstützung (hosted environment).

In beiden Fällen wird der Programmstart als der Zeitpunkt definiert, zu dem eine bestimmte C-Funktion aus der Ausführungs-Umgebung aufgerufen wird. Alle Objekte im statischen Speicher müssen vor dem Programmstart initialisiert werden. Die Programmbeendigung gibt die Kontrolle an die Ausführungs-Umgebung zurück.

Alleinstehende Ausführungs-Umgebung

In dieser Umgebung sind der Name und der Typ der Funktion, welche beim Programmstart aufgerufen wird, implementationsdefiniert. Ansonsten gibt es keine reservierten externen Bezeichner. Umfang und Inhalt von Bibliotheken, welche einem frestanding Programm zur Verfügung stehen, sind implementationsdefiniert. Die Wirkungsweise einer Programmbeendigung ist in einer freestanding environment implementations definiert.

Umgebung mit Betriebssystem-Unterstützung

Die Umgebung muß folgende Bedingungen einhalten:

Programmstart

Die Funktion, welche zum Programmstart aufgerufen wird, heißt main. Die Implementation deklariert keinen Prototyp für diese Funktion.

Programmausführung

In einer Umgebung mit Betriebssytem-Unterstützung darf ein Programm alle Funktionen, Makros, Typ-Definitionen und Objekte verwenden, welche im Bibliotheks-Kapitel beschrieben sind.

Programmbeendigung

Eine Rückkehr vom ursprünglichen Aufruf der Funktion main ist äquivalent zum Aufruf der Funktion exit mit einem Argumentwert, welcher dem Rückgabewert von main entspricht. Wenn die Funktion main ein return ausführt, welches keinen Wert angibt, dann ist der Programmende-Statuswert, welcher zum Betriebssystem zurückgegeben wird, undefiniert.

3. Transformationsregeln

3.1. Templates

- Modulheader, Modulbody

Nach Zuordnung einer Modulheaderdeklaration zur Modulbodydeklaration wird im zugehörigen Headerfile eine Struktur vom Typ „Name des Body“maschine erzeugt. Sie enthält die innerhalb des Headers definierten Interaktionspunkte und exportierte Variablen. Zusätzlich werden die im Modulbody benötigten Variablen deklariert. Nicht einem Modulbody zugeordnete Modulheader werden ignoriert.

```
struct identifiermachine {
MACHINE_INFO _m;
[+{interaction-point-declaration};]
+{interaction-point-declaration}
}
```

Das Gesamtverhalten des Moduls ist in der folgenden Struktur beschrieben:

```
/* Modulstruktur */
struct FSM_Type Modufsm_t = {
/* F_NStates */ ModuNSTATE,                /* Anzahl der Zustaende                    */
/* F_NInputs */ ModuNINPUT,                /* Anzahl der Eingaben                     */
/* F_NQueues */ ModuNQUEUES,               /* Anzahl der Warteschlangen               */
/* F_NTrList */ ModuNTRLIST,               /* Anzahl der Transitionen                 */
/* F_TrTable */ (int **) Modutr,           /* Zeiger auf Transitionszuordnungstabelle */
/* F_TrList  */ Modutrlist,                /* Zeiger auf Transitionsliste             */
/* F_TrFuncs */ Moduactfunc,               /* Zeiger auf act-Funktionspointerliste    */
/* F_PrFuncs */ Moduprdfunc,               /* Zeiger auf prd-Funktionspointerliste    */
/* F_IPInit  */ (IPINIT_FUNCPTR) ModuIPInit,/* Zeiger auf Init-Funktion fuer IPs      */
/* F_Init    */ (INIT_FUNCPTR) ModuInit,   /* Zeiger auf Init-Funktion fuer Modul     */
/* F_Size    */ sizeof(struct Modumachine),/* Groesse der Modulinstanz in Bytes       */
/* F_Nips    */ ModuNIPS                   /* Anzahl der Interaktionspunkte           */
};
```

- Modulvariablen

Die Deklaration der Modulvariablen in Estelle führt zu einer Deklaration der Modulvariable in der Namemaschine-Struktur des zugehörigen Vatermoduls. Der Typ der Modulvariable entspricht dabei der NameMaschine-Struktur des zugehörigen Modulkörpers.

- Initialisierung

Jedes Estellemodule enthält eine Initialisierungstransition, diese Transition wird für jedes Modul in der Funktion:

```
BOOL NamInit(struct Nammachine * machp, struct FSM_InitData * fsmInitData)
{
 machp->_m.STATE = 0;
 {
   if ((machp->Nam1 = (struct Nam1machine *) FSM_Init ((struct FSM *)
machp, Nam1, fsmInitData)) == NULL)
     return FALSE;
   if ((machp->Nam21 = (struct Nam2machine *) FSM_Init ((struct FSM *)
machp, Nam2, fsmInitData)) == NULL)
     return FALSE;
   if ((machp->Nam22 = (struct Nam2machine *) FSM_Init ((struct FSM *)
machp, Nam2, fsmInitData)) == NULL)
     return FALSE;
```

```
   FSM_Connect( machp->Nam1->U1IP, machp->Nam21->IPN);
   FSM_Connect( machp->Nam1->U2IP, machp->Nam22->IPN);
   ;
 }
 return TRUE;
}
```

zusammengefaßt.
Zusätzlich wird die folgende Funktion

```
void NamIPInit(struct Nammachine *machp, struct FSM_IP *ip) {
  ip->IP_Queue   = 0;
  ip->IP_DestIP  = NULL;
  ip->IP_Machine = (struct FSM *) machp;
  ip->IP_InOff   = 0;
  machp->U1IP = ip++;

  ip->IP_Queue   = 1;
  ip->IP_DestIP  = NULL;
  ip->IP_Machine = (struct FSM *) machp;
  ip->IP_InOff   = 1;
  machp->U2IP = ip;
}
```

zur Initialisierung der Interaktionspunkte erzeugt.

- from, to,when-clause

Die innerhalb eines Moduls möglichen Transitionen sind in 2 Tabellen gespeichert. Zur Laufzeit erfolgt die Auswahl der auszuführenden Transition durch die runtime-Bibliothek entsprechend dem Zustand des Moduls und den anliegenden Interaktionen.

- Provided-clause

Provided Transitionen werden innerhalb der Implementierung durch Funktionen representiert, die als Rückgabewert den Wert des in der Original clause enthaltenen Boolschen Ausdrucks enthalten. Die Entscheidung zur Ausführung der Transition wird von der runtime-Bibliothek aufgrund dieses Ausdrucks getroffen. Bei Fehlen einer Provided clause wird „boolean-expression“ zu TRUE gesetzt.

```
/* Nam2ided-Funktion ModuprdNRR */
void ModuprdNRR(struct Modumachine * machp, int queuenb)
 return boolean-expression;
}
```

- transition-block

Ein Estelle Transitionblock wird in der Implementierung zu einer sogenannten Action-Funktion zusammengefaßt. Der Aufbau dieser Funktion kann dem folgenden Template entnommen werden.

```
/* Action-Funktion ModuactNRR */
void ModuactNRR(struct Modumachine * machp, int queuenb)
{
EVP inEvent_ptr = FSM_QFirstRemove((FSMP) machp, queuenb)
compound-statement
}
```

Die Auswahl der Transitionen erfolgt durch die runtime-Bibliothek mit Hilfe der Transitions- und der Transitionszuordnungstabelle. Die Transitionstabelle enthält die

Menge der theoretisch möglichen Zustands-/Eingangswertpaare. Jeder Eintrag in dieser Liste verweist auf einen Eintrag in der Transitionszuordnungsliste. Durch diese wird auf die aktuell gültigen Action und Provided Funktionen verwiesen.

```
/* Transitionsliste */
int Nam1tr[Nam1NSTATE][Nam1NINPUT+1] = {
{ /* STATE START */
/* 0 : START X SPONTANEOUS          */  1,
/* 1 : START X DATAresponse         */  0,
/* 2 : START X DATAresponse         */  0,
}, /* STATE START */

{ /* STATE READY */
/* 0 : READY X SPONTANEOUS          */  0,
/* 1 : READY X DATAresponse         */  2,
/* 2 : READY X DATAresponse         */  3,
}, /* STATE READY */

};
```

```
/* Transitionszuordnungstabelle */
struct FSM_Trans Modutrlist[ModuNTRLIST+1] = {
/*-------------------------------------------------------------------*/
/* ENTRY #  NEXT        TRANS       RESULT          */
/*-------------------------------------------------------------------*/
/* 0 */ {   0,          0,          0                 },
/* 1 */ {   0,          1,          READY             },
/* 2 */ {   0,          2,          SAME_STATE        },
/* 3 */ {   0,          3,          SAME_STATE        },
};
```

3.2. p2c Transformationsregeln

Die folgende Zusammenstellung gibt eine Übersicht über die wichtigsten in p2c implementierten Transformationsregeln.

specification		
	specification *IDENTIFIER [systemclass]* **;** *[default-options]* *[time-options]* *body-definition* **end.**	keine direkte Übersetzung *[default-options]* *[time-options]* *body-definition*
default-options		
	default *queue-discipline* **;**	
queue-discipline		
	common queue	
	individual queue	
time-options		
	timescale *IDENTIFIER*	
body-definition		
	declaration-part *initialization-part* *transition-declaration-part*	*declaration-part* *initialization-part* *transition-declaration-part*
declaration-part		
	{declaraions}	*{declarations}*
declarations		
	CONSTANT-DEFINITION-PART	*CONSTANT-DEFINITION-PART*
	TYPE-DEFINITION-PART	*TYPE-DEFINITION-PART*
	channel-defintion	*channel-defintion*
	module-header-definition	*module-header-definition*
	module-body-definition	*module-body-definition*
	interaction-point-declaration-part	*interaction-point-declaration-part*
	module-variable-declaration-part	*module-variable-declaration-part*
	VARIABLE-DECLARATION-PART	*VARIABLE-DECLARATION-PART*
	state-defintion-part	*state-defintion-part*
	state-set-definition-part	*state-set-definition-part*
	PROCEDURE-AND-FUNCTION-DECLARATION-PART	*PROCEDURE-AND-FUNCTION-DECLARATION-PART*
constant-defintion-part		
	*[***const** *constant-definition* **;** *{ constant-definition* **;***}]*	*[constant-definition* **;** *{ constant-definition* **;***}]*
constant-definition1 (constanttype = ordinal | real)		
	identifier = constant	**const** (**short**|**int**|**float**) *identifier = constant*
constant-definition2 (constanttype = string)		
	identifier = constant	**static __String** *identifier = constant*
constant-definition3 (else)		
	identifier = constant	**static** (*type*|***type*) *identifier = constant*
type-definition-part		
	type *type-defintion* **;** {*type-defintion* **;**}]	*type-defintion* {*type-defintion* }
type-definition		
	identifier = type-denoter	***type-type-denoter*** *identifier* **;**
type-type-denoter		
	type-identifier	**typedef** *type-identifier*

	new-type	*new-type*
type-identifier		
	type-identifier	*identifier*
new-type		
	new-ordinal-type	*new-ordinal-type*
	new-structured-type	*new-structured-type*
	new-pointer-type	*new-pointer-type*
new-ordinal-type		
	enumerated-type	*enumerated-type*
	subrange-type	*subrange-type*
enumerated-type		
	(*identifier1***,***identifier2***,** *identifiern***)**	**const short** *identifier1* **1;** **const short** *identifier2* **2;** **const short** *identifiern* **n;** **typedef short**
subrange-type		
	*constant***..***constant*	**typedef** (**short**|**int**)
new-structured-type		
	unpacked-structure-type	*unpacked-structure-type*
unpacked-structure-type		
	array-type	*array-type*
	record-type	*record-type*
	set-type	
array-type		
	array [*index-type {***,** *index-type}***] of** *component-type*	**typedef** (**short** |**int**) *component-type*
component-type		
	type-denoter	*type-denoter*
record-type		
	record *field-list* **end**	**typedef struct** *identifier* **{***field-list***}**
field-list		
	[(*fixed-part*	*fixed-part* **;**
	[**;** *variant-part*]|	
	[*variant-part*)[**;**]]	
fixed-part		
	*record-section {***;** *record-section}*	*record-section*
record-section		
	identifier1, identifier2, identifiern **:**	*type-denoter identfier1***;**
	type-denoter	*type-denoter identfier2***;**
		*type-denoter identfiern***;**
new-pointer-type		
	↑ *domain-type*	**typedef *** *domain-type*
channel-definition		
	channel-heading channel-block	keine direkte Übersetzung
module-header-definition		
	module *identifier [class] [***(***parameter-list***)***]* [**ip** *+{interaction-point-declaration}***;**] [**export** *+{export-variable-declaration}***;**] **end ;**	keine direkte Übersetzung [*+{interaction-point-declaration}*] [*+{export-variable-declaration}*]
interaction-point-declaration		
	(identfier1, identifier2, .. **:** *interaction-point-type)*	**struct FMS_IP ***id1**;** **struct FMS_IP ***id2**;** **..**
	(identfier1, identifier2, .. **: array [***index-type-*	

	list] **of** *interaction-point-type)*	
interaction-point-type		
	channel-identifier **(** *role-identifier* **)** *[queue-discipline]*	keine direkte Übersetzung
export-variable-declaration		
	identifier1	*type-denoter* : ***identifier1*;
	[**,***identifier2*	*type-denoter* : ***identifier2*;
	,*identifiern]: type-denoter*	*type-denoter* : ***identifiern*;
type-denoter		
	type-identifier	*type-identifier*
	new-type	*new-type*
interaction-point-declaration-part		
	[**ip** *+{interaction-point-declaration}***;**]	[*+{interaction-point-declaration}*]
module-variable-declaration-part		
	identifier-list : header-identifier	keine direkte Übersetzung
	identifier-list : array **[***index-type-list***] of** *header-identifier*	
variable-declaration-part		
	[**var** *variable-declaration*;	[*variable-declaration*;
	{*variable-declaration* **;**}]	{*variable-declaration* **;**}]
variable-declaration		
	identifier1 [**,***identifier2* **,***identifiern]: type-denoter*	*type-denoter* : *identifier1*; *type-denoter* : *identifier2*; *type-denoter* : *identifiern*;
state-definition-part		
	state *state-identifier-list* ;	***state-identifier-list***
state-identifier-list		
	identifier1 [**,***identifier2* **,** *identifiern]*	**#define** *identifier1 1***;** *[***#define** *identifier2 2***;** **#define** *identifiern n***;***]*
state-set-definition-part		
	statese t *+{state-set-definition***;***}*	keine direkte Übersetzung
procedure-declaration-and-function-part		
	{*procedure-declaration* **;**)	*procedure-declaration*
	{*function-declaration* **;**)	*function-declaration*
procedure-declaration		
	procedure-heading **;** *directive*	*directive procedure-heading* **;**
	procedure-identification **;** *procedure-block*	*procedure-identification* **;** *procedure-block*
	procedure-heading **;** *procedure-block*	*procedure-heading* **;** *procedure-block*
procedure-heading		
	procedure *identifier [formal-parameter-list]*	**void** *identifier formal-parameter-list*
formal-parameter-list		
	(*formal-parameter-section***;** *{formal-parameter-section}***)**	**(***formal-parameter-section {***,** *formal-parameter-section}***)**
formal-parameter-section		
	value-parameter-specification	*value-parameter-specification*
	variable-parameter-specification	*variable-parameter-specification*
	procedural- parameter-specification	
	functional-parameter-specification	
value-parameter-specification		
	identifier1 *[***,***identifier2* **,** *identifiern]* **:***type-identifier*	**const** *type-identifier identifier1***;** **const** *type-identifier identifier2***;** **const** *type-identifier identifiern***;**

variable-parameter-specification		
	var *identifier1* *[,identifier2* ***,** identifiern]* **:***type-identifier*	*type-identifier*** identifier1**;*** *[type-identifier*** identifier2**;*** *type-identifier*** identifiern**;]***
directive		
	primitive	**{/* Funktion muss vom Spezifizierer implementiert werden */}**
	external	**extern;**
	forward	
	letter {letter | digit}	
procedure-identification		
	procedure *procedure-identifier*	**void** *procedure-identifier* **();**
procedure-block		
	procedure-block	*block*
block		
	label-declaration-part *constant-definition-part* *type-definition-part* *variable-declaration-part* *procedure-and-function-declaration-part* *statement-part*	 *constant-definition-part* *type-definition-part* *variable-declaration-part* *procedure-and-function-declaration-part* *block-statement-part*
block-statement-part		
	begin *statement-sequence* **end;**	**{** *statement-sequence* **}**
function-declaration		
	function-heading **;** *directive*	*function-heading* **;** *directive*
	function-identification **;** *function-block*	
	function-heading **;** *function-block*	*function-heading* **;** *function-block*
function-heading		
	function *identifier [formal-parameter-list]***:***result-type*	*result-type identifier formal-parameter-list*
function-block		
	label-declaration-part	
	constant-definition-part	*constant-definition-part*
	type-definition-part	*type-definition-part*
	variable-declaration-part	*variable-declaration-part*
	procedure-and-function-declaration-part	*procedure-and-function-declaration-part*
	function-statement-part	*function-statement-part*
function-statement-part		
	begin *statement-sequence* **end;**	**{** *type-denoter* **__fRet;** **{** *statement-sequence* **}** **return __fRet;** **}**
initialization-part		
	{**initialize** *transition-group*}	*transition-group*
transition-group		
	*+{clause-group transition-block **;**}*	*+{clause-group transition-block }*
clause-group		
	[Nam2ided-clause] *[from-clause]* *[to-clause]*	keine direkte Übersetzung keine direkte Übersetzung keine direkte Übersetzung

	[any-clause]	keine direkte Übersetzung
	[delay-clause]	
	[when-clause]	keine direkte Übersetzung
	[priority-clause]	
transition-block		
	CONSTANT-DEFINITION-PART	*CONSTANT-DEFINITION-PART*
	TYPE-DEFINITION-PART	*TYPE-DEFINITION-PART*
	VARIABLE-DECLARATION-PART	*VARIABLE-DECLARATION-PART*
	PROCEDURE-AND-FUNCTION-DECLARATION-PART	*PROCEDURE-AND-FUNCTION-DECLARATION-PART*
	[transition-name] STATEMENT-PART	*[transition-name] transition-statement-part*
transisition-statement-part		
	compound-statement	
compound-statement		
	begin *statement-sequence* **end**	*statement-sequence*
statement-sequence		
	{ *statement* {**;** *statement* }	{ *statement* {**;** *statement* }
statement		
	[label :] simple-statement	*[label :] simple-statement*
	[label :] structured-statement	*[label :] structured-statement*
simple-statement		
	empty-statement	*empty-statement*
	assign-statement	*assign-statement*
	procedure-statement	*procedure-statement*
	goto-statement	
	attach-statement	*attach-statement*
	connect-statement	*connect-statement*
	detach-statement	
	disconnect-statement	
	init-statement	*init-statement*
	output-statement	*output-statement*
	release-statement	
	terminate-statement	
empty-statement		
		\n
assign-statement		
	variable-access **:=** *expression*	*variable-access = expression*
	function-identifier **:=** *expression*	*function-identifier = expression*
	module-variable **:=** *module-variable*	
variable-access		
	entire-variable	*entire-variable*
	component-variable	*component-variable*
	identified-variable	*identified-variable*
entire-variable		
	variable-identifier	*variable-identifier*
variable-identifier		
	identifier	*identifier*
component-variable		
	indexed-variable	*indexed-variable*
	field-designator	
indexed-variable		
	array-variable **[** *index-expression* **]**	*array-variable* **[***index-expression***]**
index-expression		

	expression	*expression*
identified-variable		
	pointer-variable ↑	**(*** *pointer-variable***)**
pointer-variable		
	variable-access	*variable-access*
expression		
	simple-expression	**(***simple-expression*
	[relational-operator	*[relational-operator*
	simple-expression]	*simple-expression]***)**
simple-expression		
	[sign] term	*[sign] term*
	[adding-operator	*[adding-operator*
	term]	*term]*
sign		
	+ | -	+ | -
term		
	factor	*factor*
	{multiplying-operator	*{multiplying-operator*
	factor }	*factor }*
factor		
	variable-access	*variable-access*
	unsigned-constant	*unsigned-constant*
	function-designator	*function-designator*
	set-constructor	
	(*expression***)**	**(***expression***)**
	not *factor*	**(!** *factor***)**
unsigned-constant		
	unsigned-number	*unsigned-number*
	character-string	*character-string*
	constant-identifier	*constant-identifier*
	nil	**0**
unsigned-number		
	unsigned-integer	*unsigned-integer*
	unsigned-real	*unsigned-real*
unsigned-integer		
	digit-sequence	*digit-sequence*
digit-sequence		
	digit {digit}	*digit {digit}*
digit		
	0|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**	**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**
unsigned-real		
	*unsigned-integer***.***fractional-part [***e** *scale-factor]*	*unsigned-integer***.***fractional-part [***e** *scale-factor]*
	*unsigned-integer [***e** *scale-factor]*	*unsigned-integer [***e** *scale-factor]*
fractional-part		
	digit-sequence	*digit-sequence*
scale-factor		
	signed-integer	*signed-integer*
signed-integer		
	[sign] unsigned-integer	*[sign] unsigned-integer*
constant-identifier		
	identifier	*identifier*
function-designator		

	function-identifier [actual-parameter-list]	*function-identifier [actual-parameter-list]*
function-identifier		
	identifier	*identifier*
multiplying-operator		
	***** | / | **div** | **mod** |**and**	***** | / | / | (/|**%**) | **&&**
relational-operator		
	= | <> | < | > | <= | >=	== | **!**= | < | > | <= | >=
	in	**).includes(**
procedure-statement		
	procedure-identifier ([actual-parameter-list])	*procedure-identifier [actual-parameter-list]*
procedure-identifier		
	identifier	*identifier*
actual-parameter-list		
	(*actual-parameter {***,** *actual-parameter }***)**	**(***actual-parameter {***,** *actual-parameter }***)**
actual-parameter		
	expression	*expression*
	variable-access	*variable-access*
	procedure-identifier	*procedure-identifier*
	function-identifier	*function-identifier*
attach-statement		
	attach *external-ip* **to** *external-child-ip*	**FSM_Connect(machp->** *external-ip***, machp->** *external-child-ip* **);**
external-ip		
	interaction-point-reference	*interaction-point-reference*
interaction-point-reference		
	*interaction-point-identifier [***[***index-expression {***;** *index-expression}***]]***	*modul-variable-identifier->interaction-point-identifier[***[***index-expression {***;** *index-expression}***]]***
interaction-point-identifier		
	interaction-point-identifier	*identifier*
external-child-ip		
	external-child-ip	*external-ip*
connect-statement		
	connect *connect-ip* **to** *connect-ip*	**FSM_Connect(machp->** *connect-ip***, machp->** *connect-ip* **);**
connect-ip		
	child-external-ip	*child-external-ip*
	external-ip	*external-ip*
init-statement		
	init *modul-variable* **with** *body-identifier [***(***actual-module-parameter-list***)***]*	**if((machp->***modul-variable* **= struct** *body-identifier***machine *) FSM_Init((struct FSM *) machp,** *body-identifier***, fsmInitData)) == NULL) return FALSE;**
body-identifier		
	body-identifier	*identifier* (1.-4. buchstabe)
output-statement		
	output *interaction-reference [actual-parameter-list]*	**EVP outEvent_ptr = GetDataBuffer();FSM_IntraOutput(machp->***interaction-reference [actual-*

		*parameter-list]***);**
interaction-reference		
	*interaction-point-reference***.***interaction-identifier*	*interaction-point-reference***,** **outEvent_ptr,** *interaction-point-reference_interaction-identifier*
interaction-identifier		
	interaction-identifier	*identifier*
structured-statement		
	compound-statement	*compound-statement*
	conditional-statement	*conditional-statement*
	repetitive-statement	*repetitive-statement*
	with-statement	
	forone-statement	*forone-statement*
conditional-statement		
	if-statement	*if-statement*
	case-statement	*case-statement*
if-statement		
	if *boolean-expression* **then** *statement* *[else-part]*	**if (***boolean-expression***)** *statement* *[else-part]*
else-part		
	else *statement*	**else** *statement*
case-statement		
	case *case-index* **of** *case-list-element* [**;** *case-list-element*] [**;**] **end**	**switch (** *case-index* **) {** *case-list-element* [*case-list-element*] **}**
case-index		
	expression	*expression*
case-list-element		
	case-constant-list **:** *statement*	*case-constant-list statement* **break;**
case-constant-list		
	case-constant { , *case-constant }*	**case** *case-constant* **:**
case-constant		
	constant	constant
repetitive-statement		
	repeat-statement	*repeat-statement*
	while-statement	*while-statement*
	for-statement	*for-statement*
	all-statement	*all-statement*
repeat-statement		
	repeat *statement-sequence* **until** *boolean-expression*	**do {** *statement-sequence* **} while (!(** *boolean-expression* **))**
while-statement		
	while *boolean-expression* **do** *statement*	**while (** *boolean-expression* **)** *statement*
for-statement		
	for *control-variable* **:=** *initial-value* (**to** | **downto**) *final-value* **do** *statement*	**{ int __bound** = *final-value* **;** *control_variable = initial-value* **;** **while(** *control-variable* (<=|>=) **__bound)** **{** *statement control-variable* (++|--) **;}**
boolean-expression		
	expression	*expression*
control-variable		

	entire-variable	*entire-variable*
initial-value		
	expression	*expression*
final-value		
	expression	*expression*
all-statement		
	all (*identifier1, ..):Ordinal-type* **do** *statement*	{ *Ordinal-type identifier1*; **__IterList __list;** **if (__MI->iterate**(*identifier1*,__list, lowbound(*Ordinal-type*),upbound(*Ordinal-type*))) **do** **{** *statement* **} while (__MI->next(i,__list));** **}**
	all *identifier: Header-identifier* **do** *statement*	**{** _nr *identifier*; **__IterList __list;** **if (__MI->iterate**(*identifier***,__list))** **do {** *statement* **} while (__MI->next(***identifier***,__list));** **}**
forone-statement		
	forone (*identifier1, ..):Ordinal-type* **suchthat** *boolean-expression* **do** *statement* [**otherwise** *statement*]	**{** **int __done=0;** **{** *Ordinal-type identifier1*; **__IterList __list;** **if (__MI->iterate**(*identifier1*,__list, lowbound(*Ordinal-type*),upbound(*Ordinal-type*))) **do {** **if(__done=(machp->***boolean-expression*)) *statement* **} while (__MI->next(***identifier***, __list)&&!__done);** **}** **if (!__done)** *statement*; }
	forone *identifier:header-identifier* **suchthat** *boolean-expression* **do** *statement* [**otherwise** *statement*]	**{** **int __done=0;** **{** _nr *identifier*; **__IterList __list;** **if (__MI->iterate(***identifier***,__list))** **do {** **if(__done=(machp->***boolean-expression*)) *statement* **} while (__MI->next(***identifier***,__list)&&!__done);**

		} **if (!__done)** *statement*; }

4. Realisierung des Tools

4.1. Übersicht

Ausgehend von den gemachten Voruntersuchungen zur Bildung des Estelle-Subsets und der Auswahl eines Tools zur Realisierung des Frontends des Compilers, konnte die Gesamtstruktur des Tools zur automatisierten Erzeugung eines auf einen Zielprozessor ausgerichteten C-Codes aus einer formalen Spezifikation in Estelle abgeleitet werden [Hes92].
Dazu wird der **P**ortable **E**stelle **T**ranslator (PET) des Tools PETDINGO genutzt, um aus der Estelle-Notation eine objektorientierte Zwischensprache zu erzeugen, die durch den Generator P2C, der im Rahmen des Projekts unter Berücksichtigung der Randbedingungen der Estelle-Subsetbildung und des zu erzeugenden ANSI-C-Codes
realisiert wurde.

Compiler

Der Compiler für die Untersprache Estelle kann durch eine einfache Script-Datei realisiert werden. Wenn der Translator durch pet und der Generator durch p2c realisiert werden, so implementiert folgende triviale Script-Datei den Compiler e2c für Estelle.

```
pet -o "$@".o "$@".e &&
p2c "$@".o
```

Der Aufruf des Compilers erfolgt gemäß der Syntax:

```
e2c estelle-file-name [ .e ] ,
```

wobei estelle-file der Name eines einer Estelle-Spezifikation enthaltenden Files ist. Diese Files hat die Erweiterung ".e". Der Compiler e2c führt dann den Translator pet und den Generator p2c hintereinander, wobei der Generator erst dann ausgeführt wird, wenn der Translator erfolgreich beendet wurde.

Translator

Der PET-Translator wird in der Form aufgerufen, wie er in [PET91] beschrieben ist. Zur Realisierung des Compilers wird der „normale“ Aufruf von pet verwendet.

```
pet -o "object-file-name".o "estelle-file-name".e
```

Die "o" Option dient zur Erzeugung eines Ausgabefiles mit dem gleichen Namen wie das Eingabefile, aber mit der Erweiterung "o".

Der Translator meldet sich mit dem Text:

Portable Estelle Translator (PET)
Version 1.0
National Institute of Standards & Technology
Bldg 225, B141
Gaithersburg MD 20899, USA

>>Loading tables

0 syntax errors
0 semantic errors
0 lexical errors

Beim Auftreten von syntaktischen, semantischen oder lexikalischen Fehlern werden diese, zusammen mit ihrer gesamten Zahl, an das Ausgabegerät ausgegeben. Falls schwerwiegende Fehler gefunden wurden, kann es passieren, daß trotzdem ein fehlerhaftes Ausgabefile object-file-name erzeugt wird. Ob ein File ein ungültiges Objektfile ist, kann durch ein Erkennungsbit im Rückgabewert der Wiederherstellungsfunktion restoreFrom festgestellt werden. Der Wert muß gleich 12345 sein.

Generator

Der Generator p2c liest ein vom Translator pet erzeugtes Objektfile und generiert eine Menge von C Dateien, vier Dateien für jeden Modulkörper -- die Spezifikation inclusive --, mit Modulkörpernamen modul-name, modul-namefsm.h, modul-namefsm.c,
modul-namefunc.h und modul-namefunc.c. Es werden nur gültige Objektfiles gelesen.
Ein Objektfile ist gültig, wenn das Erkennungsbit im Rückgabewert p = restoreFrom(object-file-name) gleich 12345 ist.

Der Aufruf des Generators erfolgt gemäß der Syntax:

p2c [-n] object-file-name [.o] ,

wobei die Option "n" den Generator dazu veranlaßt, eindeutige Spezifikations- und Modulkörpernamen und somit eindeutige Filenamen zu erzeugen. Jeder Name besteht aus höchstens vier Zeichen. Bei Nichteingabe der Option "n" werden die ersten vier Zeichen von dem Spezifikations- oder Modulkörpernamen genommen. Es wird vorausgesetzt, daß sie eindeutig sind. Der Spezifizierer muß in diesem Fall für ihre Eindeutigkeit bis auf vier Zeichen selbst sorgen.

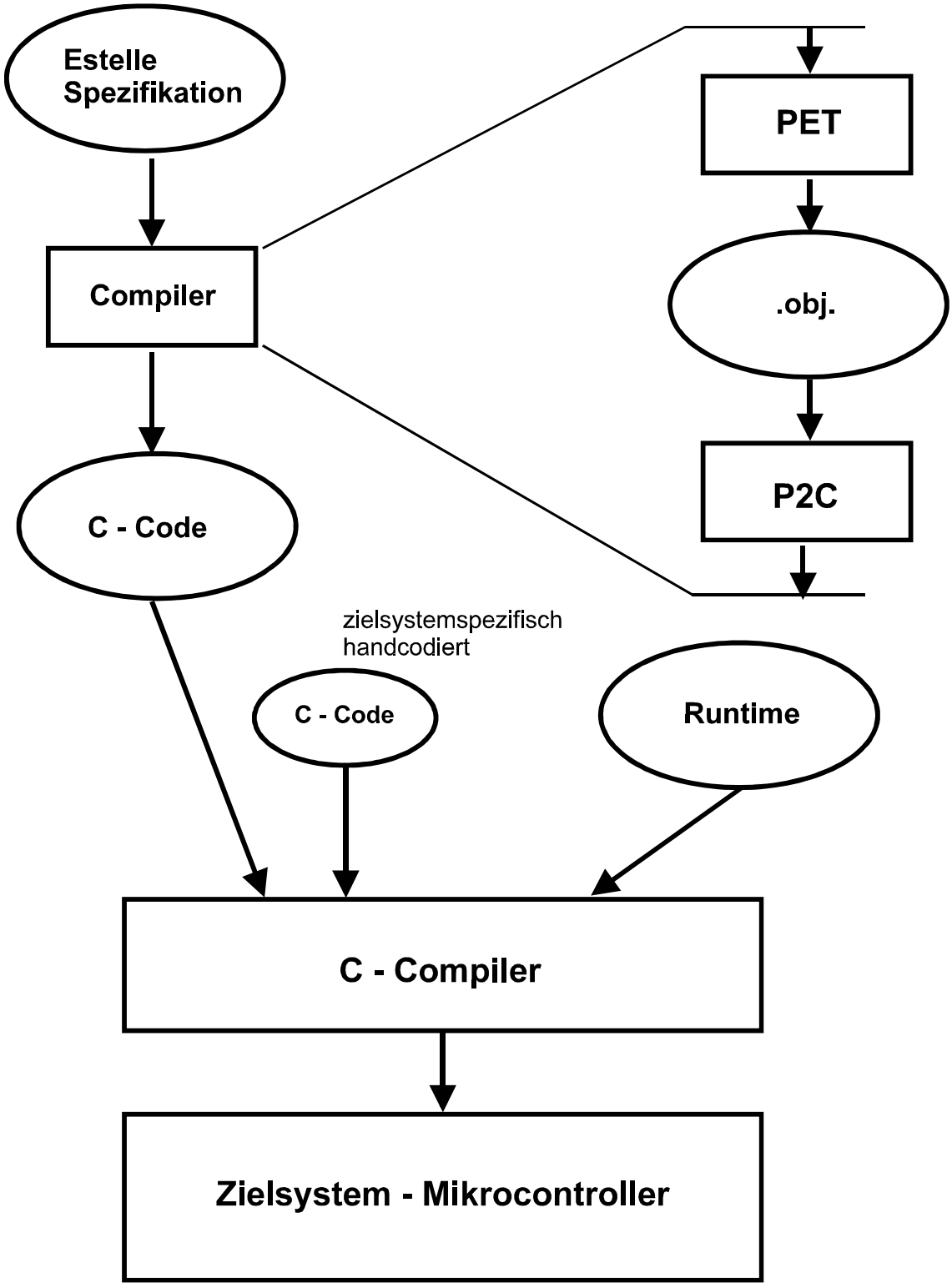

Abb. 1: Struktur des Tools

Der Generator meldet sich mit folgendem Text:

```
PET 2 C Generator (p2c)
Version 1.0
Institute for Automation
Otto-von-Guericke University of Magdeburg
D-39016 Magdeburg, GERMANY

>>Reading input, object file object-file-name.o
>>Producing C code for specification-name
```

Bei der automatischen Erzeugung von Namen für die Spezifikation und die Modulkörper wird wie folgt vorgegangen: Der oberste Modulkörper -- der Körper der Spezifikation -- erhält als Name die Folge seiner ersten vier Zeichen z.B. SPEC im Spezifikationsnamen. Jeder Untermodulkörper der Spezifikation erhält als Name -- in der Reihenfolge seines Auftretens in der Spezifikation -- ASPE, BSPE, CSPE, ..., ZSPE. Wenn ein Untermodulkörper vor dieser Umbenennung den Namen name-modul hatte, so wird jedem Untermodulkörper von dem name-modul der Namen ANAME, BNAME, CNAME, , ZNAME zugewiesen. Hierbei werden nur die ersten vier Zeichen von name-modul berücksichtigt und groß geschrieben.

4.2. PETDINGO

PETDINGO ist ein Compiler, der aus dem Translator Portable Estelle Translator (PET) und dem Generator Distributed Implementation Generator (DINGO) besteht.
Das Tool DINGO nutzt das Zwischenergebnis des Tools PET und generiert daraus eine Prototyp Implementierung, die in Estelle spezifiziert wurde. Diese Prototypimplementierung wird in C++ -Code generiert und ist auf Workstation lauffähig. Der entstehende Code ist damit nicht für die Ziele des Projekts nutzbar, da kein C -Code entsteht, der auch für Mikrocontroller geeignet ist.
Der Translator ist dagegen ein universelles Tool, denn er liefert einen definierten Zwischencode, mit dem weitergearbeitet werden kann.
Es entstehen ein Listing und ein Objektfile „*.obj“, welches das statische Modell repräsentiert. Dieses Objektfile ist für die weitere Umsetzung der Estelle Spezifikation geeignet.
Das Modell ist ein Set von Klassen, welches alle möglichen Estelle Konstrukte und Objekte beschreibt. Es wird statisches Modell genannt, weil nur statische Aspekte von Estelle Konstrukten zu diesem Zeitpunkt relevant sind. Die Klassen des statischen Modells beschreiben generische Objekte, während die Instanzen dieser Klassen mit einzelnen Ereignissen dieser Objekte einer Estelle-Spezifikation korrespondieren. Das ist in der Hauptsache eine eins zu eins Beziehung zwischen den Estelle Konstrukten und den Klassen des statischen Modells. Jede Klassen Beschreibung erfaßt die Struktur von Objekten, die sie darstellen und die verbundenen semantischen Terme einer Sammlung von Funktionen, die auf die Objekte anwendbar sind.
Während des Parserlaufes bildet PET Instanzen von Klassen des statischen Modells, die alle Objekte im Estelle Sourcefile darstellen. Diese Objekte werden abgespeichert und es werden Funktionen bereitgestellt zur Interpretation des Inhaltes dieses File´s und zur Wiederherstellung der Spezifikation im Speicher.

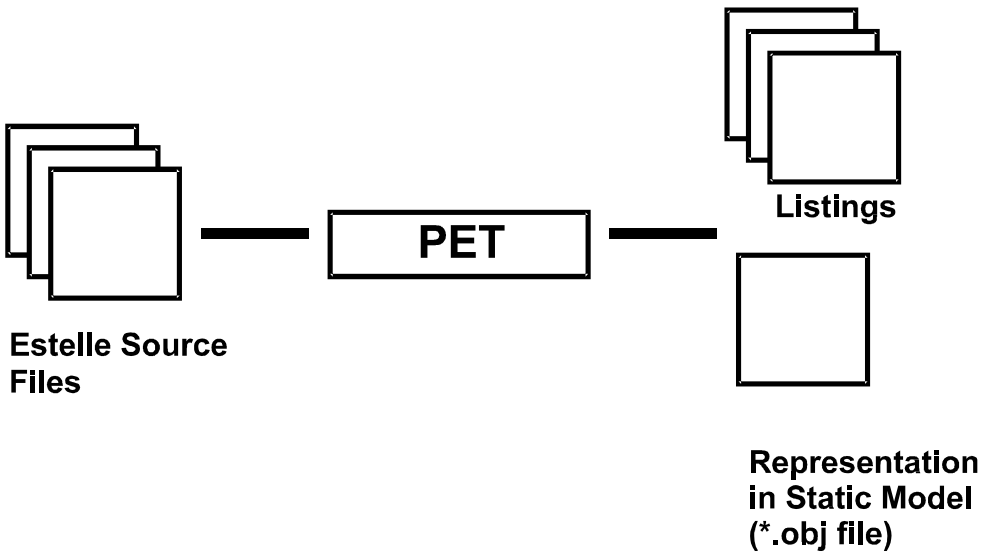

Abb. 2: Struktur des PET - Tools

Deshalb kann PET als ein frontend für jedes beliebige Tool, welches Estelle Spezifikationen verarbeitet eingesetzt werden.
Klassen des statischen Modells:

- Komponenten der Objekte
- Funktionen zur Bildung der Objekte
- Zugriffsfunktionen zu den Objekt Komponenten
- Konsistenz der Testfunktionen
- Funktionen zum speichern/laden von einem File.

Statisches Modell

defClassList.hxx	Liste der Klassen des Modells
def.hxx, def.cxx	Header- und Implementations-Files der Basis-Klassen
type.hxx, type.cxx	Klassen von Estelle Typen
const-vars.hxx, const-vars.cxx	Klassen von Estelle Konstanten und Variablen
ip.hxx, ip.cxx	Klassen von Estelle Interaktionspunkten und Kanälen
mod.hxx, mod.cxx	Klassen von Modul Headern und Körpern
trans.hxx, trans.cxx	Klassen von Transitionen
proc.hxx, proc.cxx	Klassen von Estelle Prozeduren und Funktionen
stmt.hxx, stmt.cxx	Klassen von Estelle statements
expr.hxx, expr.cxx	Klassen von Estelle Gleichungen

Andere Dateien

estgram.y	Eingang von Bison
estgramm.tab.cxx	C++ Code generiert von Bison für den Parser
lex.cxx	Lexikalische Analyse
error.hxx, error.cxx	Fehlermeldungen
container.hxx, container.cxx	Container Klassen
EST.hxx, EST.cxx	Spezifikations- und Implementationsdatei
main.cxx	Hauptdatei von PET
makefile	Benutzt C++ System des Entwicklers

4.3. p2c (Generator)

4.3.1. Komponentenfiles

Der Generator p2c besteht aus folgenden Files:

Makefile zur Compilierung von p2c.

readobj.cxx Hauptdatei zum Lesen des Objectfiles.

genmod.cxx Funktionen und Prozeduren zur Implementierung von Modulkörpern.

gentrans.cxx Funktionen und Prozeduren zur Implementierung von Transitionen.

genstmt.cxx Funktionen und Prozeduren zur Implementierung von Anweisungen.

genfunct.cxx Funktionen und Prozeduren zur Implementierung von Funktionen und Prozeduren.

gentypes.cxx Funktionen und Prozeduren zur Implementierung von Typen und Konstanten.

genexpr.cxx Funktionen und Prozeduren zur Implementierung von Ausdrücken.

p2c.ps Dokumentation.

Zusätzlich werden PET-eigene Files benötigt, die die Klassen und Methoden implementieren. Diese sind: defClassList.hxx, def.hxx, def.cxx, container.hxx, container.cxx,mod.hxx, mod.cxx, ip.hxx, ip.cxx, type.hxx, type.cxx, trans.hxx, trans.cxx, stmt.hxx, stmt.cxx, proc.hxx, proc.cxx, expr.hxx, expr.cxx, const vars.hxx, const vars.cxx.

4.3.2. Ausgabe

Für jeden Modulkörper, der Körper der Spezifikation selbst wird als ein Modulkörper aufgefaßt, mit dem Modulkörpernamen modul-name werden vier Dateien modul-namefsm.h, modul-namefsm.c, modul-namefunc.c und modul-namefunc.c erzeugt.

modul-namefsm.h	enthält Konstanten- und Typimplementierungen für den Modulkörper modul-name.
modul-namefsm.c	ist die Implementierung des erweiterten endlichen Automaten des Modulkörpers modul-name.
modul-namefunc.h	enthält die Definitionen der im Modulkörper modul-name verwendeten Funktionen und Prozeduren.
modul-namefunc.c	enthält die Implementierungen der im Modulkörper verwendeten Funktionen und Prozeduren.

Aufbau der Headerdatei modul-namefsm.h

Im Kopf der Headerdatei befinden sich vorrangig die Einfügungen der Konfigurationsdatei config.h, der Laufzeitbibliothek runtime.h, der dem Modul eigene DPR- modul-namedpr.h und die Time-Datei modul-nametime.h. Danach folgen die Konstanten- und Datentypdefinitionen und die Definition der Struktur definemodul-namemachine für den Modulkörper modul-name. Die Struktur definemodul-namemachine setzt sich aus Zeigern auf Interaktionspunkten, internen Variablendefinitionen und eventuell aus Zeigern zu den Modulvariablen der Sohnmodule zusammen. Am Ende wird der Zeiger modul-name für den Protokoll-Typ exportiert.

```
#ifndef __modul-nameFSM__
#define __modul-nameFSM__
#include <config.h>
#include <runtime.h>
#include <modul-namedpr.h>
#include <modul-nametime.h>

/* Konstantendefinitionen */

/* Datentypdefinitionen */

/* Struktur für den Modulkörper modul-name */

struct modul-namemachine {
MACHINE INFO _m;
struct FSM IP *DPM2_A; /* Zeiger auf den Interationspunkt DPM2_A */
struct FSM IP *DPM2_B; /* Zeiger auf den Interationspunkt DPM2_B */
struct FSM IP *DPM2_C; /* Zeiger auf den Interationspunkt DPM2_C */
struct FSM IP *DPM2_D; /* Zeiger auf den InterationspunktDPM2_D */
/* Variablendefinitionen */
/* Zeiger auf die Modulvariablen der Sohnmodule */
};

/* Variable für den Protokoll-Typ */
```

extern struct FSM_Type *modul-name;

#endif

Aufbau der Moduldatei modul-namefsm.c

1. /* Include */

   #include <config.h>
   #include <define.h>
   #include <runtime.h>
   #include <memory.h>
   #include <modul-namefunc.h>
   #include <modul-namedpr.h>
   #include <modul-nametime.h>

2. /* Eingaben */

   Für die Eingabesignale eines Moduls werden Konstanten gemäß

   #define input-signal-name constant

   definiert. Zunächst werden die Interaktionspunkte eines Moduls ermittelt und anhand der ihnen zugeordneten Kanäle, Rolle und Gegenrolle, die Eingabesignale in der Reihenfolge ihres Auftretens in der Kanaldefinition als Konstanten mit der Nummer ihres Auftretens definiert. Die Eingabesignale eines Interaktionspunktes sind die Signale, die der Interaktionspunkt über der Gegenrolle empfangen kann.

3. /* Ausgaben */

   Ausgabesignale werden wie bei der Definition von Eingabesignalen mit einer Konstanten deklarariert.

   #define input-signal-name constant

   Die Ausgabesignale werden hier im Gegensatzt zu oben über die Rolle in der Interaktionspunktdeklaration ermittelt.

4. /* Zustände des Moduls modul-name */

   Die Zustände eines Moduls werden ebenfalls über Konstanten definiert. Diese Konstanten geben gleichzeitig das i-te Auftretens der Zustände in der Zuständedefinition in der Estelle-Spezifikation.

5. /* Konstanten */

Weitere Konstanten, die im Deklarationsteil des Moduls auftreten, werden hier implementiert. Bei der Implementierung von Aufzählungstypen werden die einzelnen Konstantenelemente als C Konstanten mit der Nummer ihres Auftretens definiert. Diese werden hier implementiert, obwohl die eigentliche Typdefinition sich in der Header-Datei modul-namefsm.h befindet.

6. /* Init-Funktionen */

Die Initialisierungsfunktionen modul-nameIPInit und modul-nameInit implementieren die Initialisierung des Interaktionspunktes in der Modulinstanz machp. Die Zugehörigkeit zu einem physischen Kanal wird über physChan festgelegt.

```
void modul-nameIPInit(structmodul-namemachine *machp,
                                                    struct FSMIP *ip)

void modul-nameInit(struct modul-namemachine *machp,
                                                    BYTE physChan)
```

7. /* Provided-Funktionen */

Provided-Klauseln als Teil von Transitionen setzen zusätzliche Bedingungen an die Eingabeparameter, wovon der Eintritt der Transition abhängt. Für jede Transition wird eine Provided-Funktion geschrieben, die die Provided-Klausel implementiert. Der Rückgabewert TRUE oder FALSE einer Provided-Funktion besagt, ob die Bedingungen gelten oder nicht.

```
BOOL modul-nameprd001(struct modul-namemachine *machp,
                                                    int queuenb)
```

Der Name einer Provided-Funktion setzt sich aus dem Modulnamen modul-name,
dem Kürzel prd und einer dreistelligen Folge von Ziffern zusammen. Die Übergabeparameter beschreiben die Modul-Instanz machp und die Nummer der Warteschlange, in der sich das Eingabesignal befindet.

8. /* Aktion-Funktionen */

Eine Aktion-Klausel ist der eigentliche Körper einer Transition und umfaßt Anweisungen zur Veränderung von lokalen Daten, von Eingabesignalen übertragenen Werten und zum Senden von Signalen über Interaktionspunkte an ein anderes Modul. Eine Aktion-Klausel wird durch eine Aktion-Funktion implementiert.

```
BOOL modul-nameact001(struct modul-namemachine *machp,
                                                    int queuenb)
```

Der Name einer Aktion-Funktion setzt sich aus dem Modulnamen modul-name, dem Kürzel act und einer dreistelligen Folge von Ziffern.

9. /* Transitionszuordnungstabelle */

Die Transitionszuordnungstabelle ist ein zweidimensionales Feld von Integer-Werten. Ihre Dimension ergibt sich aus der Anzahl von Zuständen modul-nameNSTATE und der Anzahl von Eingabesignalen modul-nameNINPUT, beide um 1 erhöht.

```
int modul-nametr[modul-nameNSTATE+1][modul-nameNINPUT+1]
```

Die Transitionszuordnungstabelle ordnet jedem Paar bestehend aus einem Zustand und einem Eingabesignal eine Transition zu. In der Transitions-zuordnungstabelle sind alle Transitionen repräsentiert. Sie hat einen engen Zusammenhang zur Transitionsliste, da die Nummer der Transition die Position der zugehörigen Transition in der Transitionsliste angibt.

10. /* Transitionsliste */

Die Transitionsliste ist ein eindimensionales Feld von Nummern, die die nächste Transition in der Reihenfolge, das jeweilige Auftreten der Transition im Modul und
den nächsten Nachfolgerzustand angibt.

```
struct FSMTrans modul-nametrlist[modul-nameNTRLIST+1]
```

Ein Element in der Transitionsliste ist also ein Tripel der Form: { next-trans, trans, next-state }. Die Kennzeichnung der nächsten Transition in der Reihen-folge ergibt sich aus dem Grund, daß für eine Eingabe in einem gegebenen Zustand mehrere Transitionen existieren können, deren Eintreten beispielsweise von dem Wahrheitswert der Provided-Klausel abhängen.

11. /* Funktionspointerlisten */

Die Funktionspointerlisten enthalten die Verweise auf die Aktion- und Provided-Funktionen. Die Aktion- und Provided-Funktionen mit dem gleichen Index gehören zu einer Transition. Über die dort eingetragenen Funktionszeiger wird eine Aktion- oder Provided-Funktion aufgerufen.

12. /* Zeiger auf Modulstruktur */

Diese Variable repräsentiert den Modul-Typ.

```
struct FSM_Type * modul-name = & modul-namefsm_t;
```

13. /* Implementierung der Initialisierungsfunktion für Interaktionspunkte */

Für jeden Interaktionspunkt ip mit der IP-Struktur werden Initialisierungen vorge-nommen.

```
ip->IP_Queue = 0;
ip->IP_DestIP = NULL;
ip->IP_Machine = (struct FSM *) machp;
ip->IP_InOff = ...;
```

Diese betreffen die Nummer der dem Interaktionspunkt zugeordneten Warte-schlange IPQueue des Moduls, den Interaktionspunkt über den Ausgaben vom

Interaktionspunkt eingetragen werden, die Adresse der Modul-Instanz machp und den Offset-Wert, um den die Eingabenummer aller über diesen Interaktionspunkt ankommenden Eingabesignale beim Eintragen in die Warteschlange erhöht wird.

```
void modul-nameIPInit(struct modul-namemachine *machp,
                                                    struct FSM_IP *ip)
```

14. /* Implementierung der Initialisierungsfunktion */

```
void modul-nameInit(struct modul-namemachine *machp, BYTE physChan)
```

15. /* Implementierung der Provided-Funktionen */

Am Anfang jeder Provided-Funktion muß eine Funktion aufgerufen werden, um auf den ersten Wert der Eingabewarteschlange zugreifen zu können.

```
BOOL modul-nameprd017(struct modul-namemachine * machp,
                                                    int queuenb)

EVP inEvent_ptr = machp->m.MQueues[queuenb].QFirst;
```

16. /* Implementierung der Aktion-Funktionen */

Gleiches gilt auch für die Aktion-Funktion.

```
BOOL modul-nameact017(struct modul-namemachine *machp,
                                                    int queuenb)

EVP inEvent_ptr = machp->m.MQueues[queuenb].QFirst;
```

Aufbau der Headerdatei modul-namefunc.h

1. /* Include */

```
#ifndef __ModFUNC__
#define __ModFUNC__
#include <config.h>
#include <define.h>
#include <runtime.h>
#include <memory.h>
#include <Moddpr.h>
#include <Modfsm.h>
```

2. /* Definition von Funktionen und Prozeduren */

Aufbau der Funktionsdatei modul-namefunc.c

1. /* Include */

   ```
   #include <config.h>
   #include <define.h>
   #include <runtime.h>
   #include <memory.h>
   #include <Modfunc.h>
   ```

2. /* Implementierung von Funktionen und Prozeduren */

4.3.3. Generierung

Die Datei readobj.cxx

- main(int argc, char* argv[])
  ist die Hauptfunktion, die die Parameter aus dem Eingabefile liest, auf Korrektheit überprüft, das Objektfile object-file-name gemäß p = (PSpecDef) restoreFrom(object-file-name) liest und mit genSpec(p, &corresp) die Funktion zur Generierung von C-Code für die Estelle-Spezifikation anstößt.

- static void nameForSpec(PSpecDef pd)
  erzeugt bei Aufruf des Generators mit der Option n eindeutige Namen für die Spezifikation und die Module.

- static void nameForBody(PBodyDef pd)
  erzeugt bei Aufruf des Generators mit der Option n eindeutige Namen für die Module. Sie wird von der Funktion nameForSpec aufgerufen.

Die Datei genmod.cxx

- void genSpec(PSpecDef pd, ostream* correspF)
  generiert C-Code für die Spezifikation pd. correspF und ist ein File zu Debugging-Zwecken. Der C-Code wird in die entsprechenden Dateien specification-namefsm.h, specification-namefsm.c, specification-namefunc.h und specification-namefunc.c geschrieben.

- static void genBody(PBodyDef pd, NameGen* ngen)

- static void defStructOfMod(PDef pd, ostream& hxx, NameGen* ngen, char* n)
  definiert die Struktur des Moduls.

- static int nINPUT(PDef pd, NameGen* ngen)
  ermittelt die Anzahl von Eingabesignalen für das Modul pd. Der Zeiger pd kann auf die Spezifikation oder auf den Modulkörper zeigen.

- static int nOUTPUT(PDef pd, NameGen* ngen)
  berechnet die Anzahl von Ausgabesignalen für das Modul pd. Der Zeiger pd kann ein Zeiger auf die Spezifikation oder auf den Modulkörper sein.

- static int nIPS(PDef pd, NameGen* ngen)
  gibt die Anzahl der Interaktionspunkte für das Modul pd zurück.

- static PDList defInputs(PDef pd, ostream& cxx, NameGen* ngen)
  ermittelt die Eingabesignale und generiert gleichzeitig ihre Definitionen. Eingabesignale werde in der Reihenfolge ihres Auftretens durchnummeriert. Jedes Signal wird als eine Konstante mit der zugehörigen Nummer ihres Auftretens definiert.

- static void defOutputs(PDef pd, ostream& cxx, NameGen* ngen)
  ermittelt die Ausgabesignale und generiert gleichzeitig ihre Definitionen. Ausgabesignale werde in der Reihenfolge ihres Auftretens durchnummeriert. Jedes Signal wird als eine Konstante mit der zugehörigen Nummer ihres Auftretens definiert.

- static void defIPInitAndInit(ostream& cxx, char* modul-name)
  generiert in der Datei cxx die Deklarationen für die Initialisierungsfunktion modul-nameIPInit von Interaktionspunkten und die Initialisierungsfunktion modul-nameInit des Modul´s mit dem Modulnamen modul-name.

- static void genIPInit(PDef pd, ostream& cxx, NameGen* ngen, char* modul-name)
  generiert in der Datei cxx für das Modul pd mit dem Modulnamen modul-name die Implementierung der Initialisierungsfunktion modul-nameIPInit von Interaktionspunkten.

- static void genInit(PTransPart ptp, PDef pd, ostream& hxx, ostream& cxx, NameGen* ngen, char* modul-name)
  generiert die Initialisierungsfunktion modul-nameInit des Moduls mit dem Modulnamen modul-name.

- static void defTypeDesc(ostream& fsmcxx, char* modul-name)
  generiert in der Datei fsmcxx die Definition des Typ-Deskriptors fsmt für den Modulnamen modul-name. Der Typedeskriptor wird als eine Struktur FSM_Type mit den Variablen und

- static void openModFiles(filebuf* f1, filebuf* f2, filebuf* f3, filebuf* f4, ostream** th, ostream** tc, ostream** mh, ostream** mc, char* mn)

- static void defIncludeFsmh(ostream& fsmh, char* modul-name)
  definiert in der Datei modul-namefsm.h die Include-Dateien runtime.h, modul-namedpr.h und time.h. Diese werden dann definiert, wenn zuvor keine __modul-nameFSM__ Variable definiert worden war. In diesem Fall wird __modul-nameFSM__ definiert.

- static void defIncludeFsmc(ostream& fsmc, char* modul-name)
  definiert in der Datei modul-namefsm.c die Include-Dateien config.h, runtime.h, memory.h, modul-namedpr.h und time.h.

- static void defIncludeFunch(ostream& funch, char* modul-name)
  definiert in der Datei modul-namefunc.h die Include-Dateien config.h, define.h, runtime.h und modul-namedpr.h. Diese werden dann definiert, wenn zuvor keine __modul-nameFUNC__ Variable definiert worden war. In diesem Fall wird __modul-nameFUNC__ definiert.

- static void defIncludeFuncc(ostream& funcc, char* modul-name)
  definiert in der Datei modul-namefunc.c die Include-Dateien config.h, define.h, runtime.h, memory.h und modul-namefsm.h.

Die Datei gentrans.cxx

- void defActsAndPdrsList(ostream& fsmcxx, int NTRLIST, char* n)
  generiert in der Datei fsmcxx die Definitionen für die Provided- und Aktionen-Pointerlisten.

- void defPrdAndActs(ostream& fsmcxx, int NTRLIST, char* n)
  generiert in der Datei fsmcxx die Definitionen für die Funktionen, die die Provided- und die Aktion-Klauseln implementieren. Für jede Transition werden eine Provided-
  und eine Aktion-Implementierung unabhängig davon, ob die Transition eine Provided-Klausel besitzt oder die Aktion-Klausel leer ist generiert.

- static void defPrd(ostream& fsmcxx, int Pdr, char* n)
  generiert in der Datei fsmcxx die Definition der Provided-Funktion der i-ten Transition des Moduls mit Modulnamen modul-name.

- static void defAct(ostream& fsmcxx, int Act, char* n)
  generiert in der Datei fsmcxx die Definition der Aktion-Funktion der i-ten Transition des Moduls mit Modulnamen modul-name.

- void genPrds(PTransPart ptp, PDef pbd, ostream& cxx, NameGen* ngen, char* modul-name)
  generiert in der Datei cxx für den Transitionsteil ptp des Moduls pbd mit dem Modulnamen modul-name die Implementierungen der zugehörigen Provided-Funktionen.

- void genActs(PTransPart ptp, PDef pbd, ostream& hxx, ostream& cxx, NameGen* ngen, char* modul-name)
  generiert in den Dateien hxx und cxx für den Transitionsteil ptp des Moduls pbd mit dem Modulnamen modul-name die Implementierungen der zugehörigen Aktion-
  Funktionen.

- void genTransBlock(PTransBlockDef p, PDef pd, ostream& cxx, NameGen* ngen)
  generiert in der Datei cxx die Implementation des Transitionsblockes p des Moduls pd.

- void genTransTable(PTransPart ptp, ostream& cxx, PDList inEventList, PDList states, NameGen* ngen, char* n)
  generiert in der Datei cxx für den Transitionsteil ptp, die Liste von Eingabesignalen inEventList und die Liste von Zuständen states des Moduls mit dem Modulnamen modul-name und die entsprechende Transitionstabelle.

- static int genNextTrans(PTransPart ptp, PDList TrList, PTransition pt)
  gibt für die Transition pt in der Transitionsliste TrList vom Transitionsteil ptp

die nachfolgende Transitionsnummer an.

Die Datei genstmt.cxx

- void genStmt(PDef stmt, PDef pdb, ostream& cxx, NameGen* ngen, PDList imports, int ind, int& needEndLabel)
  generiert in der Datei cxx die Implementation für die Anweisung stmt mit der Importliste imports. Die ganzzahlige Konstante ind gibt die Einrücktiefe für die formatierte Ausgabe der Implementation der Anweisung stmt an. Jede Anweisung endet mit einem Semikolon ;.

  - Blockanweisung
  - Leere Anweisung
  - Ergibt-Anweisung
  - While-Anweisung
  - Repeat-Anweisung
  - If-Anweisung
  - Case-Anweisung
  - Anweisungsteil einer Case-Anweisung
  - For-Anweisung
  - With-Anweisung
  - Return-Anweisung
  - Prozeduraufruf
  - All-Anweisung
  - Forone-Anweisung
  - Output-Anweisung
  - Init-Anweisung
  - Terminate-Anweisung
  - Release-Anweisung
  - Attach-Anweisung
  - Detach-Anweisung
  - Connect-Anweisung
  - Disconnect-Anweisung.

- static int pointerFunct(char* n)

Die Datei genfunct.cxx

- void defFAndPs(PDList pdl, ostream& hxx, ostream& cxx, NameGen* ngen, char* n)

- void defForP(PForPDef pd, ostream& hxx, ostream& cxx, NameGen* ngen, char* n)

- void genHeaderDef(PForPDef p, ostream& os, int imported, NameGen* ngen, char* n)

- void genHeader(PForPDef p, ostream& os, int imported, NameGen* ngen, char* n)

- static void genImpl(PForPDef p, ostream& cxx, PDList imports, NameGen* ngen)

Die Datei gentype.cxx

- int complexType(PDef pd)
  überprüft, ob der Datentyp pd ein komplexer Datentyp ist, d.h., ein Datentyp mit der Kategorie ARRAYTYPE, RECORDTYPE, UNSPECIFIED, SETTYPE oder STRING-TYPE.

- int predefinedType(PTypeDef pd)
  überprüft, ob die Datentypdefinition pd eine direkte Referenz zu einem vordefinierten Datentyp ist, d.h., zu einem Datentyp mit der Kategorie BOOLEANTYPE, INTEGERTYPE, REALTYPE, CHARACTERTYPE oder SUBRANGETYPE.

- void genLowBound(PTypeDenoter pt, ostream& os, NameGen* ngen)
  gibt in die Datei os die untere Schranke, d.h., der minimale zulässige Wert des Datentypbezeichners pt aus. Für einen referenzierten Datentyp der Kategorie TYPEREF ist sie die untere Schranke des Typs von pt. Für die Datentypen mit der Kategorie ENUMERATEDTYPE, BOOLEANTYPE, CHARACTERTYPE ist sie 0. Für den Integertyp, d.h. der mit der Kategorie SUBRANGETYPE ist sie __MAXINT. Für einen Unterbereichstyp ist sie der minimal zugelassene Wert des Unterbereichstyps.

- void genUpBound(PTypeDenoter pt, ostream& os, NameGen* ngen)
  gibt in die Datei os die obere Schranke, d.h., den maximal zulässigen Wert des Datentypbezeichners pt aus. Für einen referenzierten Datentyp der Kategorie TYPEREF ist sie die obere Schranke des Typs von pt. Für den Aufzählungstyp mit der Kategorie ENUMERATEDTYPE ist sie gleich der Anzahl der Werte minus 1. Für den Booleantyp, mit der Kategorie BOOLEANTYPE, ist sie 1. Für den Character-Typ, mit der Kategorie CHARACTERTYPE, ist sie 127. Für den Integertyp, d.h. den mit der Kategorie SUBRANGETYPE, ist sie __MAXINT. Für einen Unterbereichstyp ist sie der maximal zugelassene Wert des Unterbereichstyps.

- int LowBound(PTypeDenoter pt, NameGen* ngen)
  gibt die untere Schranke des Datentypbezeichners pt zurück. Für einen referenzierten Datentyp ist sie die untere Schranke des referenzierenden Typs. Für den Aufzählungs- und den Integertyp ist sie gleich 0. Für einen Unterbereichstyp ist sie gleich der unteren Schranke des Unterbereichstyps selbst.

- int UpBound(PTypeDenoter pt, NameGen* ngen)
  gibt die obere Schranke des Datentypbezeichners pt zurück. Für einen referenzierten Datentyp ist sie die obere Schranke des referenzierenden Typs. Für den Aufzählungstyp ist sie gleich der Anzahl der Werte minus 1. Für einen Unterbereichstyp ist sie gleich der unteren Schranke des Unterbereichstyps selbst.

- int RefToConst(PDef pd, NameGen* ngen)
  gibt den konstanten Wert der referenzierten Konstanten pd zurück. Wenn die referenzierte Konstante mit \-\ behaftet ist, wird sie mit -1 multipliziert und zurückgegeben. Für integer und reelle referenzierte Konstanten ist der Wert der Konstante, Wert der integer und reellen Konstanten selbst. Für eine Konstantendefinition ist ihr Wert der konstante Wert der referenzierten Konstanten. Die

Konstantendefinitionen können Konstanten vom Aufzählungs- Integer-, oder Realtyp sein.

- char* refToType(PDef pd, NameGen* ngen)
- void genIdxTypeSize(PDef pt, NameGen* ngen, ostream& os)
- void defTypesAndConsts(PDList pdl, ostream& hxx, NameGen* ngen)
- void defType(PDef pd, ostream& hxx, NameGen* ngen, char* typename)
- void genRefToConst(PDef pd, NameGen* ngen, ostream& os)
- void defConst(PConstDef pd, ostream& hxx, NameGen* ngen)

Die Datei genexpr.cxx

- void defExistExprs(ostream& hxx, ostream& cxx, NameGen* ngen)
- inline int additiveExpr(PExpr e)
- void genExpr(PDef e, PDef pbd, ostream& cxx, NameGen* ngen, PDList imports)
- int isDeclared(PExpr pe, PDef pd, ostream& cxx)

Hilfsfunktionen in gennames.cxx

- void codeGenErr(char* s)
- char* openFile(char*, char*, filebuf*)
- void imported(PDList, PDef, int)
- void defImported(PDList, ostream&, ostream&, ostream&, NameGen*)
- void genNames(PDef, NameGen*)

5. Testung

Die Testung des Compilers wurde aus zeitlichen Gründen an einfacheren Beispielen, wie der Spezifikation Pingpong durchgeführt. Die Auslegung der Umgebungsbedingungen des Compilers erfolgte aber nach der Spezifikation des PROFIBUS-DP Master (Klasse 1), die ebenfalls innerhalb dieses Projektes erstellt wurde. Dies betrifft vorallem die Bereitstellung der Runtime-Funktionen.
Weitere Untersuchungen werden im Rahmen der Diplomarbeit „Untersuchungen zum Einsatz von formalen Beschreibungstechniken (Estelle) in der Automatisierungstechnik“ [Dom97] durchgeführt.
Die Übersetzung des Compilers soll hier an einem einfachen Beispiel demonstriert werden.
Ausgegangen wird von der Spezifikation Pingpong, die aus einem User und 2 Providern besteht.

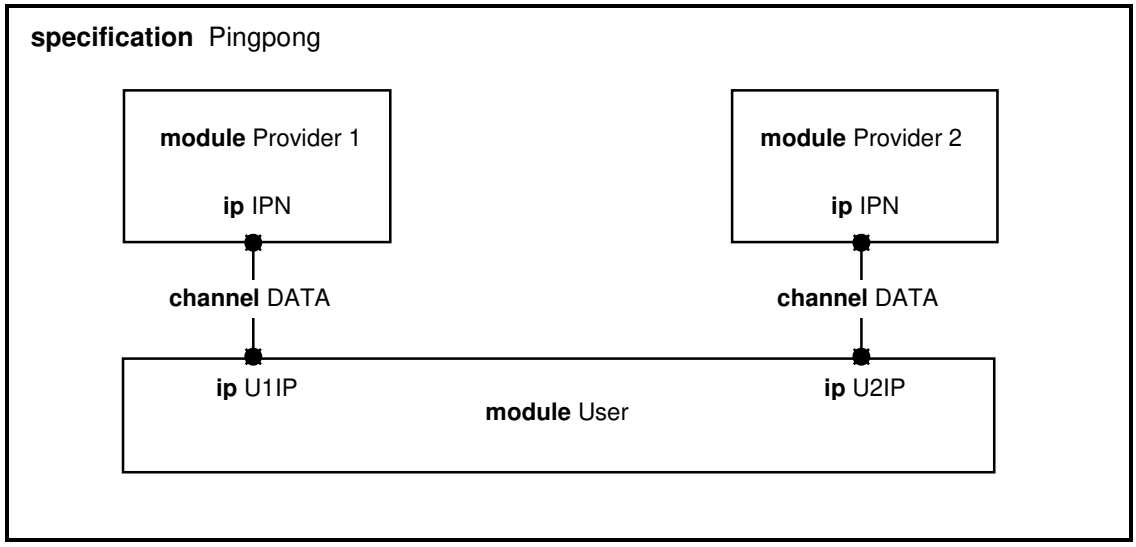

Abb. 3: Pingpong Spezifikation

Estelle Spezifikation Pingpong

```
specification Pingpong;
default individual queue;
timescale seconds;

channel DATA(User, Provider);                          Datenkanäle
      by User : DATArequest;
      by Provider : DATAresponse;

module UserType systemactivity;
      ip
        U1IP : DATA( User);                            Interaktionspunkte
        U2IP : DATA( User);
end;
```

```
module ProviderType systematicactivity;
      ip
        IPN : DATA( Provider);
end;

body UserBody for UserType;
state START, READY;
stateset ST = [START,READY];
initialize
  to
    START
      begin
      end;

trans
  from START
    to READY
      begin
      output U1IP.DATARequest;
      end;

  from READY
    to same
      when U1IP.DATAResponse
      begin
      output U2IP.DATARequest;
      end;

  from READY
    to same
      when U2IP.DATAResponse
      begin
      output U1IP.DATARequest;
      end;
end;

body ProviderBody for ProviderType;
state READY;
procedure test;
begin

end;
initialize
  to READY
    begin
    end;

  trans
    from READY
      to same
        when IPN.DATARequest
        begin
        output IPN.DATAResponse;
        end;
end;

modvar
  User            : UserType;
  Provider1       : ProviderType;
  Provider2       : ProviderType;

initialize
  begin
      init User with UserBody;
```

Modulkopf-Provider

Interaktionspunkt

Modulrumpf-Provider

Modulvariablen

```
      init Provider1 with ProviderBody;
      init Provider2 with ProviderBody;
      connect User.U1IP to Provider1.IPN;          Verbindung
      connect User.U2IP to Provider2.IPN;          User - Provider
    end;
end.
```

Das Estelle Modul ProviderType spiegelt sich nach der Compilierung in der Datei provfsm.c wieder.

Datei provfsm.c

```
#include <config.h>
#include <define.h>
#include <runtime.h>
#include <memory.h>
#include <Provfunc.h>
#include <Provdpr.h>
#include <Provtime.h>

/* Eingaben */
#define IPN_DATArequest      1

/* Ausgaben */
/* Kanalname: DATA */
#define IPN_DATAresponse     1

/* Zustaende des Moduls */
#define READY10                     0

/* Struktur - Infos des Moduls */
#define ProvNIPS             1      /* Anzahl der Interaktionspunkte  */
#define ProvNSTATE           1      /* Anzahl der Zustaende           */
#define ProvNINPUT           1      /* Anzahl der Eingaben            */
#define ProvNOUTPUT          1      /* Anzahl der Ausgaben            */
#define ProvNTRLIST          1      /* Anzahl der Zustandsuebergaenge */
#define ProvNQUEUES          1      /* Anzahl der Warteschlangen */

/* Init-Funktionen */
void ProvIPInit(struct Provmachine *, struct FSM_IP *);
BOOL ProvInit(struct Provmachine *, struct FSM_InitData *);

/* Provided-Funktionen */
BOOL Provprd001(struct Provmachine *, int);

/* Action-Funktionen */
void Provact001(struct Provmachine *, int);

/* Transitionsliste */
int Provtr[ProvNSTATE][ProvNINPUT+1] = {
{ /* STATE READY10 */
/* 0 : READY10 X SPONTANEOUS          */  0,
/* 1 : READY10 X DATArequest          */  1,
}, /* STATE READY10 */

};

/* Transitionszuordnungstabelle */
struct FSM_Trans Provtrlist[ProvNTRLIST+1] = {
/*-------------------------------------------------------*/
/* ENTRY #     NEXT          TRANS          RESULT       */
/*-------------------------------------------------------*/
/* 0 */ {      0,            0,             0            },
/* 1 */ {      0,            1,             SAME_STATE   },
};

/* Funktionspointerlisten */
PRD_FUNCPTR Provprdfunc[] = {
/* 0 */ 0,
/* 1 */ (PRD_FUNCPTR) Provprd001
};
```

```
ACT_FUNCPTR Provactfunc[] = {
/* 0 */ 0,
/* 1 */ (ACT_FUNCPTR) Provact001
};

/* Modulstruktur */
struct FSM_Type Provfsm_t = {
/* F_NStates */ ProvNSTATE,                /* Anzahl der Zustaende                      */
/* F_NInputs */ ProvNINPUT,                /* Anzahl der Eingaben                       */
/* F_NQueues */ ProvNQUEUES,               /* Anzahl der Warteschlangen                 */
/* F_NTrList */ ProvNTRLIST,               /* Anzahl der Transitionen                   */
/* F_TrTable */ (int **) Provtr,           /* Zeiger auf Transitionszuordnungstabelle   */
/* F_TrList  */ Provtrlist,                /* Zeiger auf Transitionsliste               */
/* F_TrFuncs */ Provactfunc,               /* Zeiger auf act-Funktionspointerliste      */
/* F_PrFuncs */ Provprdfunc,               /* Zeiger auf prd-Funktionspointerliste      */
/* F_IPInit  */ (IPINIT_FUNCPTR) ProvIPInit,/* Zeiger auf Init-Funktion fuer IPs        */
/* F_Init    */ (INIT_FUNCPTR) ProvInit,   /* Zeiger auf Init-Funktion fuer Modul       */
/* F_Size    */ sizeof(struct Provmachine),/* Groesse der Modulinstanz in Bytes         */
/* F_Nips    */ ProvNIPS                   /* Anzahl der Interaktionspunkte             */
};

/* Zeiger auf Modulstruktur */
struct FSM_Type * Prov = &Provfsm_t;

void ProvIPInit(struct Provmachine *machp, struct FSM_IP *ip) {
  ip->IP_Queue   = 0;
  ip->IP_DestIP  = NULL;
  ip->IP_Machine = (struct FSM *) machp;
  ip->IP_InOff   = 0;
  machp->IPN = ip;
}

BOOL ProvInit(struct Provmachine * machp, struct FSM_InitData * fsmInitData) {
 machp->_m.STATE = READY10;
 {
   ;
 }
 return TRUE;
}

/* Provided-Funktion Provprd001 */
BOOL Provprd001(struct Provmachine * machp, int queuenb) {
 return TRUE;
}

/* Action-Funktion Provact001 */
void Provact001(struct Provmachine * machp, int queuenb) {
 EVP inEvent_ptr = FSM_QFirstRemove((FSMP) machp, queuenb);
 printf("Hier ist Aktion Provact001\n");
 {
   EVP outEvent_ptr = Get_Data_Buffer();
   FSM_IntraOutput(machp->IPN, outEvent_ptr, IPN_DATAresponse);
   ;
 }
 Free_Data_Buffer(inEvent_ptr);
}
```

6. Weiterführende Arbeiten

Aufbauend auf die gemachten Erfahrungen bei der Entwicklung des Compilers e2c, konnten diese Erfahrungen verallgemeinert auch für die Inangriffnahme der Entwicklung von Compilern für andere formale Spezifikationssprachen und Zielsprachen eingesetzt werden.
Hier wurde der Weg untersucht, ob es möglich ist aus einer SDL-Spezifikation [Sch93], [Glu91], [Glu93], [Glu94] VHDL-Modelle ableiten zu können. Dabei besteht ebenfalls die Möglichkeit einen Compiler aus den Komponenten Frontend und Backend zu strukturieren.
Für das Frontend ist es möglich den SDL Parser [Lut93] des SITE-Tools, SITE-SDL Integrated Environment des Instituts für Informatik der HU Berlin, zu nutzen.

SITE-Tool

Der SDL Parser besteht aus folgenden Teilen:

- Lexical Analyzer (mit Flex erzeugt), enthält Funktionen zum Filehandling (verschiedener referenzierter SDL-Quellen), die mit Kimwitu erzeugt wurden.

- Syntax Analyzer (mit YACC erzeugt)

- Zwischencode Generator (mit Kimwitu erzeugt).

Ausgehend von diesen Voraussetzungen ist es relativ unkompliziert einen Generator zu entwicklen, der VHDL-Code erzeugt. Dieser Generator wird wiederum mittels Kimwitu erzeugt. Dabei wird anhand der Syntax der erzeugte SDL-Parsebaum ausgelesen, ausgewertet und eine vorgegebene VHDL Syntax erzeugt.

Kimwitu

Kimwitu ist ein Termprozessor(-Generator) [Eij93], der an der Universität Twente (NL) entwickelt wurde. Er ermöglicht Termprozessoren zu erzeugen, welche Code anhand einer vorgegebenen Syntax analysieren und in eine Objektdarstellung transformieren. Dabei ist Kimwitu besonders darauf ausgerichtet mit Objektbäumen zu arbeiten.
Als Eingabe für Kimwitu dienen verschiedene Informationen. Zum einen benötigt man Syntaxinformationen, welche dazu benutzt werden um Datentypen und Funktionen entsprechend der zu untersuchenden Syntax zu bilden. Des weiteren, werden Informationen benötigt, was mit der aufgefunden Syntax (dem Parsebaum) zu tun ist. Diese Informationen können entweder C-Funktionen, Unparse Regeln oder Rewrite Regeln sein.
Unparse Regeln werden dazu benutzt einen Parsebaum auszuwerten. Rewrite Regeln dazu einen Parsebaum zu verändern. Funktionen können dazu benutzt werden den Unparsebaum zu verändern oder auszuwerten.
Bei einer Übersetzung der Regeln gemeinsam mit der Syntaxinformation werden automatisch Funktionen erstellt, welche die aufgeführten Regeln für den entsprechenden Syntaxbaum ausführen. d.h. um anschließend eine gewünschte Funktionalität zu implementieren, müssen die Funktionen in dem gewünschten Kontext aufgerufen werden.

Ansätze zur SDL - VHDL Transformation

Man muß unterscheiden zwischen struktureller Beschreibung und Verhaltensbeschreibung.

Strukturbeschreibung

SDL	VHDL
System	entity mit Strukturbeschreibung
Blocksubstructure	entity mit Strukturbeschreibung
Block	component (entity mit Verhaltensbeschreibung (oder Struktur..))
Process	mehrere Prozesse (Zustandsmaschine, Kombinatorik) (oder entity mit Verhaltensbeschreibung)
Procedure	Procedures / components

Verhaltensbeschreibung

SDL	VHDL
Prozeß	mehrere Prozesse (oder entity mit Verhaltensbeschreibung)
Zustandsmaschine	Prozeß mit Zustandsmaschinentemplate
Task	kombinatorischer und sequentieller Prozeß (der mit Zustandsmaschine kommuniziert)

Kommunikation

SDL	VHDL
Channel (mit Signallist)	Signal best. Typs (Wertevorrat) oder Signalgruppe (entspr. Signallist), die den gleichen Weg hat.
Signalroute (mit Signallist)	Signal best. Typs (Wertevorrat) oder Signalgruppe (entspr. Signallist), die den gleichen Weg hat.
Input	Eintreffen eines best. Signalwertes, Handshake eines Wertes

Realisierungsvorschlag

Der SDL-VHDL Compiler besteht aus folgenden Teilen:

- SDL Parser (erzeugt CR)
- VHDL Generator (erzeugt VHDL)

Der Parser unterstützt den vollständigen SDL`92 Sprachumfang.

Der Generator ist in der Lage folgende SDL Konstrukte in entsprechende VHDL Konstrukte zu übersetzen:

SDL	**VHDL**	**Bemerkungen**
SDL System	entity mit architecture Arch	vollständige Darstellung der Struktur
Block	entity mit architecture Struct	unvollständige Darstellung der Struktur Portmapping unvollständig
Substructure	entity mit architecture SubStruct	vollständige Darstellung der Struktur
Process	entity mit architecture Behave enthält zwei Prozesse: Graph -> Zustandsmaschine Comb -> Dataflow	
Zustandsgraph	Process Graph mit Zustands-maschinentemplate	Zustände werden vollständig übernommen und dargestellt Joins werden nur vor Zuständen erlaubt Decision ist teilweise realisiert
Operationen	Operationen	Grundoperationen werden übersetzt, folgende Operationen sind nicht unterstützt: Implikation (=>)(Boolean) Range (in) Error

7. Zusammenfassung

Die formale Spezifikation, nicht nur von Kommunikationssystemen, wird in der Zukunft eine immer größere Bedeutung erhalten.
Sie stellt den ersten Schritt der Notation dar, die es erlaubt nachfolgend eine weitestgehende automatische Bearbeitung der Problemstellungen zu ermöglichen. Dadurch ist es möglich Fehlereinflüsse in dieser Entwicklung zu minimieren und die Entwurfszeiten bis zur Umsetzung zu minimieren.
Ausgehend von entsprechenden Voruntersuchungen wurde der Weg eines durchgängigen Tools von der formalen Beschreibungssprache Estelle zu einem auch auf Mikrocontrollern ablauffähigen ANSI C - Codes gewählt.Dieser Weg wurde vorallem deshalb gewählt, da ein Tool (PETDINGO) gefunden wurde, das frei zur Verfügung steht und aus der Estelle Notation einen geprüften und objektorientierten Zwischenkode erzeugt.
Damit war ein wesentlicher Bestanteil des Compilers gefunden, der es überhaupt ermöglichte, ddas Tool innerhalb der Laufzeit des Projektes zu entwickeln.
Unter Beachtung der Randbedingungen einer konkreten Estelle-Spezifikation für einen PROFIBUS - DP Master (Klasse 1) wurden die Transformationsregeln für den Compiler gefunden.
Da, daß PETDINGO Tool eine Prüfung der Estelle - Notation auf lexikalische und syntaktische Richtigkeit durchführt ist in dieser Hinsicht von einem fehlerfreien Zwischenkode auszugehen. Dieser Zwischenkode wird dann mit dem Tool p2c, welches im Rahmen des Projektes entwickelt wurde, in entsprechenden ANSI C - Kode übersetzt. Dabei müssen die entsprechenden Bibliotheken mit den konkreten Informationen der Hardwareplattformen eingebunden werden.
Aus der Verbindung der Tools PETDINGO und p2c entsteht damit das Gesamttool, der Compiler e2c.
Das entwickelte Tool e2c stellt hier einen Anfang der Entwicklung dar. Es erlaubt die weitestgehende automatische Umsetzung von Estelle-Notationen in ANSI C und damit die Umsetzung auf konkrete Computerplattformen und Mikrocontroller.
Dadurch, daß es nur ein Tool ist, ergeben sich für den Nutzer eine einfache Handhabbarkeit. Komplizierte Parametrierungen und Auswahlverfahren entfallen, die sonst zu Fehlbedienungen führen könnten.
Die gemachten Erfahrungen wurden genutzt, um Möglichkeiten der weiteren Nutzung aufzuzeigen. Hier spielt insbesondere der Aspekt der Nutzung der Erfahrungen zur Umsetzung einer in einer belibiegen formalen Spezifikationssprache gemachten Notation in einen belibiegen Zielkode. Dies ist besonders beim Hardware-/Software - Codesign von Interesse, da hier Bestandteile der formalen Spezifikation einmal z.B. in C - Code und einmal in VHDL - Code umgewandelt werden müssen. Das entwickelte Tool wird am Institut für Automation und Kommunikation e.V. (ifak) Magdeburg/Barleben genutzt und weiter getestet. Dies geschieht im Rahmen einer Diplomarbeit.
Die gemachten Erfahrungen werden dann in die weitere Entwicklung von Übersetzungswerkzeugen, vorallem des Hardware-/Software - Codesigns einfließen.

Literaturverzeichnis

[Dom97] Domnick, G.: Untersuchungen zum Einsatz von formalen Beschreibungstechniken (Estelle) in der Automatisierungstechnik, Diplomarbeit, UNI Magdeburg FET/IFAT, 1997

[EDT94] : EDT - Estelle Development Toolset, Institut National des Telecommunications, Evry 1994

[Eic93] Eichelbaum, P.: Vorhabensbeschreibung „Implementationsuntersützende Werkzeuge für Kommunikationsprotokolle im Echtzeitbereich“, Magdeburg 1993

[Eij93] van Eijk, Peter; Belinfante, Axel; ‚The Term Processor Kimwitu, Manual and Cookbook"; 1993

[Glu91] Glunz, W.; Venzl, G.: ‚Using SDL for Hardware Design"; Proceedings SDL`91: Evolving Methods; Elsevier Science Publ.; 1991

[Glu93] Glunz, W.; Rössel T.; Kruse, T.: ‚System-level hardware design with SDL"; Proceedings SDL`93: Using Objects; Elsevier Science Publ.; 1993

[Glu94] Glunz, W.: ‚Hardware-Entwurf auf abstrakten Ebenen unter Verwendung von Methoden aus dem Software-Entwurf" (Dissertation), Paderborn/München; 1994

[Her89] Herold, H.: „ANSI C“ Programmieren portabler C-Programme im künftigen Sprachstandard „ANSI C“, te-wi Verlag, München 1989

[Hes92] Hesse, W.; u.a.: Softwareentwicklung: Vorgehensmodelle, Projektführung, Produktverwaltung, R. Oldenbourg Verla, München 1992

[Hog89] Hogrefe, D.: Estelle, Lotos und SDL, Standard-Spezifikationssprachen für verteilte Systeme, Springer Verlag 1989

[Hog92] Hogrefe, D.: Formale Beschreibungstechniken für verteilte Systeme, Springer Verlag 1992

[ifa96] ifak: Dokumentation der Estelle Beschreibung für einen PROFIBUS-DP Master (Klasse 1), ifak, Barleben 1996

[Lut93] Lutsch, Andreas: ‚Ein Parser für SDL"; Jahresarbeit Humboldt-Universität zu Berlin, Fachbereich Informatik; 1993

[NIS87] : NIST. User guide for the NBS prototyp compiler for Estelle. Bericht NIST Report No. ICST/SNA-87/3, 1987

[PET91] : The Portable Estelle Translator: an overview and user guide, Technical Report NCSL/SNA-91/2, Gaithersburg 1991

[Sch93] Schade, Andreas: ‚Eine Common Representation für SDL`92 Spezifikationen" Jahresarbeit Humboldt-Universität zu Berlin, Fachbereich Informatik; 1993

[Tur93] Turner, K. J.: Using Formal Description Techniques, John Wiley & Sons, Chichester 1993