PEARL90 - Programmieren mit Objekten

Eine Fallstudie

von L. Frevert

1 Einleitung

Die Echtzeitsprache PEARL (Process and Experiment Automation Realtime Language) ist etwa gleichzeitig mit PASCAL entwickelt worden und bot daher wie PASCAL keine Sprachelemente für objektorientierte Programmierung. Bei der letzten Überarbeitung der PEARL-Norm (DIN 66253) Anfang der neunziger Jahre wurden jedoch in PEARL90 die grundsätzlichen Möglichkeiten geschaffen, auch objektorientiert programmieren zu können. Den Schlüssel dazu bildet die Tatsache, daß in PEARL90 Referenzen nicht nur auf Daten, sondern jetzt auch auf Prozeduren, Tasks, Semaphore usw. "zeigen" können.

PEARL90 ist jedoch keine objektorientierte Sprache im engeren Sinne; es gibt keine speziellen Sprachmittel für Klassen, Objekte und Methoden. keine Klassenhierarchien und keine Vererbung. Klassen, Objekte und Methoden lassen sich jedoch mit Hilfe der vorhandenen Sprachmittel ausdrücken. Im folgenden wird dieser Programmierstil anhand eines längeren Beispiels erläutert. Damit soll auch gezeigt werden, daß die Programmierung mit Objekten wesentliche Vorteile bei der Koordination von Parallelarbeit und der Lösung von Echtzeit-Problemen bietet, wie z. B. die Überwachung von Rechenzeiten und Auslösung entsprechender Ausnahmebehandlungen, wenn Zeitlimits überschritten worden sind.

2 Grundsätzliches

2.1 Objektorientierte Programmierung

2.1.1 Strukturgesichtspunkte

Technische Anlagen und Geräte bestehen im allgemeinen aus einzelnen Funktionseinheiten bzw. Baugruppen, die zur Erfüllung des Gesamtzweckes zusammenarbeiten. Diese Aufteilung erleichtert Konstruktion, Herstellung und Wartung. Es ist plausibel, daß es bei der Simulation derartiger Anlagen vorteilhaft ist, wenn die verwendete Software eine ähnliche Struktur hat, so daß den zu simulierenden Objekten entsprechende Software-Objekte entsprechen. Aus diesem Ansatz heraus wurden die Programmiersprache SIMULA entwickelt, aus der viele Elemente heutiger objektorientierter Sprachen stammen.

Strukturelle Entsprechungen zwischen Realität und Software sind immer günstig; deshalb werden insbesondere graphische Darstellungen bzw. Bedienoberflächen, die sich aus Teilelementen zusammensetzen, heute in der Regel mit objektorientierten Sprachen programmiert.

Der Ansatz, Programme so zu gliedern wie eine technischen Anlage, ist selbstverständlich auch hilfreich, wenn eine Anlage automatisiert werden soll; dann entsprechen den real vorhandenen Teilen der Anlage spiegelbildlich entsprechende Objekte in der Automatisierungs-Software.

Programmiersprachen wie PEARL und PASCAL haben eine andere Aufteilung der Programme. Sie leiten sich von Vorläufern her, die für technisch-wissenschaftliche Rechnungen entwickelt worden waren. Dort werden Eingangsdaten in aufeinanderfolgenden Rechenschritten in Endergebnisse umgerechnet; wenn diese Rechenschritte kompliziert sind, werden dafür Prozeduren geschrieben. Man nennnt diese Programmiersprachen deshalb "prozedurale Sprachen": Die Programme bestehen im wesentlichen aus einer Datenbasis und Prozeduren, die auf dieser Datenbasis arbeiten.

Eine wesentliche Steigerung der Produktivität bei der Erzeugung von Software kann dadurch erreicht werden, daß bewährte Programmteile in neuen Programmen wiederverwendet werden; bei prozeduralen Sprachen versucht man deshalb, Bibliotheken zu entwickeln, die wiederverwendbare Prozeduren enthalten.

Bezüglich der Wiederverwendung von Programmteilen kann man leicht einsehen, daß die Wiederverwendbarkeit umso besser ist, je kleiner die Programmteile sind. Dann ergibt sich andererseits aber die Schwierigkeit, daß ein Programmierer viele Arten vorhandener Teilen kennen muß, damit er ein neues Programm aus ihnen zusammensetzen kann. Je kleiner die Programmteile sind, aus denen ein neues Programm zusammengesetzt wird, desto mehr Schnittstellen zwischen diesen Teilen enthält das Programm. Auf der Ebene der Maschinenbefehle bedeutet das zusätzlichen Programmcode und höhere Laufzeiten.

Die Erfahrung hat gezeigt, daß der Löwenanteil der Softwarekosten nicht beim Schreiben der Programme entsteht, sondern bei ihrer späteren Verbesserung aufgrund von Betriebserfahrungen. Die Aufteilung von Programmen in Prozeduren und Datenbasis führt jedoch zu Schwierigkeiten gerade bei dieser Programmwartung: Änderungen an der Datenbasis erfordern Änderungen in den Prozeduren. In vielen praktischen Fällen ist jedoch schwer zu übersehen, welche Prozeduren wie geändert werden müssen.

Deshalb wurde ein Programmierprinzip entwickelt, das unter den Schlagworten "Information Hiding", "Abstrakte Datentypen" und "Datenkapselung" bekannt ist. Es besteht darin, daß Daten und diejenigen Prozeduren, die diese Daten bearbeiten, so zusammengefaßt werden, daß auf die Daten nur noch mit Hilfe der Prozeduren zugegriffen werden kann. Die Daten sind nach außen unsichtbar und von den umgebenden Prozeduren eingekapselt. Dieser Ansatz führte letztendlich zur Entwicklung von objektorientierten Programmiersprachen.

2.1.2 Prinzipien der objektorientierten Programmierung

Bei der objektorientierten Programmierung hat man keine Aufteilung des Programmes in Daten und Prozeduren, sondern Daten und Verarbeitungsschritte werden jeweils zu Objekten zusammengefaßt. Das Programm besteht aus einem Netz von Objekten.

Oft hat man es mit Objekten zu tun, die einander sehr ähnlich sind und sich praktisch nur durch ihre Namen unterscheiden; diese gleichartigen Objekte gehören dann einer "Klasse" an. Verallgemeinert werden alle Objekte aus Klassen abgeleitet.

Die Verarbeitungsschritte, die die im Objekt eingekapselten Daten verändern oder den Zugriff auf diese Daten vermitteln, werden üblicherweise "Methoden" genannt.

Wirklich programmiert werden nur Klassen und ihre Methoden; die Objekte selbst werden unter Verwendung der Klassen deklariert bzw. dynamisch erzeugt. Bei Änderungen an einer Klasse werden automatisch auch alle Objekte dieser Klasse geändert. Das erspart vor allem Kosten bei der Wartung.

Einer der Vorteile des Programmierens mit Objekten liegt darin, daß die Objekte wohldefinierte gegenseitige Schnittstellen haben und daß Programmänderungen sich oft auf eine einzige Klasse begrenzen lassen, wenn die Schnittstellen unverändert bleiben können.

Eine weitere Vereinfachung der Programmierung entsteht dadurch, daß auch Klassen einander ähnlich sein können; diese Tatsache kann man dazu verwenden, Klassenhierarchien aufzubauen, indem ähnliche Klassen aus einer Superklasse abgeleitet werden. Solche Subklassen können dann von ihrer Superklasse Eigenschaften, z. B. Methoden, erben, wie Kinder Merkmale von ihren Eltern, Großeltern oder noch früher lebenden Vorfahren erben können.

Wie bei allen Techniken gibt es neben diesen Vorteilen auch Nachteile: Es erfordert eine Änderung der Programmiergewohnheiten, objektorientiert zu programmieren. Insbesondere die Gliederung der Programme in Klassen und Methoden und der Aufbau von Klassenhierarchien erfordert eine gewisse Erfahrung. Es ist zwar oft möglich, neue Anwendungsprogramme unter Benutzung schon vorhandener Klassen aufzubauen, aber das erfordert selbstverständlich eine genaue Kenntnis darüber, welche Klassen schon vorhanden sind und welche Eigenschaften sie haben.

2.2 Objektorientierung in PEARL90

2.2.1 Referenzvariable in PEARL90

Die Möglichkeit, in PEARL90 objektorientiert programmieren zu können, beruht auf der Verwendung von Referenz-Datentypen. Deshalb sollen zum besseren Verständnis der nachfolgenden Beispiele hier die Eigenschaften von Referenzen kurz dargestellt werden.

Eine einfache ganzzahlige Variable hat bekanntlich einen Namen und einen Inhalt, der auch Variablenwert genannt wird; durch

```
DCL fixedvariablenname FIXED INIT(5);
```

wird die Variable mit Namen `fixedvariablename` und Anfangswert `5` erzeugt.

Für Leser, die PEARL90 bisher nur oberflächlich kennen, dazu noch folgende Hinweise: In der Darstellung der Syntax werden die Begriffe “Bezeichner” und “Name” verwendet. Dabei entspricht “Bezeichner” etwa dem, was umgangssprachlich Name genannt wird: ein Bezeichner besteht meistens aus Buchstaben, darf aber auch Ziffern und den Unterstrich enthalten. Mit einem Bezeichner können auch Zusammenfassungen von mehreren Daten gleichen Typs (Felder) oder verschiedenen Typs (Verbunde) benannt sein. Ein “Name” hingegen vermittelt den Zugriff auf eine bestimmte Date aus einer derartigen Zusammenfassung; er darf auch gewisse Sonderzeichen, z. B. den Punkt bei Zugriff auf eine Komponente eines Verbundes, enthalten.

In vielen Fällen ist der Name identisch mit dem Bezeichner, wie z. B. bei `fixedvariablenname`. Bei Nennung eines Variablennamens wird meistens in Wirklichkeit der Wert gemeint, den die Variable augenblicklich hat. Man nennt diesen Schritt vom Namen zum Inhalt "implizite Dereferenzierung".

Eine Referenz-Variable wird unter Benutzung des Schlüsselwortes `REF` vereinbart, z. B. durch

```
DCL fixedreferenzname REF FIXED;.
```

Der Inhalt oder Wert von `fixedreferenz` kann bei dieser Vereinbarung nur der Name einer FIXED-Variablen sein, der ihr durch Zuweisung

```
fixedreferenzname := fixedvariablenname;
```

oder bei der Deklaration

```
DCL fixedreferenzname REF FIXED INIT(fixedvariablenname);
```

gegeben werden kann.

Referenz-Variable werden auch "Zeiger" genannt: Der Zeiger `fixedreferenzname` zeigt auf die Variable `fixedvariablenname`. Im Unterschied zu "echten" Zeigern wie z. B. in C kann man mit Referenz-Variablen keine arithmetischen Operationen durchführen, man darf also z. B. keine Referenz-Variablen voneinander subtrahieren.

Referenz-Variable sind in PEARL streng typgebunden. Sie können nur auf Programmbestandteile desjenigen Typs zeigen, der in ihrer Vereinbarung steht; für Sonderzwecke gibt es jedoch auch Referenzvariable, die auf jeden beliebigen Typ zeigen können, z. B. `alleszeiger` nach der Deklaration

```
DCL alleszeiger REF STRUCT[];.
```

Referenz-Variable können unmittelbar auf alle deklarierbare Programmbestandteile zeigen außer auf Referenz-Variable, also auch auf Tasks, Prozeduren, Semaphore usw..

Es gibt eine spezielle Referenz-Konstante `NIL`, die einen Null-Inhalt hat; sie zeigt sozusagen definiert ins Leere.

Mit den Operatoren `IS` und `ISNT` kann festgestellt werden, ob zwei Referenzvariable denselben Namen enthalten oder ob eine Referenz-Variable einen bestimmten Namen als Inhalt hat:

```
IF CONT fixedreferenzname IS fixedvariablenname...
```

Auf den Wert derjenigen Variablen, deren Name der Inhalt einer Referenz-Variablen ist, kann durch Dereferenzierung zugegriffen werden; durch

```
fixedreferenzname := fixedvariablenname;
```

und `CONT fixedreferenzname := 5;`

bekommt `fixedvariablenname` den Wert 5. Die Dereferenzierung kann in vielen Fällen auch implizit erfolgen, z. B. darf in

```
fixedvariablenname := CONT fixedreferenzname + 1;
IF CONT fixedreferenzname == 5...
IF CONT fixedreferenzname IS fixedvariablenname...
```

das Schlüsselwort `CONT` auch weggelassen werden. Es ist jedoch sehr empfehlenswert, mit diesem Schlüsselwort an allen zulässigen Stellen explizit zu dereferenzieren, weil dann Flüchtigkeitsfehler (Vergessen von `REF` in einer Vereinbarung) schon vom Kompilierer erkannt werden können.

Wenn eine Referenzvariable `p_referenz` auf den Verbund (die Struktur) `p` zeigt, der z. B. eine Komponente `wert` hat, kann auf diese Komponente außer direkt mit `p.wert` auch indirekt über die Referenzvariable mit `p_referenz.wert` zugegriffen werden; links vom Punkt wird auch hier implizit dereferenziert.

Referenz-Variable können wie einfache Variable per Wert oder per Namen (mit `IDENT`) an Prozeduren übergeben werden. Bei der Übergabe per Wert dürfen auch Namen von Variablen des richtigen Typs als aktuelle Parameter benutzt werden, weil Namen ja sozusagen die "Werte" von Referenzvariablen sind.

2.2.2 Von der Datenkapselung zum Objekt.

Bei Verwendung einer Echtzeitsprache mit der Möglichkeit der Parallelverarbeitung stellt es einen grundsätzlichen Fehler dar, wenn Daten so vereinbart werden, daß sie direkt von mehreren Programmteilen benutzt werden können. In PEARL sollte man daher z. B. nie einen Druck

```
DCL druck FLOAT GLOBAL;
```

vereinbaren. Dann können nämlich beim Programmlauf mehrere Tasks auch aus anderen Moduln auf `druck` zugreifen; falls das zufällig gleichzeitig geschieht, entstehen schwere Fehler. Deshalb müssen diese Zugriffe auf jeden Fall mit Hilfe von Semaphoren oder Bolts koordiniert werden. Ein Zugriff, um der Variablen einen neuen Wert zu geben, darf daher nur wie in Beispiel 1 erfolgen.

```
REQUEST druckzugriff;
druck:=neuerwert;
RELEASE druckzugriff;
```

Beispiel 1: Zugriff auf eine globale Variable mit Koordination mittels Semaphor

Bei einem größeren Programm ist schwer zu gewährleisten, daß alle derartigen Zugriffe unter Benutzung von Semaphor- oder Bolt-Anweisungen erfolgen. Das Beispiel 2 schafft hier Abhilfe; die Variable ist in einem Modul eingekapselt.

Die Variable `druckwert` ist von außerhalb des Moduls unsichtbar und kann in anderen Moduln nur durch Aufrufe der Prozeduren geändert bzw. gelesen werden. Um diese Aufrufe vernünftiger schreiben zu können, ist am Schluß des Moduls mit `DCL druck...` eine globale Struktur als Exportschnittstelle des Moduls vereinbart worden. Sie enthält Referenzen auf die Prozeduren und wird durch eine INIT-Klausel entsprechend initialisiert; so kann das Setzen des Druckes auf den Wert 5.0 z. B. `druck.setzen(5.);` geschrieben werden.

In einem PEARL-Programm, in dem mit vielen verschiedenen Drücken gearbeitet werden soll, müßte man für jeden Druck anhand eines Mustermoduls einen eigenen Modul erzeugen, der die Deklarationen der entsprechenden `FLOAT`- und `BOLT`-Variablen und die Zugriffsprozeduren enthält. Man bekäme so eine große Anzahl kleiner Moduln, die jeder z. B. die Funktionsprozedur `lesen` enthalten. Prinzipiell kann man alle Funktionsprozeduren `lesen` jedoch durch eine einzige ersetzen, indem man dieser einzigen den Namen des zu lesenden Druckes als Parameter übergibt.

```
MODULE (Druck_als_Objekt); PROBLEM;
 DCL druckwert FLOAT; ! aendern mit z.B. Druck.setzen(5.);
 DCL zugriff BOLT;
 setz:PROC (neuerdruck INV FLOAT);
    RESERVE zugriff;
    druckwert:=neuerdruck;
    FREE zugriff;
 END;
 lies:PROC RETURNS(FLOAT);
    DCL lokalerdruck FLOAT;
    ENTER zugriff;
     lokalerdruck:=druckwert;
    LEAVE zugriff;
    RETURN(lokalerdruck);
 END;
 DCL druck STRUCT[setzen INV REF PROC(FLOAT)
                  lesen INV REF PROC RETURNS(FLOAT)] GLOBAL INIT(setz,lies);
MODEND;
```

Beispiel 2: Druck als im Modul eingekapselte Variable

```
TYPE DRUCK STRUCT[wert FLOAT,
                  zugriff REF BOLT,
                  [setzen REF PROC(REF DRUCK,FLOAT),
                  lesen REF PROC(REF DRUCK) RETURNS(FLOAT)];
```

Beispiel 3: Typvereinbarung für die Klasse DRUCK

```
DCL kesseldruck_bolt BOLT;
DCL kesseldruck DRUCK INIT(0.0,
                           kesseldruck_bolt,
                           DRUCK_setzen,
                           DRUCK_lesen);
```

Beispiel 4: Deklaration eines Druckes und des zugehörigen Bolts,

Es ist daher besser, wenn man sich statt eines Mustermoduls einen entsprechenden Datentyp schafft. Beispiel 3 zeigt eine solche Typvereinbarung. Sie enthält selbstverständlich eine Komponente für den Druckwert. In einer Typvereinbarung dürfen in PEARL keine Bolts und Prozeduren stehen; deshalb enthält sie Referenzen darauf. Bei der Vereinbarung einer Variable des Typs `DRUCK` (Beispiel 4) werden diesen Referenzen in der INIT-Liste Anfangswerte gegeben. Der zugehörige Bolt muß zusätzlich zu jeder `DRUCK`-Variablen vereinbart werden, während die beiden Prozeduren für die Zugriffe auf den Druckwert nur einmal vorhanden zu sein brauchen. (Bei der PEARL90-Implementation von WERUM müssen Programmbestandteile im Programm vorher deklariert worden sein, wenn man ihre Bezeichner als Konstanten in INIT-Klauseln verwenden will.)

Beispiel 5 zeigt diese Zugriffsprozeduren. Sie bilden die Methoden der Klasse `DRUCK`. Ihnen werden als erster Parameter jeweils Referenzen auf diejenige Variable vom Typ DRUCK übergeben, auf die sie zugreifen sollen. Beispiel 6 zeigt dann Anweisungen zum Setzen bzw. Lesen des Druckwertes.

```
DRUCK_setzen:PROC(druck INV REF DRUCK,neuerdruck INV FLOAT);
   RESERVE druck.zugriff;
    druck.wert:=neuerdruck;
   FREE zugriff;
END;
DRUCK_lesen:PROC(druck INV REF DRUCK) RETURNS(FLOAT);
   DCL lokalerdruck FLOAT;
   ENTER druck.zugriff;
    lokalerdruck:=druck.wert;
   LEAVE druck.zugriff;
   RETURN(lokalerdruck);
END;
```

Beispiel 5: Methoden der Klasse `DRUCK`

```
kesseldruck.setzen(kesseldruck,5.0); ! Setzen des Druckes
neuerdruck:=kesseldruck.lesen(kesseldruck);
```

Beispiel 6: Anweisungen zum Setzen und Lesen eines Druckes

Gegenüber der Ausgangslage, bei der für jeden Druck ein eigener Modul verwendet wurde, hat die Verwendung des Datentyps `DRUCK` aus Beispiel 3 übrigens den Nachteil, daß auf die Komponente `wert` auch direkt mit `kesseldruck.wert:=5.0;` ohne Benutzung der Zugriffsprozeduren zugegriffen werden kann. In objektorientierten Sprachen können deshalb derartige "private" Bestandteile der Objekte durch entsprechende Sprachelemente geschützt werden.

2.2.3 Konventionen bei Benennungen

Die Definition des Datentyps und der Zugriffsprozeduren (Beispiele 3 und 4) entsprechen der Vereinbarung einer "Klasse". Die Zugriffsprozeduren entsprechen dabei den "Methoden" der Klasse. Die Deklarationen in Beispiel 4 entsprechen der Deklaration eines "Objektes".

Bei den Benennungen in den Beispielen sind folgende Konventionen befolgt worden:

1. Typ- bzw. Klassenbezeichner sind mit Großbuchstaben geschrieben,
2. Bezeichner von Typkomponenten sind mit Kleinbuchstaben geschrieben,
3. Prozedur- bzw. Methodenbezeichner bestehen aus dem Klassenbezeichner, an den die kleingeschriebene Benennung der Methode mittels Unterstrich angehängt ist,
4. das zu behandelnde Objekt bildet den ersten Parameter der Methode
5. Objektbezeichner sind mit Kleinbuchstaben geschrieben,

6. Bezeichner von Bolts, Semaphoren usw. sind ebenfalls mit Kleinbuchstaben geschrieben; der Komponententyp wird mittels Unterstrich an den Objektbezeichner angehängt.

3 Grobentwurf des Programmes

3.1 Motive bei der Aufgabenwahl

Die Aufgabenstellung, nämlich die Entwicklung eines Programmes für die Steuerung der Lichtsignalanlage (Verkehrsampelanlage) einer Straßenkreuzung, wurde gewählt, weil die Funktionsweise einer derartigen Anlage relativ leicht mit alphanumerischer Ausgabe auf einem Farbbildschirm dargestellt werden kann. In einem Praktikum braucht deshalb kein Anlagenmodell vorhanden zu sein, das Programm ist auch auf Rechnern einsetzbar, die keine Prozeßperipherie mit zugehörigen Treibern haben. Andererseits ist die Aufgabenstellung doch so anspruchsvoll, daß das Programm ein gutes Beispiel für die Anwendung der Sprachmittel von PEARL90 darstellt. Insbesondere wurden die Überwachung von Rechenzeiten und die Behandlung von Ausnahmen (Exceptions) im Laufe der Programmentwicklung eingebaut, um zu zeigen, daß sich derartige Probleme bei objektorientierter Programmierung mit PEARL90 relativ leicht lösen lassen.

3.2 Grobspezifikation: Adaptive Verkehrsregelung

Die Ampelanlage einer Straßenkreuzung soll von einem Prozeßrechner gesteuert werden.

Vor den Ampeln befinden sich in passenden Abständen Sensoren (Induktionsschleifen) in der Straßendecke, die im Rechner Interrupts auslösen, wenn sie von einem Fahrzeug passiert werden. Beide Induktionsschleifen einer Straße sind auf einen Interrupt zusammengeschaltet. Diese Interrupts führen zur Schaltung von Grünphasen.

Insgesamt sollen folgende Betriebsarten existieren:

- Ruhezustand: die Anlage ist ausgeschaltet,
- Gelbblinken: sämtliche Ampeln der Anlage blinken gelb,
- Normalzustand: die Anlage regelt den Verkehr adaptiv.

Im Normalzustand soll sich die Länge der Grünphasen dem Verkehrsaufkommen der beiden Straßen automatisch anpassen:

- Die Ampeln sind bei Normalbetrieb ohne Fahrzeugverkehr alle Rot geschaltet.
- Falls sich die Anlage im Normalzustand befindet, löst ein Interrupt die Grünphase der betreffenden Straße aus.

- Falls eine Straße bereits Grün hat und ein Interrupt für die andere Straße in Querrichtung ausgelöst wird, muß abgewartet werden, bis für die erste Straße wieder Rot geschaltet worden ist. Nach einer kurzen Überlappzeit, in der beide Straßen Rot haben, wird dann für die andere Straße Grün geschaltet.
- Ein einzelnes Fahrzeug bekommt eine Grünphase von festgelegter Dauer. Wenn die Ampel Grün geschaltet ist und ein weiteres Fahrzeug sich der Ampel nähert und dabei einen Interrupt auslöst, wird die Grünphase verlängert.
- Die eben beschriebene Verlängerung der Grünphase erfolgt, bis ein Maximalwert erreicht ist.

Der Übergang vom Ruhezustand in den Normalzustand soll folgendermaßen erfolgen:

- Alle Ampeln blinken gewisse Zeit gelb, danach
- zeigen alle Ampeln kurze Zeit Gelb, danach
- wird mit Rot auf den Normalzustand geschaltet.

Beim Ausschalten der Anlage erfolgt der Übergang vom Normalzustand in den Ruhezustand durch Umkehrung des vorstehenden Ablaufs.

Für den Programmtest wird die Ampelanlage simuliert: Der Schaltzustand der Ampeln wird auf einem Farbbildschirm durch Ausgabe von farbigen Leerzeichen dargestellt.

Die Bedienung des Programmes erfolgt durch hierarchische Bediendialoge: Ausgabe von Auswahlmenüs, Eingabe der Nummer des betreffenden Menüpunktes, Verzweigung in Untermenüs bzw. Ausführung der gewünschten Programmfunktion.

Die Ausgabe der Auswahlmenüs wird durch den UNIX-Interrupt Control-C ausgelöst.

Eines der Auswahlmenüs dient zur Simulation der Sensor-Interrupts: Anwahl der zum simulierten Interrupt gehörenden Straße.

3.3 Klassen und Klassenbestandteile

3.3.1 Benötigte Klassen

Auf den ersten Blick werden folgende Klassen von Objekten benötigt:

- `KREUZUNG`: jede Kreuzung besteht aus zwei Straßen.
- `STRASSE`: jede Straße umfaßt zwei Ampeln und ein Ereignis (Interrupt), das durch die zur Straße gehörenden Sensoren ausgelöst wird.
- `AMPEL`: jede Ampel besteht aus der zugehörigen Hardware und der graphischen Darstellung auf dem Bildschirm.
- `AMPELHARDWARE`: programmiertechnisch Prozeßdatenstation zur Ausgabe eines Bitmusters, das die Ampelleuchten an- bzw. ausschaltet.
- `AMPELBILD`: farbige graphische Darstellung der jeweiligen Ampel mit entsprechender Positionierung.

- `EREIGNIS`: programmtechnisch ein Interrupt.

Dabei gilt für alle diese Klassen, daß die zugehörigen Objekte eine Schachtelungs-Hierarchie bilden: Straßen sind in Kreuzungen enthalten, Ampeln sind Bestandteile von Straßen. Außerhalb dieser Klassen muß je ein Objekt der folgenden Klasse existieren:

- `TERMINAL`: es besteht aus Datenstationen für Ausgabe bzw. Eingabe und dient zur Ausgabe der Ampelbilder und zur Durchführung von Bediendialogen.
- `DIALOG`: er vermittelt die Durchführung von Bediendialogen und bewirkt die Verzweigung in entsprechende Programmteile.

Bei genauerer Überlegung muß zu den Kreuzungen noch die Fläche hinzugefügt werden, auf der sich die Straßen überschneiden. Bevor beide Ampeln einer Straße Grün geschaltet werden, muß diese Fläche exklusiv in den Besitz der betreffenden Straße übergehen, um gleichzeitiges Grün für beide Straßen zu vermeiden.

Zu Jeder Straße müssen außerdem zwei Steuerungen gehören, eine für die Ausführung des Blinkens und eine für den oben definierten Normalbetrieb. Da jedoch die Blinker bzw. Steuerungen der beiden Straßen einer Kreuzung nur gemeinsam ein- und ausgeschaltet werden dürfen, werden Objekte benötigt, die dies für eine gesamte Kreuzung erledigen. Es ergeben sich also folgende zusätzlichen Objekte:

- `FLAECHE`: die Fläche einer Kreuzung, die beiden sich kreuzenden Straßen gemeinsam ist.
- `BLINKER`: für Start, Durchführung und Beenden des Blinkens der beiden Ampeln einer Straße.
- `ADAPTSTEUERUNG`: adaptive Steuerung der Ampelphasen einer Straße.
- `KREUZBLINKER`: für Start, Durchführung und Beenden des gemeinsamen Blinkens der vier Ampeln der sich kreuzenden Straßen.
- `KREUZSTEUERUNG`: für Start, Durchführung und Beenden der adaptiven Steuerung der sich kreuzenden Straßen.

Im Laufe der Programmentwicklung kamen dann noch folgende Klassen hinzu:

- `ZYKLUS`: die Objekte führen Aufträge (Methoden) eines Auftraggebers als Parallelarbeit immer wieder aus.
- `UEBERWACHER`: die Objekte führen Aufträge (Methoden) eines Auftraggebers aus und überwachen die Ausführungszeit; bei Überschreitung des Zeitlimits wird der Auftrag abgebrochen und eine Fehlersituation signalisiert.
- `AUSLOESER`: die Objekte warten zyklisch auf ein Ereignis und führen dann einen Auftrag für einen Auftraggeber aus.
- `AUSLOESERBEENDER`: die Objekte sind Bestandteile einer `ADAPTSTEUERUNG`; sie warten zyklisch auf ein Ereignis und lösen einen Ampelzyklus aus bzw. beenden ihn.

3.3.2 Beispiele für Klassen und deren Methoden

Die Objekte der Klasse `AMPELHARDWARE` müssen als Kern eine Prozeßdatenstation enthalten, die im Systemteil benannt und im Problemteil spezifiziert werden muß; die Objekte müssen eine Referenz auf diese Prozeßdatenstation enthalten, die zum Test vorläufig NIL gesetzt werden kann. Wie alle Datenstationen muß auch diese eröffnet werden, bevor sie zur Ausgabe einer Bitkette zum Schalten der Leuchten einer Ampel benutzt werden kann. Diese Eröffnung soll bei Ausführung der Methode `init` durchgeführt werden. Die eigentliche Ausgabe erfolgt dann durch die Methode `steuern`. Beispiel 7 zeigt die Vereinbarungen für die Klasse `AMPELHARDWARE`.

```
TYPE AMPELHARDWARE STRUCT[dation INV REF DATION OUT BASIC,
                         init INV REF PROC(INV REF INV AMPELHARDWARE),
                         steuern INV REF PROC(INV REF INV AMPELHARDWARE,
                                              INV BIT(3))];
  AMPELHARDWARE_init: PROC(ampelhardware INV REF INV AMPELHARDWARE );
    IF CONT ampelhardware.dation IS NIL THEN
       RETURN;
     ELSE
       OPEN ampelhardware.dation;
    FIN;
  END;
  AMPELHARDWARE_steuern:PROC(ampelhardware INV REF INV AMPELHARDWARE,
                             bitmuster INV BIT(3));
    IF CONT ampelhardware.dation IS NIL THEN
       RETURN;
      ELSE
       SEND bitmuster TO ampelhardware.dation;
     FIN;
  END;
```

Beispiel 7: Vereinbarung der Klasse `AMPELHARDWARE`

```
  KREUZUNG_init:PROC(kreuzung INV REF KREUZUNG);
      kreuzung.erstestrasse.init(CONT kreuzung.erstestrasse,
                                 CONT kreuzung);
      kreuzung.zweitestrasse.init(CONT kreuzung.zweitestrasse,
                                  CONT kreuzung);
      kreuzung.flaeche.init(CONT kreuzung.flaeche);
      kreuzung.blinker.init(CONT kreuzung.blinker,kreuzung);
      kreuzung.steuerung.init(CONT kreuzung.steuerung,kreuzung);
  END;
```

Beispiel 8: Methode zur Initialisierung einer Kreuzung

Die Methode `AMPELHARDWARE_init` muß durchgeführt werden, bevor die Ampelanlage einer Kreuzung in Betrieb geht. Da die Objekte der Klasse `AMPELHARDWARE` indirekt Bestandteile der Kreuzung sind, ist es offensichtlich notwendig, daß auch die Klasse `KREUZUNG` eine Methode `init`

besitzt, und damit auch die Klassen `STRASSE` und `AMPEL`. Kurz, bei fast allen Klassen muß eine Methode `init` vorhanden sein; bei der Klasse `AMPELBILD` wird dabei beispielsweise das Bild einer ausgeschalteten Ampel dargestellt. Beispiel 8 zeigt die Initialisierungs-Methode für die Klasse `KREUZUNG`, Beispiel 9 die Initialisierung für `STRASSE`. Den Straßen wird dabei mitgeteilt, zu welcher Kreuzung sie gehören.

```
STRASSE_init:PROC(strasse INV REF STRASSE,
                  kreuzung INV REF KREUZUNG);
   strasse.hin.init(CONT strasse.hin);
   strasse.her.init(CONT strasse.her);
   strasse.blinker.init(CONT strasse.blinker,strasse);
   strasse.steuerung.init(CONT strasse.steuerung,strasse);
   strasse.kreuzung:=CONT kreuzung;
END;
```

Beispiel 9: Methode zur Initialisierung einer Straße

Man erkennt an diesen Beispielen, daß bei der gegebenen Schachtelungsstruktur der Klassen nur auf der untersten Ebene der Schachtelung Anweisungen auszuführen sind, die dem eigentlichen Programmzweck dienen, z. B. `OPEN ampelhardware.dation;` zur Eröffnung der Prozeßdatenstation. Bei allen darüber liegenden Ebenen werden in der init-Methode die init-Methoden der eingeschachtelten Klassen als Prozeduren aufgerufen. Dadurch entsteht selbstverständlich auf der Ebene der Maschinenbefehle ein nicht unbeträchtlicher Zusatzaufwand an Codelänge und Ausführungszeit.

```
STRASSE_gruenschalten:PROC(strasse INV REF STRASSE);
  strasse.schaltstatus:='1'B;
  strasse.ampelstatus:=ROTGELBMUSTER;
  strasse.hin.stellen(CONT strasse.hin,strasse.ampelstatus);
  strasse.her.stellen(CONT strasse.her,strasse.ampelstatus);
  AFTER strasse.rotgelbdauer RESUME;
  strasse.ampelstatus:=GRUENMUSTER;
  strasse.hin.stellen(CONT strasse.hin,strasse.ampelstatus);
  strasse.her.stellen(CONT strasse.her,strasse.ampelstatus);
  strasse.schaltstatus:='0'B;
END;
```

Beispiel 10: Methode zum Grünschalten einer Straße

Beispiel 10 zeigt das Grünschalten einer Straße. Dabei wird zunächst auf Rot-Gelb geschaltet und eine vorgegebene Zeit gewartet. Die Bezeichner `ROTGELBMUSTER` usw. stehen für entsprechend definierte BIT(3)-Konstanten, `rotgelbdauer` ist eine Komponente von `STRASSE`, deren DURATION-Wert bei der Initialisierung eines Objektes des Typs `STRASSE` eingesetzt wird, `hin` und `her` sind die beiden Ampeln. Die zugehörige Typvereinbarung ist in Beispiel 11 enthalten.

```
TYPE STRASSE STRUCT[
     ampelstatus BIT(3),
     (schaltstatus,
     blinkstatus,
     steuerungsstatus) BIT(1),
     (gelbdauer,rotgelbdauer,blinkdauer,
      rotueberlapp,gruendauer,maxgruendauer) DUR,
     (hin,her) INV REF AMPEL,
     kreuzung REF KREUZUNG,
     induktschleife INV REF EREIGNIS,
     init INV REF PROC(INV REF STRASSE, INV REF FLAECHE),
     (rotschalten,
     gruenschalten,
     gelbschalten,
     ausschalten) INV REF PROC(INV REF STRASSE),
     blinker INV REF BLINKER,
     steuerung INV REF ADAPTSTEUERUNG];
```

Beispiel 11: Typvereinbarung für die Klasse STRASSE

3.3.3 Objekte für Nebenläufigkeit

Die bisher erwähnten Methoden haben Ausführungszeiten, die höchstens im Sekundenbereich liegen, wie z. B. das Grünschalten der Ampeln einer Straße mit einer Wartezeit bei Rotgelb. Das Blinken hingegen soll unter Umständen stundenlang erfolgen; aus diesem Grunde erfordert es eine eigene Task, die zeitlich parallel zum übrigen Programm läuft. Sie muß die Ampeln abwechselnd gelb- und ausschalten, mit einer Wartezeit im Sekundenbereich zwischen den Umschaltungen. Bei herkömmlicher Programmierung würde diese Task eine Wiederholung wie in Beispiel 12 enthalten. Zum Beenden des Blinkens müßte das Steuerbit '0'B gesetzt werden.

```
WHILE CONT steuerbit REPEAT
  AFTER blinker.strasse.blinkdauer RESUME;
  blinker.strasse.gelbschalten(CONT blinker.strasse);
  AFTER blinker.strasse.blinkdauer RESUME;
  blinker.strasse.ausschalten(CONT blinker.strasse);
END;
```

Beispiel 12: Widerholungsblock zur Durchführung des Blinkens

Derartige Tasks, die ähnliche Wiederholungsblöcke enthalten, treten auch bei der Programmierung des Rot-Grün-Rot-Zyklus auf. Es liegt daher nahe, für diese Zwecke eine Klasse `ZYKLUS` zu schaffen. Beispiel 13 zeigt die zugehörige Typvereinbarung. In dieser Typvereinbarung ist wegen `auftraggeber REF STRUCT[]` der Typ des Auftraggebers nicht festgelegt worden; deshalb kann ein Zyklus für jeden beliebigen Auftraggeber tätig werden.

```
TYPE ZYKLUS STRUCT[
            init INV REF PROC(INV REF ZYKLUS,
                              REF STRUCT[],
                              REF PROC(INV REF STRUCT[],INV REF BIT(1))),
            start INV REF PROC(INV REF ZYKLUS),
            stop INV REF PROC(INV REF ZYKLUS),
            beendet INV REF PROC(INV REF ZYKLUS),
            auftraggeber REF STRUCT[],
            auftrag REF PROC(INV REF STRUCT[],INV REF BIT(1)),
            auftragsteuerbit BIT(1),
            task INV REF TASK,
            endesema INV REF SEMA];
```

Beispiel 13: Typvereinbarung zur Klasse `ZYKLUS`

Beispiel 14 enthält die Vereinbarungen für ein Objekt dieser Klasse. Die Task `zyklusname_task`, die zum Objekt gehört, ruft mittels der Prozedur-Referenz `zyklusname.auftrag` eine Prozedur auf, der der Name eines Auftraggebers und des Auftragsteuerbits übergeben werden. Der Name der aufgerufenen Prozedur steht dabei in der Komponente `auftrag` des Objektes.

```
  zyklusname_task:TASK;
    zyklusname.auftrag(zyklusname.auftraggeber,
                       zyklusname.auftragsteuerbit);
    RELEASE zyklusname.endesema;
  END;
  DCL zyklusname_endesema SEMA;
  DCL zyklusname ZYKLUS INIT(ZYKLUS_init,
                             ZYKLUS_start,
                             ZYKLUS_stop,
                             ZYKLUS_beendet,
                             NIL,
                             NIL,
                             '0'B,
                             zyklusname_task,
                             zyklusname_endesema);
```

Beispiel 14: Objektvereinbarung zur Klasse `ZYKLUS`

Beispiel 15 zeigt die zu einem derartigen Objekt gehörenden Methoden bzw. Prozeduren. Bei der Initialisierung `ZYKLUS_init` werden dem jeweiligen Objekt die Namen von Auftraggeber und Autrag übergeben.

Wenn der Auftrag eines derartige Objektes durch `stop` beendet werden soll, läuft er im allgemeinen noch etwas weiter. Deshalb kann mit der Methode `beendet` abgewartet werden, bis der Auftrag wirklich beendet ist. Diese Synchronisierung wird mittels des Semaphors `endesema` vorgenommen. Er wird in der Methode `ZYKLUS_init` mit Hilfe des TRY-Operators auf seinen Anfangswert gesetzt, um einen Wiederstart des Programmes zu erleichtern; derart wird auch bei der Initialisierung der übrigen Semaphore verfahren.

```
ZYKLUS_init:PROC(zyklus INV REF ZYKLUS,
                 auftraggeber INV REF STRUCT[],
                 auftrag INV REF PROC(INV REF INV STRUCT[],
                 INV REF BIT(1)));
   zyklus.auftraggeber:=CONT auftraggeber;
   zyklus.auftrag:=auftrag;
   zyklus.auftragsteuerbit:='0'B;
   WHILE TRY zyklus.endesema REPEAT END;
END;
ZYKLUS_start : PROC(zyklus INV REF ZYKLUS );
  zyklus.auftragsteuerbit:='1'B;
  ACTIVATE zyklus.task;
END;
ZYKLUS_stop : PROC(zyklus INV REF ZYKLUS);
  zyklus.auftragsteuerbit:='0'B;
END;
ZYKLUS_beendet : PROC(zyklus INV REF ZYKLUS);
  REQUEST zyklus.endesema;
END;
```

Beispiel 15: Methoden der Klasse `ZYKLUS`

Im Falle des Blinkens ist ein Objekt der Klasse `ZYKLUS` ein Bestandteil jedes Blinkers. Beispiel 16 zeigt die Typvereinbarung, Beispiel 17 die Methoden dieser Klasse.

```
 TYPE BLINKER STRUCT[
        init INV REF PROC(INV REF BLINKER,INV REF STRASSE),
        (start,
        stop,
        beendet) INV REF PROC(INV REF BLINKER),
        strasse REF STRASSE,
        zyklus INV REF ZYKLUS,
        blinken INV REF PROC(INV REF BLINKER,INV REF BIT(1))];
```

Beispiel 16: Typvereinbarung zur Klasse `BLINKER`

Bei der Initialisierung wird dem zum Blinker gehörenden Zyklus der Name des Blinkers und der Auftrag `blinker.blinken` übergeben.

In `blinker.blinken` wird auch auf diejenige Straße zugegriffen, von der der Blinker ein Teil ist. Deshalb wird dem Blinker deren Name auch bei dessen Initialisierung übergeben; der Aufruf von `BLINKER_init` wiederum erfolgt in der Initialisierung der Straße.

Wie schon erwähnt, werden bei der Initialisierung eines Objektes alle Objekte initialisiert, die in das Objekt eingeschachtelt sind. Dabei wird, wie im Beispiel 17, häufig auch der Name des umschließenden Objektes an das eingeschachtelte Objekt übergeben und dort in einer Referenz-Variablen gespeichert.

```
BLINKER_init:PROC(blinker INV REF BLINKER,
                  strasse INV REF STRASSE);
      blinker.strasse:=CONT strasse;
      blinker.zyklus.init(CONT blinker.zyklus,
                          CONT blinker,
                          blinker.blinken);
END;
BLINKER_blinken:PROC(blinker INV REF BLINKER,steuerbit INV REF BIT(1));
   WHILE CONT steuerbit REPEAT
     AFTER blinker.strasse.blinkdauer RESUME;
     blinker.strasse.gelbschalten(CONT blinker.strasse);
     AFTER blinker.strasse.blinkdauer RESUME;
     blinker.strasse.ausschalten(CONT blinker.strasse);
   END;
END;
BLINKER_start:PROC(blinker INV REF BLINKER);
   blinker.zyklus.start(CONT blinker.zyklus);
END;
BLINKER_stop:PROC(blinker INV REF BLINKER);
  blinker.zyklus.stop(CONT blinker.zyklus);
END;
BLINKER_beendet:PROC(blinker INV REF BLINKER);
  blinker.zyklus.beendet(CONT blinker.zyklus);
END;
```

Beispiel 17: Methoden zur Klasse `BLINKER`

3.4 Vernetzung der Objekte

Durch Ausfüllen der INIT-Listen bzw. bei der Ausführung der init-Methoden werden die Objekte durch die Referenz-Variablen miteinander verzeigert; in jedem Objekt wird auf die Objekte verwiesen, aus denen es zusammengesetzt ist, in vielen Fällen auch auf das umgebende Objekt. Dadurch entsteht bei einer Kreuzung zunächst eine Baumstruktur: das Objekt Kreuzung ist die Wurzel, deren Bestandteile - zwei Straßen, eine Fläche usw. - sind die Unterknoten der ersten Ebene, die Straßen besitzen als Unterknoten Ampeln usw., und so fort.

```
TYPE KREUZUNG STRUCT[
     (erstestrasse,zweitestrasse) INV REF STRASSE,
     flaeche INV REF FLAECHE,
     init INV REF PROC(INV REF KREUZUNG),
     blinker INV REF KREUZBLINKER,
     steuerung INV REF KREUZSTEUERUNG];
```

Beispiel 18: Typvereinbarung zur Klasse KREUZUNG

In der Kreuzung kreuzen sich zwei Straßen auf einer Fläche, die Teil sowohl der einen als auch der anderen Straße ist. Die Ampelsteuerung soll bewirken, daß diese Fläche nur von Autos befahren werden, die sich ihr auf einer der beiden Straßen nähern. Die Kreuzungsfläche muß deshalb einerseits Bestandteil der Kreuzung sein (Beispiel 18), andererseits aber auch von den Straßen benutzt werden können. Deshalb muß die jeweilige Straße einen Verweis auf die Kreuzung enthalten, zu der sie gehört (Beispiel 11).

Wie schon erwähnt, werden die Ampelbilder mit einem Objekt der Klasse `TERMINAL` dargestellt. Die Typvereinbarung von `AMPELBILD` enthält deshalb einen Verweis auf ein Objekt der Klasse `TERMINAL` (Beispiel 19), das heißt, in alle vier Ampelbild-Objekte einer Straße werden bei der Vereinbarung Verweise auf das einzige Terminal-Objekt des Programmes eingetragen. Letztendlich sind die Objekte des Gesamtprogrammes zu einem Netz verknüpft.

Verknüpfungen in Vorwärtsrichtung erfolgen beim Schreiben des Programmes durch Ausfüllen der `INIT`-Listen; in Fällen, wo auch Rückwärtsverknüpfungen notwendig sind, werden diese beim Programmlauf während der Initialisierung der Objekte vorgenommen, um Schreibarbeit und dadurch mögliche Fehler zu sparen.

```
TYPE AMPELBILD STRUCT[geometrie GEOMETRIE,
                      status BIT(3),
                      terminal INV REF TERMINAL,
                      init INV REF PROC(INV  REF INV AMPELBILD),
                      male INV REF PROC(INV REF INV AMPELBILD,
                                        INV REF INV AMPELSTATUS)];
```

Beispiel 19: Typvereinbarung zur Klasse `AMPELBILD` (erster Entwurf)

Verknüpfungen in Vorwärtsrichtung erfolgen beim Schreiben des Programmes durch Ausfüllen der `INIT`-Listen; in Fällen, wo auch Rückwärtsverknüpfungen notwendig sind, werden diese beim Programmlauf während der Initialisierung der Objekte vorgenommen, um Schreibarbeit und dadurch mögliche Fehler zu sparen.

4 Programmdetails

4.1 Behandlung der Kreuzungsfläche

Bevor die Ampeln einer Straße auf Grün geschaltet werden können, muß sichergestellt sein, daß die Ampeln der kreuzenden Straße Rot geschaltet sind. Gewährleisten kann man das dadurch, daß die Kreuzungsfläche der jeweiligen Straße exklusiv zugeteilt wird, bevor die Grünschaltung erfolgt. Diesem Zweck dient die Methode `FLAECHE_besorgen`; in ihr wird eventuell auf einer REQUEST-Anweisung gewartet, bis `FLAECHE_freigeben` durchgeführt worden ist.

Beispiel 20 zeigt die Vereinbarungen und Methoden zur Klasse `FLAECHE`. Die Typvereinbarung enthält eine Referenz auf einen Semaphor, der zusammen mit einer Variablen des Typs Fläche vereinbart werden muß (Beispiel 21). In der Methode `init` ist Vorsorge getroffen, daß der Semaphor auch nach Abbruch und Neustart der betreffenden Programmteile jedenfalls den Wert Eins bekommt.

```
TYPE FLAECHE STRUCT[
             (init,
              besorgen,
              freigeben) INV REF PROC(INV REF FLAECHE),
              exclusiv INV REF SEMA];
 FLAECHE_init:PROC(flaeche INV REF FLAECHE);
     WHILE TRY CONT flaeche.exclusiv REPEAT END;
     RELEASE flaeche.exclusiv;
 END;
 FLAECHE_besorgen:PROC(flaeche INV REF FLAECHE);
     REQUEST flaeche.exclusiv;
 END;
 FLAECHE_freigeben:PROC(flaeche INV REF FLAECHE);
     RELEASE flaeche.exclusiv;
 END;
```

Beispiel 20: Typvereinbarung und Methoden zur Klasse `FLAECHE`

```
DCL kreuzung_flaeche_exclusiv SEMA;
DCL kreuzung_flaeche FLAECHE INIT(
      FLAECHE_init,
      FLAECHE_besorgen,
      FLAECHE_freigeben,
      kreuzung_flaeche_exclusiv);
```

Beispiel 21: Vereinbarung eines Objektes zur Klasse `FLAECHE`

4.2 Die Steuerung der Ampelphasen

4.2.1 Zwei kooperierende Endlostasks

Bei jeder Straße übernimmt ein Objekt der Klasse `ADAPTSTEUERUNG` die Steuerung der Ampelphasen. Es soll die Ampelanlage einschalten und dann auf die Auslösung von Interrupts durch die Induktionsschleifen warten; dann soll zyklisch jeweils von Rot auf Grün und wieder zurück auf Rot geschaltet werden.

Falls die Notwendigkeit einer Grünschaltung durch die Auslösung eines Interrupts erkannt wird, muß jedoch eventuell gewartet werden, bis die Grünphase für die andere Straße beendet ist. Interrupts, die während dieser Wartezeit eintreffen, dürfen keine Reaktion des Programmes hervorrufen.

Interrupts, die sich während der Grünphase ihrer Straße ereignen, sollen die Grünphase verlängern, jedoch nur bis zu einer Maximaldauer; deshalb dürfen Interrupts in einem Zeitintervall am Ende der Grünphase ebenfalls nichts bewirken.

Neben dem Rot-Grün-Rot-Zyklus muß deshalb ein zweiter Zyklus existieren, wo ein Interrupt abwechselnd irgendwelche Maßnahmen oder gar nichts bewirkt. Diese beiden Zyklen wirken zwar gegenseitig aufeinander ein, haben aber unterschiedliche Zeitpunkte für die Umschaltung ihrer Zustände.

Jede Steuerung enthält deshalb ein Objekt der Klasse `AUSLOESERBEENDER`, in dem auf die Interrupts gewartet wird. Es löst dann die Grünphasen aus und beendet sie. Die Interrupts werden dabei durch Objekte der Klasse `EREIGNIS` vertreten.

Insgesamt müssen für die Steuerung vier Zustände unterschieden werden:

- `NORMALROT`: Die Ampeln der Straße sind Rot, es wird auf ein erstes Fahrzeug gewartet,
- `GRUENGEFORDERT`: Die Ampeln der Straße sind Rot, ein Fahrzeug hat einen Interrupt ausgelöst, es kann aber eventuell noch nicht Grün geschaltet werden, weil die andere Straße Grün hat,
- `GRUENVERLAENGERN`: Die Straße hat Grün, weitere Fahrzeuge bewirken eine Verlängerung der Grünphase,
- `GRUENENDE`: Die Grünphase geht ihrer Maximaldauer entgegen, weitere Fahrzeuge bewirken keine Verlängerung der Grünphase.

```
TYPE ADAPTSTEUERUNG STRUCT[
              (anfangsblinkdauer,
              anfangsgelbdauer,
              endblinkdauer,
              endgelbdauer) DUR,
              init INV REF PROC(INV REF ADAPTSTEUERUNG,
                                INV REF ADAPTSTRASSE),
              (start,
              stop,
              beendet) INV REF PROC(INV REF ADAPTSTEUERUNG),
              rotgruenzyklus INV REF PROC(INV REF ADAPTSTEUERUNG,
                                INV REF BIT(1)),
              status REF BIT(1),
              zyklus INV REF ZYKLUS,
              ausloeserbeender INV REF AUSLOESERBEENDER,
              strasse REF ADAPTSTRASSE];
```

Beispiel 22: Typvereinbarung zur Klasse `ADAPTSTEUERUNG`

Im Programm sind diese Zustände als FIXED-Konstanten deklariert. In den Zuständen `GRUENGEFORDERT` und `GRUENENDE` dürfen Induktionsschleifen-Interrupts keine Auswirkungen auf die Steuerung haben.

Beispiele 22 und 23 zeigen für die Klassen `ADAPTSTEUERUNG` bzw. `AUSLOESERBEENDER` die jeweilige Typvereinbarung. Beide enthalten unter anderem ein Objekt der Klasse `ZYKLUS`, dem die jeweilige

Umschaltmethode (`rotgruenzyklus` bzw. `wartezyklus`) als auszuführender Auftrag übergeben wird.

Die beiden zyklischen Umschaltmethoden koordinieren sich über die gemeinsame Variable `schaltstatus` und die Semaphore `statusschutz` und `ausloesung`. Sie sind Bestandteile des Ausloeserbeenders; auf sie wird zugegriffen in dessen Methoden `abwarten`, `ausloesen` und `gruenschalten`, die die Koordination der beteiligten Tasks bewirken.

```
TYPE AUSLOESERBEENDER STRUCT[
      init INV REF PROC(INV REF AUSLOESERBEENDER,
                        INV REF ADAPTSTEUERUNG),
      statusaendern INV REF PROC(INV REF AUSLOESERBEENDER,
                                 status INV FIXED),
      (gruenbeenden,start,stop,beendet,abwarten,ausloesen,gruenschalten)
           INV REF PROC(INV REF AUSLOESERBEENDER),
      eingeschaltet BIT(1),
      gruenschalttask REF TASK,
      zyklus REF ZYKLUS,
      wartezyklus REF PROC(INV REF AUSLOESERBEENDER,
                           INV REF BIT(1)),
      weitertask INV REF TASK,
      ereignis REF EREIGNIS,
      gruenstartzeit CLOCK,
      schaltstatus FIXED,
      statusschutz REF SEMA,
      ausloesung REF SEMA,
      steuerung REF ADAPTSTEUERUNG];
```

Beispiel 23: Typvereinbarung zur Klasse `AUSLOESERBEENDER`

Beispiel 24 zeigt den Rot-Grün-Rot-Schaltzyklus der Klasse `Ausloeserbeender`. Nach dem Einschalten der Steuerung blinken zunächst die beiden Ampeln der Straße eine gewisse Zeit, werden dann auf Gelb und schließlich auf Rot geschaltet. In Beispiel 24 sind die dafür notwendigen Anweisungen durch einen Kommentar ersetzt.

Danach beginnt der eigentliche Steuerungszyklus (Beispiel 24) (Er kann an zwei Stellen durch eine EXIT-Anweisung verlassen werden, wenn das `zyklusbit` auf '0'B gesetzt wird): Mit Aufruf der Methode `abwarten` des Ausloeserbeenders wird auf Auslösung des Grün-Rot-Ampelzyklus gewartet. Danach wird die Fläche exklusiv besetzt bzw. vorher auf deren Freigabe durch die andere Straße gewartet. Dann erfolgt das Grünschalten als Methode des Ausloeserbeenders (Beispiel 25), in der die Startzeit für die Grünphase und der neue Zustand notiert werden. Außerdem wird dem Ausloeserbeender mitgeteilt, durch welche Task seine Grünschalt-Methode aufgerufen worden ist, damit er den Wartezustand dieser Task bei Bedarf verlängern kann.

```
ADAPTSTEUERUNG_rotgruenzyklus:PROC(steuerung INV REF ADAPTSTEUERUNG,
                            zyklusbit INV REF BIT);
   /* Ampel nach Gelbblinken und Gelb auf Rot schalten */
   WHILE CONT zyklusbit REPEAT
      steuerung.ausloeserbeender.abwarten(CONT steuerung.ausloeserbeender);
      IF NOT CONT zyklusbit THEN EXIT; FIN;
      steuerung.strasse.kreuzung.flaeche.besorgen
                (CONT steuerung.strasse.kreuzung.flaeche);
      IF NOT CONT zyklusbit THEN
         steuerung.strasse.kreuzung.flaeche.freigeben
             (CONT steuerung.strasse.kreuzung.flaeche);
         EXIT;
      FIN;
      steuerung.ausloeserbeender.gruenschalten(CONT steuerung.ausloeserbeender);
      SUSPEND;
      steuerung.strasse.rotschalten(CONT steuerung.strasse);
      AFTER steuerung.strasse.rotueberlapp RESUME;
      steuerung.strasse.kreuzung.flaeche.freigeben
               (CONT steuerung.strasse.kreuzung.flaeche);
   END; !  WHILE REPEAT
  /* Ampel nach Gelbphase und Blinken abschalten */
END; ! PROC
```

Beispiel 24: Rot-Grün-Rot-Ampelzyklus

```
AUSLOESERBEENDER_gruenschalten:PROC (ausloeserbeender INV REF AUSLOESERBEENDER);
   SPC gruendauer DUR IDENT(ausloeserbeender.steuerung.strasse.gruendauer);
   REQUEST ausloeserbeender.statusschutz;
   ausloeserbeender.gruenstartzeit := NOW;
   ausloeserbeender.steuerung.strasse.gruenschalten
                        (CONT ausloeserbeender.steuerung.strasse);
   ausloeserbeender.schaltstatus := GRUENVERLAENGERN;
   ausloeserbeender.gruenschalttask := TASK;
   AFTER gruendauer ACTIVATE ausloeserbeender.weitertask;
   RELEASE ausloeserbeender.statusschutz;
END;
```

Beispiel 25: Grünschalt-Methode des Ausloeserbeenders

Nach Ende der Grünphase wird in Beispiel 24 eine kurze Zeitdauer `rotueberlapp` gewartet, in der beide Straßen Rot zeigen; dann wird die Fläche freigegeben, damit sie jetzt eventuell von der anderen Straße belegt werden kann.

Beispiel 26 zeigt die Endlosschleife eines Ausloeserbeenders. In ihr wird zunächst auf ein Ereignis (die Auslösung eines Interrupts) gewartet. Je nachdem, welchen Schaltstatus die Steuerung hat, wird in der `CASE`-Verzweigung entweder die Grünphase ausgelöst oder gar nichts getan oder die Grünphase verlängert.

Weil zwei Tasks, nämlich diejenige des Ausloeserbeenders und diejenige der Steuerung, auf die Variable `ausloeserbeender.schaltstatus` zugegreifen, werden die Zugriffe mit dem Semaphor `statusschutz` koordiniert.

```
AUSLOESERBEENDER_wartezyklus:PROC ( ausloeserbeender INV REF AUSLOESERBEENDER,
                                    eingeschaltet INV REF BIT(1));
   DCL (startzeit,jetzt) CLOCK,
       (gruendauer,restdauer,maxgruendauer)DUR;
   gruendauer := ausloeserbeender.steuerung.strasse.gruendauer;
   maxgruendauer := ausloeserbeender.steuerung.strasse.maxgruendauer;
   REQUEST ausloeserbeender.statusschutz;
   REPEAT
      RELEASE ausloeserbeender.statusschutz;
      ausloeserbeender.ereignis.erwarten(ausloeserbeender.ereignis);
      REQUEST ausloeserbeender.statusschutz;
      IF NOT CONT eingeschaltet THEN EXIT; FIN;
      CASE ausloeserbeender.schaltstatus
        ALT (NORMALROT)
         ausloeserbeender.schaltstatus := GRUENGEFORDERT;
         RELEASE ausloeserbeender.ausloesung;
        ALT (GRUENGEFORDERT);
        ALT (GRUENVERLAENGERN)
         startzeit := ausloeserbeender.gruenstartzeit;
         PREVENT ausloeserbeender.weitertask;
         TERMINATE ausloeserbeender.weitertask;
         jetzt := NOW;
         IF jetzt+gruendauer<startzeit+maxgruendauer
           THEN
            restdauer := gruendauer;
           ELSE
            restdauer := startzeit+maxgruendauer-jetzt;
            ausloeserbeender.schaltstatus := GRUENENDE;
         FIN;
         AFTER restdauer ACTIVATE ausloeserbeender.weitertask;
        ALT (GRUENENDE);
      FIN;
   END;
   RELEASE ausloeserbeender.statusschutz;
END; ! PROC
```

Beispiel 26: Reaktion auf Ereignisse im Ausloeserbeender

Die Verlängerung bzw. Beendigung der Grünphase erfolgt in den Beispielen 25 und 26 durch die Einplanungs-Anweisung

`AFTER restdauer ACTIVATE ausloeserbeender.weitertask;`.

Wenn mehrere Fahrzeuge aufeinander folgen, wird diese Anweisung nacheinander für jedes Fahrzeug ausgeführt. In PEARL90 hat jedoch nur die zeitlich letzte Einplanung für eine Task Geltung.

Deshalb wird durch jede neue Einplanung der Start der `weitertask` immer weiter in die Zukunft verschoben.

Die `weitertask` setzt den Rot-Grün-Zyklus durch Aufruf der Methode aus Beispiel 27 fort.

4.2.2 Probleme mit SUSPEND und CONTINUE

Ein wesentliches Problem bei einem Programm, in dem mehrere Tasks gleichzeitig laufen, stellen zeitliche Zufälligkeiten dar, die sehr selten auftreten und deshalb im Programmtest nicht gefunden werden können. Sie erfordern eine sorgfältige Analyse des Programmes, bei der alle möglichen Zufälle im Zusammenwirken der Tasks berücksichtigt werden müssen. Ein Beispiel dafür bietet die Beendigung der Grünphase.

In Beispiel 25 wartet die `gruenschalttask`, die den Rot-Grün-Zyklus ausführt, mit einem SUSPEND auf das Ende der Grünphase. Eine CONTINUE-Anweisung für diese Task ist nur dann sinnvoll, wenn sie wirklich suspendiert ist. Im verwendeten PEARL-System löst deshalb die Anweisung `CONTINUE diesetask;` ein SIGNAL bzw. eine Fehlermeldung aus, wenn `diesetask` nicht suspendiert ist.

Die Fehlermeldung kann zwar durch eine ON-Klausel unterdrückt werden, wenn aber die Anweisung `AFTER restdauer CONTINUE diesetask;` lautet, kann die Anweisung selbst kein SIGNAL auslösen, weil die eigentliche Ausführung des CONTINUE ja erst später erfolgt. Deshalb ist dann die Unterdrückung der Fehlermeldung nicht möglich.

```
AUSLOESERBEENDER_gruenbeenden:PROC ( ausloeserbeender INV REF AUSLOESERBEENDER);
   REQUEST ausloeserbeender.statusschutz;
   ausloeserbeender.schaltstatus := NORMALROT;
   CONTINUE ausloeserbeender.gruenschalttask;  ! Fortsetzung des Rot-Gruen-Zyklus
   RELEASE ausloeserbeender.statusschutz;
END;
```

Beispiel 27: Zum Beenden der Grünphase ruft eine eingeplante Task diese Prozedur auf

In der ersten Version des Programmes wurde die Fortsetzung der Gruenschalttask durch eine Anweisung `AFTER restdauer CONTINUE ...` direkt vom Ausloeserbeender eingeplant. Sobald die Gruenschalttask hinter dem `SUSPEND;` fortgesetzt wurde, änderte sie sofort den Schaltzustand der Steuerung auf `NORMALROT`. Dadurch sollte der Ausloeserbeender erkennen können, daß die Task nicht mehr suspendiert war.

Wenn nun in der sehr kurzen Zeit zwischen der Fortsetzung der Gruenschalttask und ihrer Änderung des Schaltzustandes ein Fahrzeug einen Interrupt auslösen würde, hätte der Ausloeserbeender noch keine Kenntnis davon, daß die Gruenschalttask in Wirklichkeit nicht mehr auf dem SUSPEND wartet. Der Ausloeserbeender würde dann ihre Fortsetzung noch einmal mit der Anweisung `AFTER restdauer CONTINUE ...` einplanen. Das würde dann zu der eben geschilderten Fehlersituation führen.

Aus diesem Grunde wird das CONTINUE durch eine zusätzliche Task `weitertask` ausgeführt, die außerdem den Schaltzustand ändert. Durch Semaphoranweisungen wird dafür gesorgt, daß der Ausloeserbeender den Schaltzustand während dieser Zeit nicht abfragen kann (Beispiel 27).

4.3 Ereignisse und Interrupts

Ereignisse im technischen Prozeß werden einem PEARL90-Programm durch Interrupts gemeldet. Beim hier vorliegenden System werden die Interrupts durch Induktionsschleifen oder ähnliche Sensoren ausgelöst. Diese Interrupts werden im objektorientierten Programm durch Objekte der Klasse `EREIGNIS` repräsentiert, deren zugehörige Typvereinbarung (Beispiel 28) eine Referenz auf den zugehörigen Interrupt enthält. Die Methoden dieser Klasse sind in den Beispielen 29-31 enthalten; sie sind so geschrieben, daß das Programm auch getestet werden kann, wenn der zum Test verwendete Rechner keine entsprechenden Interrupts besitzt und diese simuliert werden müssen.

```
TYPE EREIGNIS STRUCT[(init, erwarten, simulieren)INV REF PROC(INV REF EREIGNIS),
 wartend BIT(1), (schutz, weiter)INV REF SEMA, irpt REF IRPT];
    TYPE EREIGNIS STRUCT[
              (init,
               erwarten,
               simulieren) INV REF PROC(INV REF EREIGNIS),
               wartend BIT(1),
               task REF TASK,
               irpt REF INTERRUPT];
```

Beispiel 28: Typvereinbarung zur Klasse `EREIGNIS`

```
EREIGNIS_init:PROC (ereignis INV REF EREIGNIS);
   WHILE TRY ereignis.schutz REPEAT END;
   ereignis.wartend := '0'B;
   RELEASE ereignis.schutz;
   WHILE TRY ereignis.weiter REPEAT
   END;
   IF CONT ereignis.irpt IS NIL THEN
      RETURN;
     ELSE
      ENABLE ereignis.irpt;
   FIN;
END;
```

Beispiel 29: Initialisierung eines Objektes der Klasse `EREIGNIS`

Die wichtigste Methode ist `erwarten`. Sie wird von einer Task aufgerufen, die auf den Interrupt warten will. Bei der Simulation ist im deklarierten `EREIGNIS`-Objekt bei der Referenz auf den Interrupt `NIL` eingetragen. Die Task wartet dann nicht auf der Anweisung `WHEN ereignis.irpt RESUME;` auf den Interrupt, sondern auf einer REQUEST-Anweisung.

```
EREIGNIS_erwarten:PROC (ereignis INV REF EREIGNIS);
   IF CONT ereignis.irpt ISNT NIL
     THEN
      WHEN ereignis.irpt RESUME;
     ELSE
      REQUEST ereignis.schutz;
      ereignis.wartend := '1'B;
      RELEASE ereignis.schutz;
      REQUEST ereignis.weiter;
   FIN;
END;
```

Beispiel 30: Warten auf einen Interrupt oder auf seine Simulation

In PEARL90 hat ein Interrupt, auf den mit `WHEN ereignis.irpt RESUME;` gewartet werden soll, keine Wirkung, wenn die Task, die warten soll, nicht auf dieser Anweisung wartet, sondern irgend etwas anderes ausführt. Deshalb muß bei der Simulation festgestellt werden, ob die Task auf der REQUEST-Anweisung wartet; das geschieht durch Setzen und Rücksetzen eines Merkbits. Da diese Zugriffe durch konkurrierende Tasks erfolgen, werden sie durch einen Semaphor koordiniert.

```
EREIGNIS_simulieren:PROC (ereignis INV REF EREIGNIS);
   IF CONT ereignis.irpt ISNT NIL
     THEN
      TRIGGER ereignis.irpt;
     ELSE
      REQUEST ereignis.schutz;
      RELEASE ereignis.weiter;
      ereignis.wartend := '0'B;
      RELEASE ereignis.schutz;
   FIN;
END;
```

Beispiel 31: Simulation eines Interrupts in der Klasse `EREIGNIS`

4.4 Ein- und Ausgabe über Tastatur und Bildschirm

4.4.1 Terminals als Ausführende von Aufträgen

Auf dem Bildschirm wird einerseits oben das Bild der Kreuzung mit den Ampeln und deren Zustand dargestellt; andererseits soll der untere Teil des Bildschirmes und die Tastatur jedoch auch für Bediendialoge verwendet werden. Die betreffenden Programmteile sollen gleichzeitig in nebenläufigen Tasks laufen. Das macht erforderlich, daß der Bildschirm für jeden Dialog und für jede Ausgabe des Kreuzungsbildes exklusiv für die jeweilige Task reserviert wird, weil sonst z. B. das Echo von Eingaben mitten im Ampelbild erscheinen könnte.

Im objektorientierten Ansatz sind Bildschirm und Tastatur Bestandteile eines Objektes der Klasse `TERMINAL`. Andererseits existieren auch Objekte der Klasse `AMPELBILD`, die zur Ausgabe der Ampelbilder auf dem Bildschirm dienen und dabei ein Terminal benutzen müssen. Das Verhältnis zwischen Ampelbild-Objekten und einem Terminal ist daher demjenigen sehr ähnlich, das zwischen einem Blinker-Objekt und einem Zyklus-Objekt besteht: letzteres führt einen Auftrag für den jeweiligen Blinker aus. Ebenso könnte ein Terminal einen Ausgabeauftrag für ein Ampelbild ausführen.

Dabei gibt es jedoch einen Unterschied: Jeder Blinker besitzt seinen eigenen Zyklus, während das Terminal von allen Ampelbildern gemeinsam benutzt wird. Wie oben bereits angedeutet, müssen deshalb die Aufträge mit einem Semaphor koordiniert werden, damit sie nicht gleichzeitig, sondern nacheinander ausgeführt werden.

Beispiel 32 zeigt die Typvereinbarung der Klasse `TERMINAL`, Beispiel 33 diejenige Methode, die einen Auftrag ausführen soll. Ihr werden wieder Auftrag und Auftraggeber als Argumente bzw. Parameter übergeben. Ein Terminal enthält außer Referenzen auf die Datenstationen `input` und `output` und die Methoden `init`, `close` und `ausfuehren` noch weitere Komponenten, die teilweise erst in späteren Abschnitten beschrieben werden.

```
TYPE TERMINAL STRUCT[
     fehler FIXED,
     input REF DATION IN ALPHIC,
     output REF DATION OUT ALPHIC,
     einzeln REF SEMA,
     open INV REF PROC(REF TERMINAL),
     close INV REF PROC(REF TERMINAL),
     ausfuehren INV REF PROC(REF INV TERMINAL,
                             auftraggeber INV REF STRUCT[],
                             auftrag INV REF PROC
                                        (auftraggeber INV REF STRUCT[]))];
                             zeitlimit INV DUR),
     auftrag REF PROC(auftraggeber INV REF STRUCT[]),
     auftraggeber REF STRUCT[],
     zeitlimit DUR,
     ueberwacher INV REF UEBERWACHER];
```

Beispiel 32: Typvereinbarung zur Klasse `TERMINAL`

```
TERMINAL_ausfuehren:PROC(terminal INV REF TERMINAL,
                         auftraggeber INV REF STRUCT[],
                         auftrag REF PROC(auftraggeber INV REF STRUCT[]));
  REQUEST terminal.einzeln;
    auftrag(CONT auftraggeber);
  RELEASE terminal.einzeln;
END;
```

Beispiel 33: Methode ausfuehren zur Klasse `TERMINAL` (erster Entwurf)

4.4.2 Behandlung von E/A-Exceptions

Die Methode aus Beispiel 33 hat jedoch einen Haken: Bei Ausführung eines E/A-Auftrages können bei der Programmausführung Fehlersituationen auftreten, die ein SIGNAL auslösen. Die Standard-Reaktion des PEARL-Systems auf SIGNALs ist eine Fehlermeldung; bei gravierenden Fehlern erfolgt außerdem ein Abbruch der Task, in der der Fehler aufgetreten ist. Im allgemeinen führt das zum Zusammenbruch des gesamten Programmes und macht einen Neustart erforderlich.

Da das in den meisten Prozeßlenkungsprogrammen nicht hinzunehmen ist, kann der Anwender auch seine eigene SIGNAL-Reaktionen in ON-Klauseln formulieren. Eine weitere Möglichkeit besteht in der Verwendung des RST-Formates in den E/A-Anweisungen. Beide Möglichkeiten erfordern aber, daß der Programmierer die richtigen Reaktionen auf alle denkbaren Fehler vorsieht, um den Abbruch von Tasks zu vermeiden.

```
TERMINAL_ausfuehren:PROC(terminal INV REF TERMINAL,
                         auftraggeber INV REF STRUCT[],
                         auftrag REF PROC(auftraggeber INV REF STRUCT[]),
                         zeitlimit INV DUR);
 ON zeitueberschreitung:BEGIN
       RELEASE terminal.einzeln;
       INDUCE zeitueberschreitung;
   END;
  REQUEST terminal.einzeln;
   terminal.auftrag:=auftrag;
   terminal.auftraggeber:=CONT auftraggeber;
   terminal.zeitlimit:=zeitlimit;
   terminal.weiter:='1'B;
   TO 5 WHILE terminal.weiter REPEAT
    terminal.weiter:='0'B;
    TERMINAL_hilfe(terminal);
   END;
  RELEASE terminal.einzeln;
END;
TERMINAL_hilfe:PROC(terminal INV REF TERMINAL);
 ON eafehler RST(terminal.fehler):TERMINAL_exception(terminal);
 terminal.ueberwacher.beauftragen(terminal.ueberwacher,
                                  terminal.auftrag,
                                  terminal.auftraggeber,
                                  terminal.zeitlimit);
END;
```

Beispiel 34: Methode ausfuehren zur Klasse `TERMINAL`

Offensichtlich wäre es von großem Vorteil, wenn die Behandlung von derartigen Fehlern zentral im Terminal-Objekt erfolgen könnte, statt dezentral innerhalb eines jeden Auftrages. Die Prozedur `TERMINAL_ausfuehren` darf jedoch keine ON-Klausel enthalten; das würde nämlich dazu führen, daß sie vor Ausführung von `RELEASE terminal.einzeln;` abgebrochen würde, wenn das betreffende Signal bei der Ausführung des Auftrages ausgelöst worden wäre. Falls eine E/A-Anweisung nämlich

zur Ausführung einer ON-Klausel führt, wird die Ausführung der betreffenden E/A-Anweisung abgebrochen, die in der ON-Klausel stehende Anweisung ausgeführt und danach die Prozedur oder Task abgebrochen, die das Signal ausgelöst hatte.

Deshalb wurde die in Beispiel 33 dargestellte Methode TERMINAL_ausfuehren so ergänzt, wie es Beispiel 34 zeigt. (Das Beispiel enthält noch eine weitere Ergänzung, die in Kapitel 4.6 besprochen wird.)

Im Beispiel 34 wird eine Hilfsprozedur `TERMINAL_hilfe` aufgerufen, die die ON-Klausel enthält. Deren Aufruf befindet sich in einem REPEAT-Block; dadurch können Aufträge, bei deren Ausführung Fehler aufgetreten sind, bis zu fünfmal wiederholt werden. `TERMINAL_hilfe` reicht den Auftrag an einen Überwacher weiter, der ihn ausführt.

Falls jetzt innerhalb des Auftrages ein Signal `eafehler` ausgelöst wird, wird die Art des Fehlers in `terminal.fehler` notiert. Dann wird die Prozedur `TERMINAL_exception` (Beispiel 35) zur Behandlung des Fehlers aufgerufen und die Prozedur `TERMINAL_hilfe` beendet.

```
TERMINAL_exception:PROC(terminal INV REF TERMINAL);
  CASE  terminal.fehler
    ALT (C_ERR_ILLEGAL_NUMBER)
       GET FROM terminal.input BY SKIP;
       PUT 'Falsche Zahleneingabe' TO terminal.output BY A,SKIP;
       terminal.weiter:='1'B;
    ALT (C_ERR_DATION_NOT_OPENED)
       OPEN terminal.input;
       OPEN terminal.output;
       PUT 'exception ist ausgelöst' TO terminal.output BY A,SKIP;
       PUT 'Fehler: ',terminal.fehler TO display BY A,F(6),SKIP;
       terminal.weiter:='1'B;
      OUT PUT 'normaler E/A-Fehler ',terminal.fehler TO terminal.output
            BY SKIP,A,F(6),SKIP;
          terminal.weiter:='0'B;
  FIN;
END;
```

Beispiel 35: Beispiele für die Behandlung von Exceptions

Beispiel 35 zeigt die Fehlerbehandlung durch `TERMINAL_exception` in zwei Sonderfällen: wenn der Fehler im Auftrag dadurch verursacht worden war, daß die Datenstationen `terminal.input` und `terminal.output` nicht eröffnet worden waren, werden sie eröffnet und es wird versucht, den Auftrag durch nochmaligen Aufruf von `TERMINAL_hilfe` doch auszuführen. Auch bei falscher Eingabe einer Zahl, z. B. durch Eingabe eines Buchstabens statt einer Ziffer, wird der Fehler gemeldet und dafür gesorgt, daß der Auftrag wiederholt wird.

4.5 Bediendialoge

Ein Bediendialog gibt die Zeilen eines Menüs numeriert aus und fordert dann dazu auf, eine der Nummern einzugeben. Üblicherweise wird mit einer CASE-Anweisung in den gewünschten Programmzweig verzweigt.

Dieses Verfahren ist etwas fehleranfällig, weil die Menütexte (als CHAR()-Array) und die CASE-Anweisungen zwar logisch zusammengehören und genau aufeinander abgestimmt sein müssen, aber im Programm im Deklarationsteil bzw. Anweisungsteil und damit an verschiedenen Stellen stehen.

```
TYPE DIALOG STRUCT[
      init INV REF PROC(INV REF DIALOG,INV REF () ZWEIG),
      start INV REF PROC(INV REF DIALOG),
      aendern INV REF PROC(INV REF DIALOG,
                           zweige INV REF () ZWEIG),
      terminal INV REF TERMINAL,
      ueberschrift CHAR(30),
      voreinstellung FIXED(31),
      zweige REF () ZWEIG];
```

Beispiel 36: Typvereinbarung für die Klasse `DIALOG`

Im Beispielprogramm werden Bediendialoge durch ein Objekt der Klasse `DIALOG` (Beispiel 36) vermittelt. Wichtigste Komponente eines derartigen Objektes ist die letzte Komponente, welche eine Referenz auf ein Feld von Zweigen enthält. Dieses bildet die Schnittstelle zu den Programmteilen, die durch den Dialog ausgewählt und angestoßen werden können.

```
TYPE ZWEIG STRUCT[
        text CHAR(50),
        auftrag INV REF PROC];

 DCL dialog_zweige_start (3) ZWEIG INIT(
     'Blinker einschalten                              ',
     kreuzung_blinker_ein,
     'Steuerung einschalten                            ',
     kreuzung_steuerung_ein,
     'Programm beenden                                 ',
     programmende);
```

Beispiel 37: Deklaration einer Schnittstelle zwischen Bediendialog und Programm

Beispiel 37 zeigt die Vereinbarung eines derartigen Feldes für den Bediendialog auf oberster Ebene. Es enthält paarweise die Texte der Menüzeilen und Referenzen auf parameterlose Prozeduren, die den einzelnen Programmzweigen entsprechen. Beispiel 38 zeigt eine solche Prozedur; in ihr wird die Steuerung der Kreuzung gestartet und dann in der Methode `aendern` der Klasse `DIALOG` (Beispiel 40) die Komponente `zweige` aus Beispiel 36 überschrieben, um auf den Unterdialog umzuschalten, der zur Kreuzungssteuerung gehört.

```
kreuzung_steuerung_ein:PROC;
   kreuzung.steuerung.start(kreuzung.steuerung);
   dialog.aendern(dialog,dialog_steuerung_bedienen);
END;
```

Beispiel 38: Durch Bediendialog anwählbare Prozedur zum Start der Steuerung

```
DIALOG_aendern:PROC(dialog INV REF DIALOG,
                    zweige INV REF () ZWEIG);
    dialog.zweige:= zweige;
    dialog.voreinstellung:=1;
END;
```

Beispiel 39: Laden eines neuen Unterdialoges

Zur Ausführung eines Dialoges wird ein Objekt der Klasse `TERMINAL` verwendet. Deshalb enthält die Typvereinbarung in Beispiel 36 eine Komponente für die Referenz auf ein derartiges Objekt. Die Ausführung eines Dialoges und die Verzweigung in den gewünschten Teil des Programmes erfolgt folgendermaßen:

Der Dialog wird durch die Methode `start` der Klasse `DIALOG` (Beispiel 40) gestartet. In ihr wird dem Terminal durch Aufruf der Terminal-Methode `ausfuehren` der Auftrag für den eigentlichen Bediendialog `DIALOG_auftrag` (Beispiel 41) erteilt: Das Menü wird samt Überschrift auf dem Display `terminal.output` ausgegeben und dann in der Prozedur `FIXEDEINGABE` die Komponente `voreinstellung` geändert. Schließlich wird die entsprechende Prozedur über die Referenz `auftrag` im Feld `dialog`.zweige aufgerufen, dessen Komponente über den Index `voreinstellung` ausgewählt ist.

```
DIALOG_start:PROC(dialog INV REF DIALOG);
    ON zeitueberschreitung:BEGIN ! falls Dialog zu lange dauert
     PUT 'Höchstzeit überschritten: RETURN eingeben!'
                      TO  dialog.terminal.output BY A,SKIP;
    END;
   dialog.terminal.ausfuehren(CONT dialog.terminal,
                              dialog,
                              DIALOG_auftrag); ! Eingabe von voreinstellung
   IF dialog.voreinstellung <= UPB dialog.zweige
       AND dialog.voreinstellung >= 1 THEN
     dialog.zweige(dialog.voreinstellung).auftrag;
   FIN;
END;
```

Beispiel 40: Methode zum Starten eines Dialoges

```
DIALOG_auftrag:PROC(dialog INV REF DIALOG);
    PUT GANZUNTEN TO dialog.terminal.output BY A,SKIP;
    PUT LOESCHEREST TO dialog.terminal.output BY A,SKIP;
    PUT dialog.ueberschrift TO dialog.terminal.output BY A,SKIP;
    FOR i TO UPB dialog.zweige REPEAT
      PUT i, dialog.zweige(i).text TO dialog.terminal.output
                    BY F(3),X(2),A,SKIP;
    END;
    PUT UPB dialog.zweige+1,'EXIT' TO dialog.terminal.output
                BY F(3),X(2),A,SKIP;
   CALL FIXEDEINGABE('Gib Nr. des ausgewählten Menuepunktes ein: ',
                    1 LWB dialog.zweige,1 UPB dialog.zweige +1,
                    dialog.voreinstellung,
                    dialog.terminal.input,
                    dialog.terminal.output);
      PUT  GANZUNTEN TO dialog.terminal.output BY A;
 END;
```

Beispiel 41: Durchführung einer Dialogeingabe

4.6 Behandlung von Zeitüberschreitungen

4.6.1 Überwachung von Ausführungszeiten

Echtzeitprogramme unterscheiden sich bekanntlich von "normalen" Programmen dadurch, daß es in ihnen Obergrenzen für Rechenzeiten gibt; beispielsweise müssen häufig Stellwerte, die an einen technischen Prozeß ausgegeben werden sollen, innerhalb sehr kurzer Zeiten aus eingelesenen Prozeßdaten berechnet werden. Deshalb müssen nicht selten Rechenzeiten überwacht und in Notsituationen Ersatzwerte bestimmt oder andere Abhilfen geschaffen werden.

In dem hier beschriebenen Beispiel tritt ein derartiges Problem dadurch auf, daß das Terminal sowohl für die Darstellung der Ampelbilder als auch für den Bediendialog verwendet wird. Während des Bediendialoges ist das Terminal mit Semaphor-Anweisungen exklusiv reserviert, so daß während dieser Zeit kein neues Ampelbild ausgegeben werden kann.

Aus diesem Grunde unterliegen alle Aufträge, die dem Terminal gegeben werden, einer Zeitüberwachung. Falls das im Programm gegebene Zeitlimit überschritten wird, wird ein SIGNAL ausgelöst.

Im Beispielprogramm gibt es dafür eine Klasse `UEBERWACHER`. Beispiel 42 zeigt die zugehörige Typvereinbarung. Ein Objekt, das eine Methode enthält, deren Ausführungzeit von einem Überwacher überwacht werden soll, muß dem Überwacher beim Aufruf von `beauftragen` diese Methode als Auftrag übergeben; außerdem werden dem Überwacher der Auftraggeber und die Höchstzeit für die Ausführung des Auftrages mitgeteilt.

```
    TYPE UEBERWACHER STRUCT[
         rechtzeitig BIT(1),
         auftrag REF PROC(auftraggeber INV REF STRUCT[]),
         auftraggeber REF STRUCT[],
         beauftragen INV REF PROC(
                  INV REF UEBERWACHER,
                  auftrag INV REF PROC(auftraggeber INV REF STRUCT[]),
                  auftraggeber INV REF STRUCT[],
                  zeitlimit DUR),
         (normalarbeit,
         alarmarbeit) INV REF PROC(INV REF UEBERWACHER),
         zeitlimit DUR,
         normaltask INV REF TASK,
         wachetask INV REF TASK,
         drittetask INV REF TASK,
         normalbit BIT(1),
         (endesema,
         schutz) INV REF SEMA,
         eafehlernr FIXED];
```

Beispiel 42: Typvereinbarung für die Klasse `UEBERWACHER`

```
UEBERWACHER_beauftragen:PROC(ueberwacher INV REF UEBERWACHER,
              auftrag INV REF PROC(auftraggeber INV REF STRUCT[]),
              auftraggeber INV REF STRUCT[],
              ersatzauftrag INV REF PROC(ersatzauftraggeber INV REF STRUCT[]),
              ersatzauftraggeber INV REF STRUCT[],
              zeitlimit DUR);
   ueberwacher.auftrag := auftrag;
   ueberwacher.auftraggeber := CONT auftraggeber;
   ueberwacher.zeitlimit := zeitlimit;
 REQUEST ueberwacher.schutz;
  IF ueberwacher.normalbit
     THEN ACTIVATE ueberwacher.normaltask PRIO PRIO;
     ELSE ACTIVATE ueberwacher.drittetask PRIO PRIO;
   FIN;
   AFTER ueberwacher.zeitlimit ACTIVATE ueberwacher.wachetask PRIO PRIO-1;
  RELEASE ueberwacher.schutz;
   REQUEST ueberwacher.endesema;
   IF NOT ueberwacher.rechtzeitig THEN INDUCE zeitueberschreitung; FIN;
   IF ueberwacher.eafehlernr/=0 THEN
     INDUCE eafehler RST(ueberwacher.eafehlernr);
   FIN;
END;
```

Beispiel 43: Methode zur Beauftragung eines Ueberwachers

Der Überwacher startet zwei Tasks, die erste sofort, die zweite über eine Einplanung nur dann, wenn das Ende des Zeitlimits erreicht ist. Die erste Task versucht den Auftrag auszuführen; falls

ihr das nicht rechtzeitig gelingt, wird sie von der zweiten Task terminiert; letztere bewirkt dann die Auslösung eines SIGNALs.

Beispiel 43 zeigt die Methode `beauftragen` des Überwachers. Normalerweise ist das Bit `normalbit` gesetzt. Der Überwacher startet daher eine Task `normaltask`. Außerdem wird eine Task `wachetask` so eingeplant, daß sie nach Überschreitung des Zeitlimits aktiviert wird. Dann wird auf einer REQUEST-Anweisung gewartet.

Die `normaltask` erhält beim Start durch die Floskel `PRIO PRIO` genau die Priorität, die durch die Einbaufunktion `PRIO` (ihr Aufruf ist das zweite `PRIO`) ermittelt wird. Die Wachetask bekommt mit `PRIO-1` höhere Priorität.

Die Task `ueberwacher.normaltask` führt die die Methode `normalarbeit` aus (Beispiel 44). In ihr wird zunächst mit der Ausführung des Auftrag des Auftraggebers begonnen. Wenn der Auftrag rechtzeitig innerhalb des Zeitlimits beendet wird, wird die Wachetask ausgeplant, so daß sie nicht aktiviert wird.

```
    UEBERWACHER_normalarbeit:PROC(ueberwacher INV REF UEBERWACHER);
      ON eafehler RST(ueberwacher.eafehlernr):GOTO ende;
       ueberwacher.auftrag(ueberwacher.auftraggeber);
ende:REQUEST ueberwacher.schutz;
       ueberwacher.rechtzeitig:='1'B;
       PREVENT ueberwacher.wachetask;
       TERMINATE ueberwacher.wachetask;
       RELEASE ueberwacher.endesema;
      RELEASE ueberwacher.schutz;
    END;
```

Beispiel 44: Methode `UEBERWACHER_normalarbeit`; Ausführung durch ueberwacher.normaltask

```
    UEBERWACHER_alarmarbeit:PROC(ueberwacher INV REF UEBERWACHER);
      REQUEST ueberwacher.schutz;
       IF ueberwacher.normalbit THEN
          ueberwacher.normalbit:='0'B;
          TERMINATE ueberwacher.normaltask;
         ELSE
          ueberwacher.normalbit:='1'B;
          TERMINATE ueberwacher.drittetask;
        FIN;
       ueberwacher.rechtzeitig:='0'B;
       RELEASE ueberwacher.endesema;
      RELEASE ueberwacher.schutz;
    END;
```

Beispiel 45: Methode `UEBERWACHER_alarmarbeit`

Die Wachetask führt die Methode `ueberwacher.alarmarbeit` (Beispiel 45) aus, wenn das Zeitlimit überschritten worden ist. Sie terminiert die Task `ueberwacher.normaltask` und notiert, daß

die `normalarbeit` nicht rechtzeitig beendet wurde. Dann synchronisiert sie sich mit der Methode `beauftragen` und setzt dadurch den Auftraggeber fort.

Da mehrere Tasks auf Merkbits zugreifen bzw. sich gegenseitig beeinflussen, werden sie mit dem Semaphor `schutz` koordiniert.

4.6.2 Signalisierung von Zeitüberschreitungen

Beide Tasks, Normaltask und Wachetask, synchronisieren sich durch eine RELEASE-Anweisung mit derjenigen Task, die den Überwacher beauftragt hat und in der Methode `beauftragen` darauf wartet, daß eine der beiden Tasks zum Ende kommt. Wenn das der Fall ist, wird am Inhalt des Bits `rechtzeitig` festgestellt, ob der Auftrag rechtzeitig beendet worden ist. Falls das nicht der Fall ist, wird in Beispiel 45 das SIGNAL `zeitueberschreitung` durch eine INDUCE-Anweisung ausgelöst.

Die INDUCE-Anweisung bewirkt, daß in Beispiel 45 die Methode `beauftragen` des Überwachers abgebrochen wird. Desgleichen würden alle Methoden (Prozeduren) abgebrochen, die `beauftragen` direkt oder indirekt aufrufen, und letztendlich auch die Task, die den Aufruf von `beauftragen` enthalten hat. Um das zu vermeiden, muß der Auftraggeber der Überwachung mit einer ON-Klausel auf das SIGNAL reagieren.

Im hier betrachteten Fall ist das Terminal der direkte Auftraggeber (Beispiel 34). Bei einer Zeitüberschreitung werden dessen Methoden `hilfe` und `ausfuehren` abgebrochen. Dadurch kann die RELEASE-Anweisung am Ende von `ausfuehren` nicht erfolgen. Deshalb steht ein RELEASE in der ON-Klausel am Anfang von `ausfuehren`, deren BEGIN-END-Block bei Zeitüberschreitungen ausgeführt wird. Danach wird das SIGNAL `zeitüberschreitung` durch INDUCE an den Auftraggeber für den Terminal-Auftrag weitergereicht. Im Falle eines Dialoges enthält deshalb dessen Methode `start` (Beispiel 40) eine ON-Klausel, die zur Ausgabe einer Fehlermeldung führt.

4.6.3 Komplikationen bei Einlese-Aufträgen

Ein Überwacher enthält außer den bisher erwähnten Tasks `normaltask` und `wachetask` noch eine Task `drittetask`. Sie führt die `normalarbeit` alternativ aus. Der Grund dafür ist folgender: Einlese-Operationen werden bei der benutzten Implementation von PEARL90 durch Tochter-Tasks ausgeführt, von denen der Benutzer nichts weiß. Die Mutter-Task, die eine derartige Tochter-Task startet, wird zwar aus der Liste der aktiven Tasks gestrichen, wenn sie terminiert wird, sie kann aber erst wieder neu zum Ablauf kommen, wenn die Tochter-Task nach Eingabe von Return beendet ist. Deshalb kann die `normaltask` des Überwachers erst wieder ablaufen, wenn bei Zeitüberschreitung eines Dialog-Auftrages Return eingegeben worden ist. Um trotzdem ein Ampelbild als Auftrag auf den Bildschirm zu bringen, wird deshalb in Beispiel 45 alternativ die `normaltask` oder die `drittetask` aktiviert.

4.6.4 Weitergabe von Ausnahme-Signalen

Wie schon erwähnt, lösen Ausnahmen (Exceptions) in PEARL SIGNALs aus, auf die man mit ON-Klauseln reagieren kann. Die Auslösung eines SIGNALs in einer Task oder Prozedur, bewirkt deren sofortigen Abbruch. Wenn die Task bzw. Prozedur eine ON-Klausel enthält, wird vor dem Abbruch die Anweisung in der ON-Klausel - eventuell ein BEGIN-END-Block - ausgeführt. Die Auslösung eines E/A-Fehlersignals während der Bearbeitung eines E/A-Auftrages durch einen Überwacher würde also den Abbruch der `normaltask` oder der `drittetask` bewirken, durch die der Überwacher den Auftrag ausführen läßt. Um das zu vermeiden, muß entweder der E/A-Auftrag oder die Methode `normalarbeit` des Überwachers eine ON-Klausel enthalten. Vorsichtshalber enthält daher die `normalarbeit` (Beispiel 45) eine ON-Klausel für E/A-Ausnahmen.(Falls weitere Ausnahmen, z. B. Tasking-Fehler, in einem Überwacher-Auftrag auftreten könnten, müßten auch die in `normalarbeit` abgefangen werden; andernfalls würden sie im Beispielprogramm zur Auslösung des Signals führen, daß das Zeitlimit für den Auftrag überschritten worden ist.)

In dieser ON-Klausel wird die genaue Ursache des E/A-Fehlers durch die RST-Klausel in der Komponente `eafehlernr` notiert. Danach wird durch eine GOTO-Anweisung dafür gesorgt, daß die `normalarbeit` nicht abgebrochen, sondern ordnungsgemäß beendet wird.

Die Arbeit des Überwachers ist durch die Methode `beauftragen` (Beispiel 45) angestoßen worden, die in irgendeiner anderen Task indirekt in der Methode `ausfuehren` des Terminals aufgerufen worden ist. Dort werden alle E/A-Ausnahmen zentral abgefangen, wie in Kapitel 4.4.2 dargelegt worden ist. Deshalb wird in `beauftragen` die Ausnahme, die im E/A-Auftrag ausgelöst und in `normalarbeit` vorläufig abgefangen wurde, erneut mit einer INDUCE-Anweisung ausgelöst.

4.7 Ampelbildausgaben als nebenläufige Tasks

4.7.1 Start und Ausführung der Nebenläufigkeit

Nach jeder Änderung des Schaltzustandes einer Ampel sollte ein neues Ampelbild auf dem Bildschirm ausgegeben werden. Für diese Ausgaben und für die Bediendialoge wird der selbe Display benutzt. Deshalb können während eines Dialoges keine Ampelbilder dargestellt werden. Die Tasks für die Steuerung der Ampelphasen müssen aber trotzdem weiterlaufen und dürfen nicht auf die Erledigung der Bildausgaben warten, weil ja z. B. beim Blinken jede Sekunde geschaltet werden muß. Deshalb werden die Bildausgaben durch zusätzliche Tasks ausgeführt, die von den Steuerungstasks aktiviert werden.

Beispiel 46 zeigt die Typvereinbarung für die Klasse `AMPELBILD`. Sie enthält eine Referenz auf diese zusätzliche `maltask`, die bei der Deklaration eines Objektes vom Typ `AMPELBILD` vereinbart werden muß. Sie wird in der Methode `male` aktiviert und führt die Methode `malprozedur` aus. Diese wiederum übergibt dem Terminal den Malauftrag, der die PUT-Anweisungen `PUT ... TO ampelbild.terminal.output BY ...;` enthält.

```
TYPE AMPELBILD STRUCT[geometrie GEOMETRIE,
                      status BIT(3),
                      tERminal REF TERMINAL,
                      init INV REF PROC(INV  REF INV AMPELBILD),
                      male INV REF PROC(INV REF AMPELBILD,INV REF INV AMPELSTATUS),
                      maltask INV REF TASK,
                      taskschutz INV REF SEMA,
                      malprozedur INV REF PROC(INV REF AMPELBILD)];
```

Beispiel 46: Typvereinbarung für die Klasse `AMPELBILD`

4.7.2 Startversuch einer noch aktiven Task als Komplikation

Dabei ergibt sich wieder eine kleine Komplikation: ein Bediendialog kann bis zu fünf Sekunden dauern (danach bricht ihn der Überwacher ab). Während dieser Zeit wird beim Blinken mit Sicherheit die `malttask` für die Bildausgabe aktiviert, die dann darauf warten muß, daß das Terminal den Ausgabeauftrag ausführt. Die Blinksteuerung möchte aber diese `malttask` schon nach einer Sekunde erneut aktivieren, während letztere noch aktiv ist; das würde beim verwendeten PEARL-System zu einer Warnung führen. Deshalb muß vor einer erneuten Aktivierung festgestellt werden, ob die `malttask` beendet ist und wieder aktiviert werden darf.

Man könnte das scheinbar am einfachsten erreichen, wenn man am Ende der `malprozedur` ein RELEASE für den Semaphor `taskschutz` machen würde; vor Aktivierung der `malttask` würde man dann zunächst durch eine TRY-Klausel versuchen, ein REQUEST auf den Semaphor `taskschutz` zu machen; wenn diese TRY-Klausel den Wert '1'B zurückgibt, ist die `malprozedur` und damit auch die `maltask` beendet und könnte wieder aktiviert werden.

Falls die TRY-Klausel den Wert '0'B zurückgibt, ist die RELEASE-Anweisung am Ende der `malprozedur` noch nicht ausgeführt, die `maltask` läuft noch und darf nicht aktiviert werden. Dann wird aber auch das Bild des allerletzten Ampelzustandes nicht gemalt. Dieser letzte Ampelzustand kann aber stundenlang bestehen bleiben, wenn nämlich die Ampel auf Rot geschaltet ist und keine weiteren Fahrzeuge eintreffen.

Um diese Fehlfunktion zu beseitigen, muß man dafür sorgen, daß auf jeden Fall das Ampelbild von der `maltask` später noch einmal dargestellt wird. Deshalb enthält die Methode `male` (Beispiel 47) eine CASE-Abfrage, in welchem Zustand sich die `maltask` befindet; nur wenn sie nicht aktiv ist, wird sie aktiviert; andernfalls wird nur die Zustandsvariable `taskzustand` auf `NOCHMAL` gesetzt.

Am Ende der `malprozedur` (Beispiel 48) wird abgefragt, welchen Wert die Taskzustands-Variable hat; wenn sie auf `AMPELBILD_NOCHMAL` gesetzt worden ist, wird die `maltask` nach einer Sekunde noch einmal gestartet.

Der Taskzustand `AMPELBILD_NICHTAKTIV` wird von der `malprozedur` gesetzt, um der Methode `male` zu signalisieren, daß die `malprozedur` und damit die `maltask` beendet sind und daß letztere wieder aktiviert werden darf. Auch dabei kann es zu einer Komplikation kommen: wenn `male`

und `maltask` gleichzeitig ausgeführt werden, kann `male` den Taskzustand möglicherweise abfragen, während die `malprozedur` gerade beendet wird und die `maltask` in Wirklichkeit noch aktiv ist.

```
AMPELBILD_male:PROC(aMPelbild INV REF AMPELBILD,
                    status INV REF INV BIT(3));
   ON taskfehler: BEGIN
         AFTER 1 SEC ACTIVATE aMPelbild.maltask;
         RELEASE aMPelbild.taskschutz;
   END;
   aMPelbild.status:=CONT status;
   REQUEST  aMPelbild.taskschutz;
     CASE aMPelbild.taskzustand
       ALT (AMPELBILD_NICHTAKTIV)
           aMPelbild.taskzustand:=AMPELBILD_AKTIV;
           ACTIVATE aMPelbild.maltask;
       ALT (AMPELBILD_AKTIV)
           aMPelbild.taskzustand:=AMPELBILD_NOCHMAL;
       ALT (AMPELBILD_NOCHMAL)
     FIN;
   RELEASE aMPelbild.taskschutz;
END;
```

Beispiel 47: Methode `male` für den Start der Ampelbildausgabe

```
AMPELBILD_malprozedur:PROC(aMPelbild INV REF AMPELBILD);
   ON zeitueberschreitung:BEGIN
           AFTER 1 SEC ACTIVATE aMPelbild.maltask;
   END;
   aMPelbild.tERminal.ausfuehren(CONT aMPelbild.tERminal,
                                 CONT aMPelbild,
                                 AMPELBILD_malauftrag,
                                 1 SEC);
   REQUEST  aMPelbild.taskschutz;
     CASE aMPelbild.taskzustand
       ALT (AMPELBILD_NICHTAKTIV)
       ALT (AMPELBILD_AKTIV)
           aMPelbild.taskzustand:=AMPELBILD_NICHTAKTIV;
       ALT (AMPELBILD_NOCHMAL)
           aMPelbild.taskzustand:=AMPELBILD_AKTIV;
           AFTER 1 SEC ACTIVATE aMPelbild.maltask;
     FIN;
   RELEASE aMPelbild.taskschutz;
END;
```

Beispiel 48: Die `malprozedur`, die von der `maltask` aufgerufen wird

Die Wahrscheinlichkeit, daß dieser Fehler bei der Ausführung des Programmes auftritt, ist sehr gering; trotzdem muß entsprechende Vorsorge getroffen werden. Deshalb enthält `male` (Beispiel 47)

eine ON-Klausel, die dafür sorgt, daß die Aktivierung der aktiven Task nicht zu einer Meldung führt, sondern zur Aktivierung nach einer Sekunde.

Wenn nur die ON-Klausel programmiert worden wäre, ohne die Abfragen des Taskzustandes und der entsprechenden Reaktionen, wäre man nicht sicher gewesen, daß sich die `maltask` beim Versuch der erneuten Aktivierung bei den letzten Aufräumarbeiten befindet; dann könnte der Startversuch nach einer Sekunde zur erneuten Auslösung des Signales `taskfehler` führen, das in diesem Falle nicht abgefangen werden könnte.

Die Methode `malprozedur` enthält eine On-Klausel für das Signal `zeitueberschreitung`. Das erscheint auf den ersten Blick etwas unsinnig zu sein, denn beim Erteilen des Auftrages in der folgenden Zeile wird als Zeitlimit 1 Sekunde genannt; das Malen eines Ampelbildes dauert aber sicher nur Bruchteile einer Sekunde. Durch Fehlbedienungen kann es jedoch passieren, daß beide Tasks `normaltask` und `drittetask` des Überwachers (Beispiel 43) durch je einen Dialog-Versuch blockiert sind, weil nach Zeitüberschreitungen für Dialogeingaben jeweils auf Eingabe von Return gewartet wird. Deshalb kann dann der Überwacher den `malauftrag` nicht ausführen, und die `wachetask` stellt dann eine Zeitüberschreitung fest.

4.8 Reaktion auf Tastatur-Interrupts

```
TYPE AUSLOESER STRUCT[
     steuerbit BIT(1),
     task INV REF TASK,
     endesema REF SEMA,
     ereignis INV REF EREIGNIS,
     init INV REF PROC(ausloeser REF AUSLOESER,
                  auftraggeber REF STRUCT[],
                  auftrag REF PROC(auftraggeber REF STRUCT[])),
     start INV REF PROC(INV REF AUSLOESER),
     stop INV REF PROC(INV REF AUSLOESER),
     beendet INV REF PROC(INV REF AUSLOESER),
     ausfuehrung INV REF PROC(INV REF AUSLOESER),
     auftraggeber REF STRUCT[],
     auftrag REF PROC(auftraggeber REF STRUCT[])];
```

Beispiel 49: Typvereinbarung für die Klasse `AUSLOESER`

Jeder Bediendialog wird durch einen Tastatur-Interrupt Control-C ausgelöst. Zu diesem Zweck enthält das Programm ein Objekt der Klasse `AUSLOESER` (Beispiel 49). Diese Klasse ähnelt sehr der Klasse `ZYKLUS`, enthält aber zusätzlich zu deren Komponenten noch eine Referenz auf einen Interrupt und eine Methode `ausfuehrung` (Beispiel 50). In dieser Methode wird in einer Schleife auf den Interrupt gewartet und dann eine Methode `auftrag` eines Objektes `auftraggeber` ausgeführt. Referenzen auf `auftrag` und `auftraggeber` werden dem Ausloeser bei seiner Initialisierung übergeben.

```
   AUSLOESER_ausfuehrung:PROC(ausloeser INV REF AUSLOESER);
      WHILE ausloeser.steuerbit REPEAT
        ausloeser.ereignis.erwarten(CONT ausloeser.ereignis);
        ausloeser.auftrag(CONT ausloeser.auftraggeber);
      END;
      RELEASE ausloeser.endesema;
   END;
```

Beispiel 50: Ausführen eines Programmteiles nach Auslösung eines Ereignisses

4.9 Programmstart und -ende

Das Programm enthält zum Start durch das Betriebssystem eine Starttask, die in Beispiel 51 gezeigt wird. In ihr werden die im Programm enthaltenen Objekte durch Auruf ihrer `init`-Methoden in den Anfangszustand versetzt. Dabei wird dem Objekt `ausloeser`, das zur eben beschriebenen Klasse `AUSLOESER` gehört, der Auftrag `dialog.start` des Auftraggebers `dialog` übergeben. Mit `ausloeser.start` wird dann die Endlosschleife gestartet, in der auf den Tastatur-Interrupt gewartet wird, die den Dialog starten soll.

```
start:TASK MAIN;
     meinterminal.init(meinterminal);
     controlc.irpt:=control_c_irpt;
     PUT HOMECLEAR TO display BY A;
     ausloeser.init(ausloeser,dialog,dialog.start);
     dialog.init(dialog,dialog_zweige_start);
     kreuzung.init(kreuzung);
     ausloeser.start(ausloeser);
     ausloeser.beendet(ausloeser);
END;
```

Beispiel 51: Startask des Programmes

```
   programmende:PROC;
     ausloeser.stop(ausloeser);
     meinterminal.close(meinterminal);
   END;
```

Beispiel 52: Beendigung des Programmes

Einer der Dialogzweige ist die Prozedur `programmende` (Beispiel 52). In ihr wird durch `ausloeser.stop` vorbereitet, daß die Interrupt-Warteschleife in `AUSLOESER_ausfuehrung` verlassen wird, und dann in `ausloeser.beendet` darauf gewartet, bis dies geschehen ist.

5 Softwaretechnische Details

5.1 Benutzte Programmierwerkzeuge

Das Programm wurde unter dem Betriebssystem Linux mit Hilfe des Editors Emacs entwickelt. Programmiertechnisch erwies sich als besonders vorteilhaft, daß Bezeichner und kleinere Programmteile sehr schnell mit der Maus an andere Stellen kopiert werden können; dadurch ist es möglich, fast ohne zusätzlichen Aufwand lange, aussagekräftige Bezeichner zu verwenden.

Das PEARL-System war dasjenige der Firma WERUM, Lüneburg, zu dem eine Anpassung des Editors an Entwicklungen für PEARL gehört. Diese Programmierumgebung wurde in einzelnen Punkten geändert, um die Bedienung noch mehr zu vereinfachen.

Das Programm wurde allerdings nicht sofort in PEARL geschrieben, sondern entstand als Top-down-Entwurf, aus dem mit Hilfe eines Programmierwerkzeuges `proker` das PEARL-Programm automatisch erzeugt wird.

```
*/ MODULE(Adaptkreuzung); /*
  #S Systemteil
  #P Problemteil
*/ MODEND; /*

#P */ PROBLEM; /*
  #D Dations
  #T Datentypen und zugehörige Konstanten
  #K Konstanten
  #C Klassen
  #O Objekte
  #F Prozeduren
  #%M Starttask zum Testen
```

Beispiel 53: Anfang des Top-down-Entwurfs

Das Programm besteht aus einem Modul und enthält etwa 1800 PEARL-Zeilen; bei einem derart großen Modul ist es nicht leicht, sich den Überblick über seine Bestandteile zu erhalten und bei Wartungsarbeiten diejenigen Stellen zu finden, an denen Änderungen durchgeführt werden sollen. Das verwendete Top-down-Programmierverfahren erleichtert diese Aufgaben; zusätzliche Übersichts-Dokumentation ist deshalb nicht notwendig.

Beispiele 53 und 54 zeigen den Anfang bzw. ein Detail des Progammentwurfs. Aus Beispiel 53 ist ersichtlich, daß der Programmentwurf mit einem Inhaltsverzeichnis beginnt, das auf Unterverzeichnisse verweist. Beispiel 54 zeigt ein Detail: der Programmabschnitt ist zunächst in Pseudocode formuliert worden. Aus ihm entsteht durch Anwendung des Werkzeuges `proker` der PEARL-Code aus Beispiel 24.

Die mit einem führenden Doppelkreuz `#` versehenen Schlüssel-Codes treten im Entwurf paarweise auf und assoziieren Pseudocode-Anweisungen mit Verfeinerungen bzw. Stücken von PEARL-Code.

Pseudocode und PEARL-Code lasssen sich dabei mischen, wobei die Pseudocodes als Kommentare notiert sind. Beispiel 55 zeigt die Verfeinerung einer Pseudocode-Zeile aus Beispiel 54, die aus reinem PEARL-Code besteht.

```
*/   ADAPTSTEUERUNG_rotgruenzyklus:PROC(steuerung INV REF ADAPTSTEUERUNG,
                                        zyklusbit INV REF BIT); /*
#Co251 Blinken einschalten, blinken, Blinken ausschalten
#Co252 Normalzustand setzen
#Co253 Gelbschalten, anfangsgelbdauer abwarten, rotschalten
#Co254 Wiederhole bis zum Abschalten
#Co255    Warte auf Auslösung des Grünzyklus
#Co256    Belege Kreuzungsfläche exklusiv
                bzw. warte (Interrupts sind unscharf)
#Co257    schalte auf Grün, verändere Status und bereite Ende der Grünphase vor
#Co258    warte auf Ende der Grünphase
#Co259    schalte auf Rot
#Co25a    warte vorgegebene kurze Dauer, gib Kreuzungsfläche frei
     */ END; /* der Grün-Rot-Grün-Rot-usw.-Schleife
#Co25b Synchronisiere mit der anderen Straße
#Co25c schalte für gewisse Dauer gelb, dann blinken, dann aus
     */END; /*
```

Beispiel 54: Top-down-Entwurf eines Teiles der Ampelsteuerung

Die so geschriebenen Programmentwürfe besitzen Baumstruktur: jeder Schlüssel stellt einen Knoten bzw. ein Blatt des Baumes dar. Besonders deutlich wird das in Beispiel 47, das Verweise auf die obersten Knoten Systemteil und Problemteil enthält und in letzterem die Verweise auf die Folgeknoten Dations usw., die sozusagen ein Inhaltsverzeichnis des Problemteiles darstellen.

```
#Co251 */
         steuerung.strasse.blinker.start
         (CONT steuerung.strasse.blinker);
         AFTER steuerung.anfangsblinkdauer RESUME;
         steuerung.strasse.blinker.stop
                (CONT steuerung.strasse.blinker);
         steuerung.strasse.blinker.beendet
            (CONT steuerung.strasse.blinker);/*
```

Beispiel 55: Verfeinerung des Pseudocodes `#Co251` durch PEARL-Anweisungen

Im Gegensatz zu “normalen” PEARL-Programmen enthält dieses Inhaltsverzeichnis keine Hinweise auf Tasks außer auf die Starttask. Alle übrigen Tasks sind in Objekte eingebettet!

Die vom Editor Emacs gebotene Programmierumgebung wurde an dieses Entwurfsverfahren angepaßt: Bei Anklicken eines Schlüssels wie `#Co251` mit der Maus positioniert der Editor automatisch auf die zugehörige Verfeinerung. Auf diese Weise kann ein bestimmter Knoten sehr rasch gefunden und auf dem Bildschirm dargestellt werden.

5.2 Erfahrungen bei der Programmentwicklung und -wartung

Die Programmierung mit Zeigern wird oft als fehleranfällig angesehen; das gilt jedoch nicht für die Programmierung mit Referenzvariablen. Wegen der strengen Typbindung wurden fast alle Schreib- und Flüchtigkeitsfehler im Programm schon bei der Kompilation gefunden, wenn die Möglichkeiten voll ausgeschöpft worden waren, die PEARL90 in dieser Hinsicht bietet.

Zum Beispiel war in einem Fall bei der Vereinbarung eines formalen Parameters, über den wie in Beispiel 7 mit Wertübergabe der Inhalt einer Referenzvariablen (ein Variablenname) in eine Prozedur übergeben werden sollte, das Schlüsselwort `REF` vergessen worden. Der aktuelle Parameter beim Aufruf war der Name eines Objektes; wegen des vergessenen Schlüsselwortes `REF` wurde nicht dieser Name, sondern der Inhalt der betreffenden Variablen in die Prozedur übergeben. Das führte zu einem Laufzeitfehler, dessen Ursache erst nach einer beträchtlichen Suchzeit gefunden wurde. Auf Grund dieser Erfahrung ist an allen Stellen, an denen PEARL90 Referenz-Variable implizit dereferenziert, nach Möglichkeit die explizite Dereferenzierung mit CONT verwendet worden; dadurch können Fehler wie der eben erwähnte schon bei der Kompilation entdeckt werden. Desgleichen wurden alle invarianten Prozedurparameter und Datentyp-Komponenten mit dem Schlüsselwort INV vereinbart.

Da es sich bei dem beschriebenen Programm um den ersten größeren Versuch des Verfassers handelte, objektorientiert mit PEARL90 zu programmieren, weicht die jetzige Version in vielen, auch umfangreichen Einzelheiten von der ersten lauffähigen Version ab.

Als sehr positiv kann vermerkt werden, daß sich die meisten Änderungen wegen des objektorientierten Ansatzes nur auf eine Klasse und deren zugehörige Objekte erstreckten.

In vielen Fällen wurden eine zu einer Klasse gehörende Typvereinbarungen verbessert; natürlich mußten dann auch die INIT-Listen in den Objektdeklarationen geändert werden. Formale Fehler, die bei diesen Wartungsarbeiten entstanden, wurden mit großer Sicherheit schon bei der Kompilation erkannt. Hier machte sich sehr positiv bemerkbar, daß PEARL90 strenge Regeln bezüglich der Verwendung von Datentypen und der Anzahl von Prozedurparametern hat.

5.3 Wünsche bezüglich weiterer Hilfsmittel

Wenn für eine Klasse eine weitere Methode eingeführt wurde, erforderte dies relativ viel Handarbeit, weil die INIT-Listen der zugehörigen Objekt-Vereinbarungen entsprechend ergänzt werden mußten. Wie Beispiel 14 zeigt, enthält eine Objektvereinbarung andererseits nur sehr wenige variable Bestandteile; im Beispiel 14 ist dies lediglich der `zyklusname`. Schon die Verwendung eines Makrogenerators könnte daher sowohl das Schreiben als auch das Ändern eines derartigen Programmes wesentlich erleichtern. Noch größere Erleichterung bei Entwicklung und Wartung könnte durch einen Vorübersetzer erzielt werden, der bei Objektvereinbarungen den PEARL-Code automatisch aus entsprechenden Klassenvereinbarungen erzeugt.

6 Kritische Endbetrachtung

6.1 Ursachen für gute Wartbarkeit

Erfahrungsgemäß entsteht der Löwenanteil - bis zu 90% - der Gesamtkosten eines Programmes bei der Wartung. Deshalb sollte bei der Erstentwicklung eines Programmes vor allem auf gute Wartbarkeit geachtet werden.

Es ist plausibel, daß sich zwei Programmiersprachen in Bezug auf Wartbarkeit der mit ihnen geschriebenen Programme unterschiedlich verhalten. Es ist jedoch schwer, quantitative Aussagen darüber zu gewinnen, weil man dazu die selben Programmieraufgaben von gleich guten Programmierern in beiden Sprachen lösen lassen müßte. Es ist jedoch plausibel und wird durch die Erfahrung bestätigt, daß Programmiersprachen umso wartungsfreundlicher sind, je mehr Irrtümer und Inkonsistenzen bereits bei der Kompilierung erkannt werden können und je weniger sie zu Programmiertricks einladen.

Leichter lassen sich derartige Aussagen machen, wenn es sich zwar um die selbe Sprache, aber unterschiedliche Programmierstile handelt.

Aus den Erfahrungen mit dem hier vorgestellten Programm heraus läßt sich die Schlußfolgerung ziehen, daß die Wartbarkeit eines Programmes wesentlich von folgenden Gegebenheiten abhängt:

1. Der Nutzungsbereich eines Programmbestandteiles sollte möglichst kurz sein.
2. Ähnliche Anweisungsfolgen sollten nur einmal (z. B. als Prozeduren) geschrieben werden.
3. Die Anzahl der unmittelbaren Fernwirkungen im Programm sollte möglichst gering sein.
4. Die indirekten Auswirkungen bei Änderungen eines Programmbestandteiles sollten gering an Zahl und möglichst durchschaubar sein.

Unter Nutzungsbereich wird hier der logisch zusammenhängende Textabschnitt eines Programmes verstanden, in dem der fragliche Programmbestandteil verwendet wird. Der Nutzungsbereich einer Variablen ist höchstens so groß wie der Sichtbarkeitsbereich, in dem der Zugriff auf eine Variable erlaubt ist; im allgemeinen wird aber eine Variable nur in einem Teil des Sichtbarkeitsbereichs in einem bestimmten logischen Zusammenhang benutzt.

Eine Fernwirkung entsteht dadurch, daß es mehrere nicht zusammenhängende Nutzungsbereiche eines Programmbestandteiles gibt, wenn z. B. eine Statusvariable an einer Stelle gesetzt und an einer anderen Stelle abgefragt wird.

Die Änderung einer Typvereinbarung verursacht indirekte Auswirkungen: sie wirkt sich auf alle Variablen-Deklarationen aus, die die Typvereinbarung benutzen, und damit indirekt auch auf die Nutzungsbereiche dieser Variablen.

Im Grunde sind Prozeduren, Strukturierte Programmierung, Datenkapselung und viele andere Bestandteile von Programmiersprachen oder Programmierstilen entwickelt worden, um die oben gesteckten Ziele zu erreichen.

Nach der Deklaration `DCL druck FLOAT(23) GLOBAL;` besteht z. B. der Sichtbarkeits- bzw. Nutzungsbereich für `druck` aus dem gesamten Modul und allen übrigen Moduln des Programmes, in dem Spezifikationen `SPC druck FLOAT(23) GLOBAL;` stehen. Wenn bei der Wartung die Genauigkeit in FLOAT(53) geändert wird, erfordert das zumindest eine Änderung aller Spezifikationen. Es kann aber auch im gesamten Sichtbarkeitsbereich einen Rattenschwanz von Änderungen in Ausdrücken verursachen, in denen auf `druck` zugegriffen wird und die deshalb Typanpassungen erfordern. Die Wartung ist in diesem Fall deshalb schwierig, weil es mehrere Nutzungsbereiche gibt, die sich über Fernwirkungen gegenseitig beeinflussen können.

Es stellt in solchen Fällen schon ein Problem dar, überhaupt diejenigen Stellen zu finden, an denen Änderungen notwendig sind. Strenge Typbindung und der Zwang, Typanpassungen explizit vornehmen zu müssen, sodaß die Stellen für notwendige Änderungen vom Compiler entdeckt werden können, bilden dabei eine wesentliche Hilfe.

In Beispiel 2 ist der Nutzungsbereich für die Variable `druckwert` zwar klein; durch die Benutzung der Modulschnittstelle in anderen Moduln entstehen jedoch Fernwirkungen.

Die Verwendung der Klasse `DRUCK` (Beispiele 3 und 5) für die Deklaration von `kesseldruck` (Beispiel 4) ergibt zunächst eine Verschlechterung der Wartbarkeit: Die Änderung der Genauigkeit bei der Komponente `wert` in Beispiel 3 pflanzt sich auf alle Objekte fort, also auch auf das Objekt `kesseldruck`, und damit indirekt auch auf alle Anweisungen, in denen `kesseldruck` verwendet wird.

```
TYPE KESSEL STRUCT[......,
                   ......,
                   druck INV REF DRUCK,
                   ......];
```

Beispiel 56: Verwendung eines Druckes als Komponente einer Klasse `KESSEL`

Aus diesem Grunde wird im Ampelprogramm nie direkt auf deklarierte Objekte zugegriffen, sondern stets über Referenzen, die Komponenten anderer Klassen sind. Ein Objekt `kesseldruck` wäre damit z. B. Teil eines Objektes `dampfkessel` der Klasse `KESSEL` (Beispiel 57), deren (unvollständige) Typvereinbarung in Beispiel 56 enthalten ist.

```
DCL dampfkessel KESSEL INIT(......,
                            ......,
                            kesseldruck,
                            ......);
```

Beispiel 57: Verwendung des Objektes `kesseldruck` aus Beispiel 4

Auf diese Weise wird erreicht, daß sich eine Änderung der Klasse `DRUCK` nur auf diejenigen Klassen auswirken kann, deren Typvereinbarungen Referenzen auf Objekte der Klasse `DRUCK` enthalten.

Im Verlauf der Entwicklung wurde das Programmbeispiel von ungefähr 1000 Zeilen auf etwa die doppelte Länge ergänzt. Diese Wartungsarbeiten waren relativ gut durchführbar:

1. Die Typvereinbarungen und Methoden einer Klasse stehen unmittelbar benachbart im Programmtext. Der Nutzungsbereich der Typkomponenten besteht lediglich aus den Methoden der Klasse.
2. Die Nutzungsbereiche von Tasks, Semaphoren und Bolts erstrecken sich nur auf die Methoden der Klasse bzw. der Objekte, zu denen sie gehören.
3. Wenn Objekte zusammenwirken, die zwei verschiedenen Klassen angehören, brauchen bei Änderungen nur die Methoden oder Typvereinbarungen dieser Klassen geändert zu werden.
4. Bei Hinzufügung oder Streichung von Typkomponenten brauchen zusätzlich nur die INIT-Listen in den Objekt-Deklarationen geändert zu werden.
5. Die Bezeichner von Objekten und der zugehörigen Tasks, Semaphore usw. werden nur bei der Deklaration der Objekte selbst und in INIT-Listen verwendet.

In den drei ersten Fällen betrugen die Programmabschnitte, in denen Änderungen vorgenommen werden mußten, höchstens 2 Bildschirmseiten. In den letzten beiden Fällen waren die betreffenden INIT-Listen zwar über den gesamten Modul verstreut, aber leicht zu finden (spätestens dadurch, daß der Kompilierer Inkonsistenzen bezüglich Anzahl oder Typen meldete), auszufüllen bzw. zu ändern.

Offensichtlich würde es sehr leicht sein, beliebig viele weitere Kreuzungen durch das Programm steuern zu lassen: erforderlich wären nur weitere Deklarationen entsprechender Objekte und das Ausfüllen der in ihnen enthaltenen INIT-Listen.

Die objektorientierte Programmierung bewährt sich besonders bei der Überwachung von E/A-Ausführungszeiten und bei der Reaktion auf Ein/Ausgabe-Exceptions: ohne die Möglichkeit, einem entsprechenden Überwacher-Objekt bzw. einem Terminal-Objekt Referenzen auf die zu überwachenden E/A-(Prozedur-)Methoden zu übergeben, hätte diese Aufgabe an fünf verschiedenen Stellen gelöst werden müssen.

Der größte Vorteil dürfte aber darin liegen, daß praktisch alle Tasks in Objekte eingebettet sind und daß ihre Aktivierung und Koordination nur einmal in den betreffenden Methoden der Klassen programmiert zu werden braucht. So brauchen z. B. die Vorsichtsmaßnahmen, die bei der Aktivierung einer Task zu Ausgabe der Ampelbilder getroffen werden müssen (Kapitel 4.7) nur ein einziges Mal formuliert zu werden und haben dann Geltung für alle Ampelbilder beliebig vieler Kreuzungen.

Im Beispielprogramm sind praktisch alle Tasks in Objekte eingebettet. In einigen Fällen (z.B. bei den Klassen `ZYKLUS` und `UEBERWACHER`) können sie mit der Durchführung beliebiger Methoden beauftragt werden. Auf diese Weise sind Klassen entstanden, die auch bei der Entwicklung neuer Programme verwendet werden können.

6.2 Grenzen der Objektorientierung in PEARL90

6.2.1 Dynamische Kreierung von Objekten

In objektorientierten Sprachen wie C++ oder Java werden neue Objekte dynamisch erzeugt; PEARL90 hingegen bietet in dieser Hinsicht keine dynamische Speicherverwaltung, sondern alle Objekte müssen explizit deklariert werden. Bei der dynamischen Erzeugung von Objekten besteht nämlich immer die Gefahr, daß ein Programm aus Mangel an verfügbarem Speicher suspendiert oder abgebrochen werden muß. Für ein PEARL-Programm, das keine rekursiven Prozeduraufrufe enthält, läßt sich hingegen immer der höchstmögliche Speicherbedarf ermitteln.

6.2.2 Vererbung

Die Klasse `ADAPTSTEUERUNG`, durch die die Ampelsteuerung im Beispielprogramm realisiert worden ist, ist relativ kompliziert; eine einfachere Ampelsteuerung der Klasse `SIMPLESTEUERUNG` könnte mit Rot- und Grünphasen fester Dauer arbeiten, die einander abwechseln und nicht durch Induktionsschleifen beeinflußt werden. Eine derartige Steuerung würde einer Blinksteuerung (Klasse `BLINKER`) sehr ähnlich sein, bei der ja auch zwei Phasen - Gelb und Aus - einander abwechseln. Beide würden sich nur durch die Methode zur Phasenumschaltung unterscheiden, während die Methoden `init`, `start`, `stop` und `beendet` gleiche Aufgaben ausführen müßten; auch `SIMPLESTEUERUNG` würde Referenzen auf Objekte der Klasse `ZYKLUS` enthalten. Es wäre daher naheliegend, die genannten Methoden nicht für jede der beiden Klassen als Prozeduren gesondert zu schreiben, sondern nur einmal für beide Klassen gemeinsam.

In objektorientierten Sprachen wird dieser Gedanke dadurch gelöst, daß Kindklassen gebildet werden können, die von ihren Mutterklassen Eigenschaften und Methoden erben können. Die Klassen `BLINKER` und `SIMPLESTEUERUNG` könnten z. B. Kindklassen einer gemeinsamen Mutterklasse `STEUERUNG` sein, von der sie die genannten Methoden erben würden.

```
STEUERUNG_start:PROC(alleszeiger INV REF STRUCT[]);
   DCL steuerung REF STEUERUNG;
   steuerung:=alleszeiger;
   steuerung.zyklus.start(CONT steuerung.zyklus);
   CONT steuerung.status:='1'B;
END;
```

Beispiel 58: Startmethode für eine Mutterklasse `STEUERUNG`

Bei dem hier beschriebenen Verfahren, mit PEARL90 objektorientiert zu programmmieren, bereitet die Vererbung jedoch Schwierigkeiten: eine Methode wird ja durch eine Prozedur dargestellt, der als erster Parameter eine Referenz auf dasjenige Objekt übergeben wird, für das sie arbeiten soll. Auch wenn eine Methode (der Prozedurkörper) bei zwei Klassen identisch ist, unterscheiden sich die Prozedurköpfe im Typ der Referenzen.

Einen Ausweg bieten die nicht typgebundenen Zeiger `REF STRUCT[]`. Beispiel 58 zeigt die Verwendung eines derartigen Zeigers für eine Methode `start` der Mutterklasse `STEUERUNG`, die sowohl für die KindKlasse `BLINKER` als auch für `SIMPLESTEUERUNG` verwendet werden kann.

Der Inhalt der nicht-typgebundenen Referenz-Variablen `alleszeiger` (der Name eines Objektes aus einer der Kindklassen von `STEUERUNG`) wird einer Referenzvariablen vom Typ der Mutterklasse `STEUERUNG` übergeben. Dadurch kann auf die einzelnen Komponenten der Variablen zugegriffen werden, auf die `alleszeiger` zeigt.

Voraussetzung hierbei ist selbstverständlich, daß die Typvereinbarungen für die Mutterklasse und deren Kindklassen bezüglich der Typen und der Reihenfolge der Komponenten gleich sind. Die Kindklassen dürfen allenfalls am Ende der Typvereinbarungen zusätzliche Komponenten enthalten. Insofern dürfte diese Art der Vererbung relativ fehlerträchtig sein, insbesondere, wenn bei einer der Klassen im Verlauf der Wartung Komponenten geändert werden. Problemlos wäre die Vererbung nur, wenn es, wie z. B. in Java, spezielle Sprachelemente dafür geben würde.

Es kann auch bezweifelt werden, daß Vererbung in einer objektorientierten Sprache für die Automatisierungstechnik eine große Rolle spielen wird: bei Objekten, welche Teile von technischen Anlagen softwaremäßig spiegeln, werden sich nur schwer gemeinsame Mutterklassen mit gemeinsamen Methoden bilden lassen: zwar gehören Verbrennungsmotoren und Elektromotoren der gemeinsamen Klasse Motoren an, dürften aber nur wenige Methoden (z. B. Drehzahlmessung) gemeinsam haben.

6.2.3 Zugriffschutz

In objektorientierten Sprachen wie Java können Variable, die in ein Objekt eingekapselt sind, unter Zugriffsschutz gesetzt werden dergestalt, daß sie nicht von außerhalb des Objektes gelesen oder verändert werden können, sondern nur durch die Methoden des Objektes. Im Beispiel 3 bzw. 5 können demgegenüber die Komponenten `wert` und `zugriff` auch ohne Benutzung der Methoden `setzen` und `lesen` verändert bzw. gelesen werden, da in PEARL die Komponenten einer modulglobalen Variablen immer für den gesamten Modul sichtbar sind; nur wenn das Objekt wie in Beispiel 2 in einem eigenen Modul enthalten ist, läßt sich auch in PEARL ein Schutz vor Zugriffen von außen erreichen.

Das Beispielprogramm enthält knapp 40 Objekte und müßte aus dieser Anzahl Moduln bestehen, wenn von den Objekten nur die Methoden-Schnittstellen nach außen sichtbar sein sollten; es ist einzusehen, daß das nicht praktikabel ist. Wenn man das Programm in mehrere Moduln aufteilen würde, würde man das unter dem Gesichtspunkt tun, daß logisch zusammengehörende Teile in je einem Modul zusammengefaßt werden sollten. Im Beispielprogramm wären das zwei Komplexe: die komplette Kreuzung und die zur Terminal-Ein/Ausgabe gehörenden Teile. Bei dem Kreuzungsmodul würde man dann nur die Methoden zum Ein- und Ausschalten der Steuerung und des Blinkers von außen sichtbar machen; damit wäre ausreichender Schutz vor mißbräuchlicher Benutzung von Objektkomponenten gewährleistet, weil sie nur im betreffenden Modul vorkommen könnte und dort leicht auffindbar wäre.

Die Bezeichner der deklarierten Objekte dürfen nämlich nur in den INIT-Listen und in den Prozedur-Aufrufen innerhalb der Tasks und Dialog-Schnittstellen vorkommen.

6.3 Laufzeit-Effizienz

Im Programmbeispiel wird das Blinken der Ampeln einer Straße durch Rücksetzen eines Bits in einem Objekt der Klasse `ZYKLUS` durch Aufruf der Methode `stop` abgeschaltet. Dabei besteht eine Objektschachtelung `KREUZBLINKER - BLINKER - ZYKLUS`. Auf Ebene des Bediendialogs wird das Blinken durch Aufruf einer parameterlosen Prozedur bewirkt, die ihrerseits die Methode `stop` des betreffenden Objektes der Klasse `KREUZBLINKER` aufruft, die dann die gleichnamige Methode eines Blinkers benutzt, usw.. Das Rücksetzen des einzelnen Bits erfordert deshalb die Ausführung von drei Prozeduraufrufen, also einen beträchtlichen Überhang an Maschinenbefehlen und Rechenzeit.

Dieser Überhang kann jedoch verschmerzt werden, weil bei der Bedienung keine Restriktionen bezüglich der Ausführungszeit existieren. Bedenklicher wäre ein derartiger Überhang, wenn er in häufig ausgeführten Schleifen aufträte.

In Beispiel 24 existiert innerhalb einer Schleife die Programmzeile

```
steuerung.strasse.kreuzung.flaeche.besorgen
                                    (CONT steuerung.strasse.kreuzung.flaeche);
```

In ihr findet ein Prozeduraufruf statt, bei dem insgesamt neunmal dereferenziert wird. Er bewirkt letztendlich die Ausführung einer einzigen `REQUEST`-Anweisung, bei der noch einmal dereferenziert wird.

Dieser Überhang an Rechenzeit, der im Beispiel durch die Programmierung mit Objekten entsteht, ist hier sicher zu verschmerzen, weil es bei den Abläufen in der Ampelsteuerung nicht auf ein paar Millisekunden ankommt und die Task ohnehin später in der Schleife suspendiert wird; bei zeitkritischen Anwendungen könnte ein derartiger Überhang jedoch durchaus stören. Er ist der Preis, der für die leichtere Programmierung und Wartung sowie für die Wiederverwendung vohandener Programmteile gezahlt werden muß.

Im obigen Beispiel kann übrigens leicht optimiert werden, indem

```
steuerung.strasse.kreuzung.flaeche
```

vor der Schleife einer Hilfsvariablen zugewiesen wird, die dann in der Schleife benutzt wird. Ein guter optimierender Kompilierer würde das automatisch tun.

6.4 Abschließende Würdigung

Das hier vorgestellte Beispiel zeigt, daß ein PEARL90-Programm als Netz von Objekten programmiert werden kann, die aus Klassen abgeleitet sind. Dieser Ansatz bietet die Vorteile, die die objektorientierte Programmierung bezüglich Programmentwurf, Wiederverwendung von Programmteilen und insbesondere bezüglich der Programmwartung bietet.

Programmdetails wie die Koordination von Tasks bei Benutzung gemeinsamer Variablen und die Behandlung von Ausnahmen lassen sich in diesem Ansatz leichter lösen als bei “normaler” Programmierung. Insofern dürfte der Ansatz auch manche Vorschläge für zusätzliche Sprachelemente obsolet machen, die die Zuverlässigkeit und Sicherheit von PEARL-Programmen erhöhen sollen.

Die Umsetzung des Klassen-Objekt-Ansatzes in ein lauffähiges Programm erfordert jedoch noch relativ viel Aufwand, der durch entsprechende Hilfsmittel vermindert werden könnte. In einem ersten Schritt müßte ein Vorübersetzer für aufgesetzte objektorientierte Sprachelemente hilfreich sein. Wünschenswert wäre die Integration derartiger Sprachelemente in die Sprache.

Inhaltsverzeichnis