Die synchrone Programmiersprache E S T E R E L

Samuel Eickelberg Johannes Dahlke

Ausarbeitung zum Vortrag am 25 Feb. 2013 über die 1992 von Berry und Gonthier in der Fachzeitschrift „Science of Computer Programming“veröffentlichte wissenschaftliche Arbeit „The ESTEREL synchronous programming language: design, semantics, implementation“[7] .

Kurzreferat Esterel ist eine synchrone, imperative und nebenläufige Sprache zur Programmierung reaktiver Systeme. Reaktive Systeme sind Softwaresysteme, welche mit ihrer Umgebung in permanenter Interaktion stehen [17] . Sie finden Einsatz in der Automobilindustrie, Luftfahrt, Medizintechnik sowie Haushalts- und Unterhaltungselektronik. Synchronen Programmiersprachen liegt die Synchronitätshypothese zugrunde: Jede Reaktion auf ein Signal erfolgt unverzüglich und zudem atomar. Somit finden Eingaben sowie die dazugehörigen Ausgaben zu einer Zeit statt. Dem liegt die Annahme zugrunde, dass die Taktgeschwindigkeit des ausführenden Rechners unbegrenzt ist.
In dieser Seminararbeit über [7] gehen wir auf die wesentlichen Esterel-Eigenschaften synchron, parallel und deterministisch ein, zeigen den grundlegenden Programmaufbau sowie stellen den Befehlssatz Basic Esterel und dessen Erweiterung Plain Esterel vor. Abschließend wird eine Verhaltensemantik und eine Ausführungssemantik von Esterel betrachtet.

Themenbereiche D.3.1 [**Programming languages**]: Formal Definitions and Theory — Syntax; semantics; D.3.2 [**Programming languages**]: Language Classifications — Synchronous Languages; Esterel

Stichwörter Synchrone Programmiersprache, reaktive Systeme, Echtzeitsysteme, endliche Automaten, Synchronitätshypothese, *sharing law*, Semantik

1 Einleitung

Diese Seminararbeit handelt von der synchronen, deterministischen und nebenläufigen Programmiersprache Esterel. In dieser Ausarbeitung geben wir einen Überblick über das Papier „The ESTEREL synchronous programming language: design, semantics, implementation“[7] von Berry und Gonthier, beziehen uns dabei aber auch auf weitere Publikationen. Insbesondere seien hier die Arbeiten „The Synchronous Hypothesis and Synchronous Languages“[25] von Potop, de Simone und Talpin sowie „The Constructive Semantics of Pure Esterel“ [3] von Berry und „The ESTEREL Synchronous Programming Language and its Mathematical Semantics“ [5] von Berry und Cosserat genannt.

Dieses Kapitel behandelt einführend einige Grundlagen der Welt reaktiver Systeme hinführend auf die Synchronitätshypothese, auf welche sich alle synchronen Programmiersprachen stützen.

1.1 Reaktive Systeme

Der Begriff „reaktive Systeme“ wurde von Harel und Pnueli [17] für Softwaresysteme eingeführt, welche mit ihrer Umgebung in permanenter Interaktion stehen. Softwaresysteme werden als reaktive Systeme klassifiziert, wenn

ihre Kernfunktionen überwiegend von reaktiven Programmen gesteuert werden.

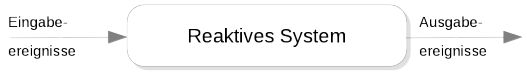

Abbildung 1.1: Reaktives System als Blackbox

Ein besonderes Merkmal reaktiver Systeme ist das eingabegesteuerte Verhalten. Ein reaktives System wartet auf zumeist äußere aber auch innere Ereignisse, reagiert auf diese durch Verarbeitung der Eingabe, liefert das produzierte Ergebnis wieder in die ursprüngliche Umwelt zurück und wartet nun erneut auf weitere Eingaben (siehe Abbildung 1.1). Auf diese Weise wird das System durch die Umgebung synchronisiert.

Das zu unbeschränkter Laufzeit führende Nichtterminieren des Systems nach erfolgter Berechnung des Ergebnisses ist ein weiteres Unterscheidungsmerkmal reaktiver Systeme in Abgrenzung zu herkömmlichen transformationellen Systemen [27] .

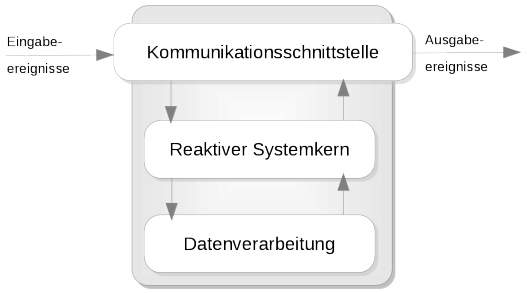

Abbildung 1.2: Dreischichtiger Aufbau reaktiver Systeme

Beispiele für reaktive Systeme gibt es zahlreiche: Von Maschinensteuerungen zum Beispiel bei CNC-Maschinen, diverse Steuerungen im PKW wie Airbag-Steuerung, Antiblockiersystem und elektronisches Motormanagement über Echtzeitbetriebssysteme bis hin zu Videospielen und Digitaluhren. Betriebssystemtreiber und Schnittstellen für Eingabegeräte wie Maus oder Tastatur sind Beispiele für in komplexere Systeme eingebettete reaktive Programme.

Reaktive Systeme lassen sich in ihrem Aufbau in drei Schichten unterteilen. Die erste Schicht bildet die Schnittstelle zur Umwelt. Über die Schnittstelle nimmt das System seine Eingaben entgegen und gibt seine Ergebnisse aus.
In die Ebene der Schnittstelle (*interface*) fallen Aufgaben wie die Behandlung von Interrupts, das Lesen von Sensoren und das Aktivieren von Aktoren zum Beispiel zur Steuerung mechatronischer Systeme. Kurz gesagt: Die Schnittstelle setzt physikalische Eingabeereignisse der Umwelt in interne logische Signale um und umgekehrt.
Die reaktive Systemlogik (*reaktive kernel*) ist in der zweiten Schicht beherbergt. Der Systemkern steuert Berechnungen und Ausgaben in Abhängigkeit der jeweiligen Eingaben.
Die Datenverarbeitungsschicht (*data handling layer*) führt die von der Systemlogik zugewiesenen Anweisungen aus.

Im Weiteren liegt der Fokus auf der Ebene des Systemkerns, denn die synchrone Programmiersprache ESTEREL konzentriert sich lediglich auf diese zentrale Komponente, sodass für die Spezifikation der Schnittstelle sowie für Berechnungen eine *Hostsprache* wie zum Beispiel C eingesetzt werden muss.

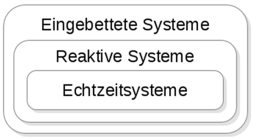

Abbildung 1.3: Klassifikation eingebetteter Systeme

Wie in Abbildung 1.3 veranschaulicht sind Reaktive Systeme eine echte Untermenge eingebetteter Systeme sowie eine echte Obermenge von Echtzeitsystemen. ESTEREL ist als eine Programmiersprache für Echtzeitsysteme entwickelt worden. Zu den Echtzeitanforderungen an Echtzeitsysteme gehört auch, dass diese innerhalb definierter Zeitschranken Eingaben verarbeiten.

1.2 Determinismus und Nebenläufigkeit

Automaten eignen sich zum Erstellen einfacher und überschaubarer reaktiver Systeme, welche sequentiell arbeiten und damit deterministisch sind. Reaktive Systeme sind aber oft sehr komplex. Sie können aus mehreren und auf verschiedene Rechner verteilte unabhängig voneinander

Werkzeuge zur Entwicklung von reaktiven Echtzeitsystemen

Automaten Deterministische endliche Automaten (*deterministic finite automaton*, DFA) [19] finden in kleinen reaktiven Systemen wie zum Beispiel Steuergeräten Verwendung. Automaten sind für ihre effiziente Laufzeit bekannt und zudem mathematisch definiert. So nutzen Berry und Gonthier [7, S. 140] auch deterministische Automaten, um Esterel-Programme auf Echtzeitsystemen auszuführen, da die Geschwindigkeit der Esterel-Interpreter, welche auf der in Abschnitt 3.4 „Ausführungssemantik“ auf Seite 17 vorgestellten Semantik basieren, zwar für reaktive Systeme akzeptabel ist, nicht aber hinreichend für Echtzeitsysteme. Wesentlicher Nachteil der Automaten ist, dass diese mit zunehmender Größe schnell unübersichtlich und damit fehleranfällig sowie schwer wartbar und erweiterbar werden. Zudem ist eine Überführung mehrerer deterministischer Automaten in einen nebenläufigen Automaten (concurrent finite automaton, CFA) [20] zwar möglich aber sehr aufwändig.

Petrinetze Programmierbare Kontroller sind ein Anwendungsgebiet für Petrinetze [11] . Einfache Nebenläufigkeit lässt sich mit auf Petrinetzen basierende Programmierwerkzeugen realisieren, jedoch ist das *Debuggen* und Programmieren sehr unkomfortabel.

Asynchrone Programmiersprachen Bekannte asynchrone Programmiersprachen für Echtzeitsysteme sind Ada (benannt nach Ada Lovelace) [26] , Communicating Sequential Processes (CSP) [18] , Occam [22] und RTL/2 [2] und haben gemein, dass sie nichtdeterministisch sind. Des Weiteren ermöglichen sie es dem Programmierer in der Regel alle der drei in Abbildung 1.2 dargestellten Ebenen eines reaktiven Systems in ein und derselben Sprache zu schreiben.

Synchrone Programmiersprachen Neben der imperativen Sprache Esterel gehören unter anderem die graphischen Sprachen Argos [21] und SyncCharts [1] sowie die deklarativen Datenflusssprachen Lustre [16] und Signal [12] zu der Familie der synchronen Programmiersprachen.

laufende Teilkomponenten bestehen, welche untereinander sowie mit ein oder mehreren Benutzern kommunizieren [13] .

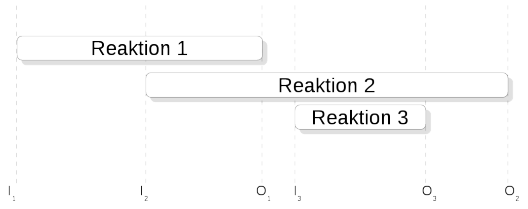

Abbildung 1.4: Szenario zum Ablauf mehrerer Reaktionen

Mit Petrinetzen sowie asynchronen Programmiersprachen lassen sich nebenläufige reaktive Systeme realisieren, welche allerdings nichtdeterministisch sind. Eingabesignale können das System jederzeit erreichen und gerade bei Echtzeitsystemen ist eine unverzügliche Reaktion erforderlich. Zeitnah eintreffende Signale können, wie Abbildung 1.4 zeigt, zu Überschneidungen von an sich unabhängigen Reaktionen führen. Parallel ablaufende Reaktionen führen zu Konkurrenz um die Ressourcen, denn Interaktion im nebenläufigen System geht mit Abhängigkeiten einher.
Ein Beispiel ist Kooperation der Komponenten mittels gemeinsamer Variablen (*shared variables*)[14] . Hierbei kann konkurrierender Zugriff weiteren Nichtdeterminismus im Programmablauf bewirken.
Aber auch das zeitgleiche Auftreten von Signalen kann nichtdeterministisches Verhalten bewirken. Fährt zum Beispiel ein Personenaufzug bei gleicher Ausgangslage und gleichzeitiger Anforderung von zwei Personen in unterschiedlichen Stockwerken stets dieselbe Etage an? [23]

Ein reaktives System verhält sich deterministisch, wenn es auf eine bestimmte Sequenz an Eingabeereignissen stets mit der gleichen Abfolge an Ausgabeereignissen reagiert.

Das Ausschließen von nichtdeterministischen Verhalten ist offensichtlich sinnvoll, da deterministische Programme um einiges einfacher zu schreiben und zu analysieren sowie gegebenenfalls zu *debuggen* sind als nichtdeterministische Programme. Die meisten reaktiven Systeme lassen sich in deterministische nebenläufige Systeme überführen.

Um Nichtdeterminismus in nebenläufigen Systemen zu unterbinden, ist eine Synchronisation der ansonsten zeitlich entkoppelten parallel laufenden Subsysteme erforderlich. Synchrone Programmiersprachen wie ESTEREL schließen Nichtdeterminismus über die ihnen allen gemeinsame Synchronitätshypothese aus und unterstützen dennoch Nebenläufigkeit.

1.3 Synchronitätshypothese

Die Synchronitätshypothese besagt: Jede Reaktion auf ein Signal erfolgt unverzüglich (*zero-delay* Hypothese) und zudem atomar. Dem liegt die Annahme zugrunde, dass die Taktgeschwindigkeit des ausführenden Rechners unbegrenzt ist. Hieraus wiederum leitet sich die „control takes no time“-Hypothese ab die besagt, dass das Ausgabeereignis O dem Eingabeereignis I ohne Zeitverzug folgt, also die Reaktion als Ganzes in einem Augenblick ($\Delta t = t_I - t_O = 0$) erfolgt. Der Bezug dieser Hypothese erstreckt sich von der Anweisungsausführung über die Handhabe von Prozessen sowie der Interprozesskommunikation (IPC) bis hin zur Datenverarbeitung. So erfolgt die Interprozesskommunikation in synchronen Programmiersprachen unverzüglich durch *Broadcast*; alle Prozesse haben die gleiche Sicht auf ihre Umgebung.
Anweisungen in synchronen Programmiersprachen nehmen nur genau dann Zeit in Anspruch, wenn sie zum Zwecke des Zeitverzugs ($\Delta t > 0$) eingesetzt werden. ESTEREL entscheidet hier zwei Klassen von Anweisungen. Erstens imperative Standardanweisungen zur Steuerung des Programmablaufs mittels Kontrollstrukturen wie bedingte Verzweigung, Schleifen oder Traps sowie primitive Operationen wie Zuweisung oder Addition und auch Nebenläufigkeit; diese schlagen sich nicht in der Laufzeit nieder. Zweitens Anweisungen mit Zeitbezug wie `delay`, dessen einziger Zweck gerade die Inanspruchnahme von Zeit ist, Trigger wie **`await`** und Watchdogs wie zum Beispiel das **`do`**...**`watching`**-Konstrukt. Im Gegensatz zu asynchronen Sprachen verfügen synchrone Sprachen nicht über interne Zeitgeber. Das Programm reagiert nur und ausschließlich auf externe Impulse. Damit erübrigt sich bei synchronen Sprachen die Frage nach der Definition von Zeiteinheiten sowie daraus resultierenden Fragen wie zum Beispiel:

$$\texttt{delay(2s) + delay(3s)} \stackrel{?}{=} \texttt{delay(5s)}$$

In OCCAM zum Beispiel kann man sicher nicht von Gleichheit ausgehen, da das asynchrone OCCAM Eingangssignale mittels Nachrichten weiterleitet [5] . In synchronen Programmiersprachen können vielfältige Zeitformate (*multiform time*) genutzt werden, deren Taktung jedoch stets auf externe Ereignisse zurückzuführen ist. Damit entspricht die Anweisung **`await`** 30 `MILLISECOND` exakt 30 Millisekunden bei einem garantiert millisekündlichem Auftreten des Signals `MILLISECOND` und **`every`** 1000 `MILLISECOND` **`do emit`** `SECOND` **`end`** sendet genau alle 1000 Millisekunden das Signal `SECOND`.

2 Syntax

Im Vergleich zu SIGNAL oder Lustre, die beide den deklarativen Programmiersprachen zuzurechnen sind, verfolgt ESTEREL einen imperativen Ansatz. ESTEREL wurde entwickelt, um reaktive Systeme zu programmieren. Jedoch ist ESTEREL keine vollständige Programmiersprache; ausschließlich mit ESTEREL läßt sich noch kein lauffähiges reaktives Softwareprodukt erstellen. Die Sprache ist auf die Einbettung in eine *Hostsprache* angewiesen.
ESTEREL unterscheidet zwei Befehlssätze, auf dessen Unterschiede im Verlauf dieses Berichts noch genauer eingegangen wird: Den einfachen Befehlssatz BASIC ESTEREL und den erweiterten Befehlssatz PLAIN ESTEREL, wobei ersterer eine echte Untermenge von letzterem ist.

Eine typische ESTEREL-Anweisung ist:

```
1  do
2    every STEP do
3      emit JUMP
4    end
5  watching 100 METER
```

Esterel-Code 2.1: Typische Anweisung in ESTEREL [7]

Haupteigenschaften von ESTEREL

DAYARATNE, ROOP und SALCIC nennen in „Direct Execution of Esterel Using Reactive Microprocessors“ vier Merkmale [9, S. 2] der synchronen Sprache ESTEREL, welche die besondere Eignung von ESTEREL als Entwurfssprache für reaktive eingebettete Systeme hervorheben:

Synchrones Zeitmodell Besonders wichtig für die Einhaltung und Umsetzung der Synchronitätshypothese ist die Einführung eines synchronen, diskreten Zeitmodells. Dieses Modell unterteilt die Ausführungszeit in sogenannte *Ticks*, welche die kleinsten unteilbaren Zeiteinheiten darstellen. Die Ausführung aller ESTEREL-Anweisungen ohne Zeitbezug erfolgt innerhalb eines *Ticks*.

Präemption und Prioritätsauflösung ESTEREL verfügt über Befehle, welche eine zeitweise Unterbrechung der Bearbeitung eines Codeblocks zugunsten eines auftretenden Ereignisses ermöglichen. Die Priorisierung des Ereignisse erfolgt statisch mittels des `abort`-Mechanismus.

Nebenläufigkeit ESTEREL setzt die synchrone Nebenläufigkeit mittels synchronem Rundruf (*broadcast*) um, sodass das Emittieren eines Signals zeitgleich – innerhalb desselben *Ticks* – allen anderen *Threads* bekannt ist.

Verzögerungsfreiheit Das in der Synchronitätshypothese formulierte Model der Verzögerungsfreiheit (*zero-delay*) zwingt das in einer Reaktion verarbeitete Ausgabeereignis in die Zeitspanne des *Ticks*, in welchem das die Reaktion auslösende Eingangssignal das System erreicht hat.

In eine natürliche Sprache übersetzt heißt dies: „Springe jeden Schritt innerhalb von 100 Metern.“ Wir sehen hier zwei Schleifen. Die innere Schleife sendet bei jedem Vorkommen des Signals `STEP` das Signal `JUMP` aus. Die äußere Schleife definiert die Abbruchbedingung. Das Konstrukt wird nach dem 100sten Auftreten des Signals `METER` verlassen.

Wie bereits erwähnt lassen sich ESTEREL-Anweisungen in zwei Klassen unterteilen: Klassische imperative Anweisungen wie zum Beispiel Zuweisungen, Schleifen oder Ausnahmen und temporale Anweisungen wie Trigger, Watchdogs oder Zeitschleifen. ESTEREL unterscheidet auch zwischen klassischen Variablen, die nur innerhalb des Geltungsbereiches (*scope*) von ESTEREL bekannt sind, und Signalen, die verwendet werden, um zwischen nebenläufigen Prozessen, sowie mit der „Außenwelt“ (*Hostsprache*) zu kommunizieren.

2.1 Modul

Ein ESTEREL-Programm besteht aus mindestens einem Modul. Ein Modul ist ein in sich unabhängiges Programmfragment. Es stellt eine Einheit von Verhalten dar [23] . Größere reaktive Systeme sind eine Komposition aus einer Vielzahl an Modulen. Jeglicher Programmcode muss innerhalb eines Moduls notiert werden. ESTEREL kennt keinen globalen Geltungsbereich für Daten. Die Deklaration von Signalen und Variablen erfolgt stets lokal innerhalb eines Modules. Das Aussenden eines Signals hingegen kann nicht gerichtet erfolgen sondern ist immer allen Modulen bekannt.

Module dienen der Modularisierung von ESTEREL-Programmen sowie der Wiederverwendbarkeit von Programmteilen. Hierzu wurde in PLAIN ESTEREL die Direktive `copymodule` eingeführt, welche in späteren ESTEREL-Versionen durch das Schlüsselwort `run` ersetzt wurde[28] . Die Syntax und Aufbau erinnert an das *Unit*-Konzept aus WIRTHs PASCAL [29] und dessen Nachfolger MODULA [30] – beides klassische imperative Programmiersprachen. Noch treffender ist der Vergleich mit C-Makros. Der ESTEREL-Compiler ersetzt in `copymodule M` einfach den Namen eines Modules `M` im Code durch die Definition des Modules `M`.

Die Syntax für Module ist:

```
1  % this is a line comment
2  module MODULE_NAME :
3    declaration_part % signals, vars, ...
4    body % any executable statement
5  % end of module
```

Esterel-Code 2.2: Syntax für Module [7, 23]

Hier sehen wir bereits einige Konventionen von

ESTEREL. *schlüsselwörter* werden mit Kleinbuchstaben und *BEZEICHNER* in Großschrift notiert. Die im Deklarationteil aufgeführten Datenobjekte der Datenverarbeitungsschicht, Signale und Sensoren definieren die Schnittstelle des Modules. Alle Objekte sind vor der ersten Verwendung im Modulrumpf zu deklarieren. Der Modulrumpf besteht aus einer Anweisung des ESTEREL-Befehlssatzes, in welcher gegebenenfalls verschachtelt weitere Anweisung aufgeführt sein können. In der neueren Sprachversion 7 [28] wird auch das Ende des Moduls durch ein abschließendes `end module` gekennzeichnet.

2.2 Parallelität

Der Operator für Parallelität in ESTEREL ist ||. Wenn `P1` und `P2` jeweils ESTEREL-Programme sind, dann ist es auch die Parallelausführung `P1 || P2` [23] . Eingabeereignisse, die `P1` erreichen, erhält auch `P2`. Ebenso wie für Module gilt auch generell für Programmfragmente: Alle Ausgaben sind öffentlich, und zwar im selben Moment, in dem die Ausgaben signalisiert wurden, und genau für diesen Augenblick. Die parallele Ausführung beider Programmfragmente `P1` und `P2` ist beendet, sobald sowohl `P1` als auch `P2` terminieren. Während der parallelen Ausführung können sich `P1` und `P2` keine Variablen oder Daten teilen.

2.3 Datendeklarationen

BASIC ESTEREL verfügt im Wesentlichen lediglich über zwei primitive Datentypen: `integer` und `boolean`. In der Erweiterung PLAIN ESTEREL stehen darüber hinaus noch die Datentypen `string` und `float` zur Verfügung. Zusammengesetzte Datentypen wie `array` oder Verbundtypen wie `record` sind nicht auf der Kernelebene (siehe Abbildung 1.2 auf Seite 2) verfügbar. Die Behandlung von Daten mit komplexeren Strukturen erfolgt abstrakt über den Aufruf von Datenbehandlungsroutinen, welche in der Datenverarbeitungsebene mittels *Hostsprache* definiert werden. Als Beispiel die Deklaration zweier abstrakter Typen `DOUBLE` und `TIME`:

```
type DOUBLE, TIME;
```

Die Deklaration von Konstanten kann mittels der primitiven Typen ...

```
constant FIX_NUMBER : integer;
```

... aber auch über zuvor definierte abstrakte Typen erfolgen.

```
constant PI : DOUBLE, NOON : TIME;
```

Ebenso wird zur Deklaration von Funktionen auch nur deren Signatur im ESTEREL-Code notiert:

```
function SQRT(DOUBLE) : DOUBLE;
function EQUALS(TIME,TIME) : boolean;
```

Die Definition der Funktionen erfolgt dann in der gewählten *Hostsprache*, wobei eine von Seiteneffekten freie Implementierung angenommen wird. Daher ist auch nur die Angabe des Parametertypes in der Signatur notwendig.

Ähnlich wie in PASCAL [29] sind Prozeduren auch in ESTEREL Subroutinen, welche keinen Rückgabewert liefern.

```
procedure INC_TIME(TIME)(integer);
```

In der Signatur der Prozeduren folgen zwei Parameterlisten auf den Bezeichner. In der ersten Liste erfolgt die Parameterübergabe als Referenz (*call by reference*). Die zweite Liste ist eine Sequenz von Werteparametern (*call by value*). Die Verwendung von Referenzparameter kann Nebenwirkungen mit sich bringen.

2.4 Schnittstellen

Im Deklarationsteil eines Modules werden Signale und Sensoren als Schnittstelle zur Außenwelt deklariert. Signale signalisieren externe Ereignisse wie Interrupts, Zeitgeber, Tastendrücke oder ankommende Nachrichten augenblicklich. Das Signalisieren eines solchen Ereignisses wird in ESTEREL ein *Tick* genannt. Ausgelöste Signale können auch wertbehaftet sein. Mittels des Operators `?` lässt sich der augenblickliche Wert eines Signales `S` mit der Anweisung `?S` ermitteln. Der Wert eines Signales kann sich ausschließlich mit dem Auftreten eines *Ticks* ändern.

BASIC ESTEREL untererscheidet bezüglich der Modulschnittstelle zwei Typen von Signalen: Eingabe- und Ausgabesignale. Eingabesignale sind in BASIC ESTEREL stets mit Angabe eines Types t zu deklarieren.

```
input S (t)
```

Bei der Deklaration eines Ausgabesignales ...

```
output O (combine t with c)
```

... muss eine Funktion c mit angegeben werden,

Schnittstelle zwischen ESTEREL und C

Dieser kleine Exkurs soll zeigen, wie ESTEREL in die Hostsprache C eingebunden werden kann.

Wie bereits bekannt werden in ESTEREL Funktionen, Typen und Prozeduren nur deklariert. Deren Implementierung erfolgt dann in der gewählten Hostsprache. Folgendes Beispiels aus dem Systemhandbuch [6] des ESTEREL-Compilers Version 5.21 soll einen kleinen Einblick in die die Schnittstellendefinition mittels C geben.

C-Datentypen, welche im ESTEREL-Kernel Verwendung finden, sollten mittels **typedef** definiert werden.

```
typedef struct {
  int hours;
  int minutes;
  int seconds;
} TIME;
```

C-Code 2.3: Datentypdefinition in C [6]

Damit in ESTEREL Vergleiche mit dem Datentyp `TIME` möglich sind, muss in C ein entsprechender Vergleichsoperator definiert werden.

```
int _eq_TIME ( t1, t2)
  TIME t1, t2;   //  old-style syntax
{
  return (    t1.hours == t2.hours
           && t1.minutes == t2.minutes
           && t1.seconds == t2.seconds);
}
```

C-Code 2.4: Implementierung des Gleichheitsoperators [6]

Für das Konvertieren des Datentyp `TIME` in eine Zeichenkette und umgekehrt müssen in C die entsprechenden Konvertierungsfunktionen definiert werden.

```
void _text_to_TIME (time_ptr, str)
  TIME* time_ptr;
  char* str;
{
  sscanf( str, "%d:%d:%d",
          &(time_ptr->hours),
          &(time_ptr->minutes),
          &(time_ptr->seconds));
}
```

C-Code 2.5: Konvertierungsfunktion: TIME in Text[6]

```
char* _TIME_to_text (time)
  TIME time;
{
  static char buf[9]="";
  sprintf( buf, "%02d:%02d:%02d",
           time_ptr->hours,
           time_ptr->minutes,
           time_ptr->seconds);
  return (buf);
}
```

C-Code 2.6: Konvertierungsfunktion: Text nach TIME[6]

Die dazugehörigen Deklarationen in ESTEREL:

```
procedure _text_to_TIME (TIME, string) ();

function _TIME_to_text () (time) : TIME;
```

Esterel-Code 2.7: Deklarationen der Konvertierungsfunktionen[6]

Wir erinnern uns, dass bei Prozeduren die erste Argumentliste die Variablen enthält, die per Call-by-reference übergeben werden (Zeiger in C) und die zweite Liste die Variablen, die per Call-by-value übergeben werden. Folglich ist im obigen Beispiel bei der Deklaration der Prozedur die zweite Liste leer.

Für Interessierte sei darauf hingewiesen, dass in [6] der Schnittstelle zwischen ESTEREL und C ein ganzes Kapitel gewidmet ist.

welche zwei Ausgabewerte kombiniert.

$$c(v_1, c(v_2, \ldots c(v_{n-2}, c(v_{n-1}, v_n)) \ldots))$$

Diese individuelle Funktion mit Signatur der Form `function c(t,t):t` wird zur Kombination der verschiedenen Werte eines mehrfach gleichzeitig emittierten Signales eingesetzt.

`emit S(1)||emit S(2)||...||emit S(n)`

Denn zu einem Augenblick kann ein Ausgabesignal auch nur einmal gesendet werden; und dessen Wert muss eindeutig sein. Somit käme es ohne Angabe der Kombinationsfunktion c zur *Kollision* der emittierten Werte $v_1, v_2, \ldots, v_n$ (siehe „*sharing law*“ auf Seite 13). In PLAIN ESTEREL kann die Funktion c eigenverantwortlich weggelassen werden, sofern der Programmierer eine *Kollision* ausschließen kann.

Die strikte Trennung zwischen Eingabe- und Ausgabesignale lockert PLAIN ESTEREL indem es einen kombinierten Signaltyp einführt.

`inputoutput BUS_REQUEST`

Desweiteren kann in PLAIN ESTEREL die Angabe des Types des Signalwertes entfallen. Denn in der Praxis ist die Angabe eines zusätzlichen Parameters oft nicht notwendig. Aus Kompatibilitätsgründen zwischen den beiden ESTEREL-

Befehlssätzen wurde für die Konvertierung eines PLAIN-Signal in ein BASIC-Signal der primitive Datentyp `triv` in BASIC ESTEREL mit aufgenommen.

Neu in PLAIN ESTEREL ist der Schnittstellentyp Sensor für passive externe Geräte wie zum Beispiel Thermometer, welche selbst keine *Ticks* produzieren. Sensoren werden analog zu Eingabesignalen deklariert.

```
sensor TEMPERATURE (CELSIUS)
```

In ihrer Funktion sind Sensoren gegenüber Eingabesignalen stark eingeschränkt. Sie lassen sich lediglich mit dem Operator ? auslesen. Sensoren werden als permanent present angenommen [23] .

2.4.1 Lokale Signale

Signale lassen sich auch lokal deklarieren, zum Beispiel innerhalb von Schleifen. Der Gültigkeitsbereich lokaler Signale erstreckt sich nur auf den Bereich zwischen der Deklaration **signal** `S` (**type**) **in** und dem schließendem **end** (siehe Codeschnipsel 2.11 „Beispiel für 2 simultan abgegebene Signale ohne Kombinationsfunktion [7] “ auf Seite 10). Lokale Signale finden oft Einsatz in Verbindung mit nebenläufigen Codefragmenten welche ihrerseits Module einbinden. BOUSSINOT und DE SIMONE zeigen die Verwendung lokaler Signale anhand des Beispiels Maustreiber [8] .

2.4.2 Klassifikation von Signalen

In dem 2001 im „Embedded Systems Programming“-Magazine erschienenen Artikel „An introduction to Esterel“ [23] klassifiziert PALSHIKAR Signale nach den Kriterien Sichtbarkeit, Informationsgehalt und Zugriff:

Sichtbarkeit Schnittstellensignale versus lokale Signale

Informationsgehalt Wertelose Signale `S`, `?S` = `triv` versus wertbehaftete Signale `S(v)`, `?S` = `v`

Zugriff Nur-Eingabe Signale, Nur-Ausgabe Signale, Ein-und-Ausgabe Signale, Sensoren (Nur-Abfrage Signale)

2.4.3 Relationen

ESTEREL bietet mit *Relationen* die Möglichkeit durch Zusicherung die Kombinationen möglicher Eingabesignale zu reduzieren. Für die Verwendung von *Relationen* sind zwei Gründe anzuführen. Erstens kann der Ausschluss möglicher Signalkombinationen gewünscht sein und zweitens reduzieren solche Zusicherungen auch die Zustände in Automaten (siehe Paragraph „Automaten“ auf Seite 3).

ESTEREL unterscheidet zwei Arten von *Relationen*:

Inkompatibilitätsrealtionen Die Relation `LEFT_BUTTON # RIGHT_BUTTON` sichert zu, dass die zwei Signale `LEFT_BUTTON` und `RIGHT_BUTTON` nicht gleichzeitg auftreten.

Synchronitätsrelation Die Relation `SECOND => HUNDREDTH_OF_SECONDS` gibt an, dass wenn immer das Signal `SECOND` auftritt, auch dass Signal `HUNDREDTH_OF_SECONDS` zugegen ist.

2.5 Einfacher ESTEREL-Befehlssatz

Dieses Kapitel stellt den BASIC ESTEREL-Befehlssatz vor, auf welchen PLAIN ESTEREL aufsetzt.

2.5.1 Anweisungen und Ausdrücke

ESTEREL Anweisungen und Ausdrücke dienen der Manipulation von Signalen und Variablen. Innerhalb einer Anweisung werden alle Signaltypen gleich behandelt. Ausdrücke bestehen aus Konstanten, Variablen, Signalwerten, Operatoren und Funktionsaufrufen. Syntax der Bezeichner von Konstanten und Variablen sowie Funktionsnamen orientiert sich an der Hostsprache. Variablen, Signale und *Trap*-Marken müssen vor der ersten Verwendung deklariert sein. Operatoren sind die üblichen logischen und arithmetischen Operatoren sowie Vergleichsoperatoren. In Anweisungsblöcken wird dem Anweisungstrenner ; höhere Priorität zugeschrieben denn dem Parralleloperator ||. Zur Gruppierungen von Anweisungen steht das eckige Klammerpaar zur Verfügung.

BASIC ESTEREL Befehlssatz

```
% Dummy-Anweisung (NOP): Bewirkt nichts
nothing

% Halte-Anweisung: Erzwingt das Ende
% der Programmausführung
halt


% Zuweisung
X := exp

% Externer Routinenaufruf
call P (variable list) (expression list)

% Emission des Signals S mit dem Wert,
% den der Ausdruck exp liefert
emit S(exp)

% Anweisungssequenz
stat1 ; stat2


% Endlosschleife
loop
  stat
end

% Bedingte Verzweigung
if exp
  then stat1
  else stat2
end
```

```
% Test auf Vorhandensein des Signals S
% mit bedingter Ergebnisbehandlung
present S
  then stat1
  else stat2
end

% Watchdog: Führt stat solange
% wiederholend aus, bis das Signal S
% empfangen wird.
do stat watching S

% Parallelanweisung
stat1 || stat2

% Definition der Trap-Marke T
trap T in
  stat
end

% Austritt aus der Trap mit Marke T
exit T

% Lokale Variablendeklaration
var X : type in
  stat
end

% Lokale Signaldeklaration
signal S (combine type with comb) in
  stat
end
```

Die Liste der BASIC ESTEREL-Befehle ist in [7, S. 100, 3.2 Basic statements] zu finden. Metavariablen *type*, *exp* und *stat* stehen allgemein für Typen, Ausdrücke oder Anweisungen. *comb* steht für eine Kombinationsfunktion (siehe Abschnitt 2.4 „Schnittstellen“ auf Seite 6). Der Großteil der hier aufgeführten Anweisungen und Kontrollkonstrukte ist auch in anderen imperativen Sprachen wiederzufinden. ESTEREL-eigen sind **emit**, **present**, **watching** und **signal**; diese sind speziell auf die Signalbehandlung ausgelegt.

Variablen unterscheiden sich von den in Abschnitt 2.4 „Schnittstellen“ auf Seite 6 behandelten Signalen im Wesentlichen lediglich in puncto Kommunikation; wie bereits in Abschnitt 2.2 „Parallelität“ auf Seite 6 erwähnt können Variablen nicht gemeinsam genutzt werden.

```
1  VAR := 0;
2  [ VAR := VAR +1 || VAR := 1]
```

Esterel-Code 2.8: Unerlaubte Zuweisung [4]

Für den Fall, dass ein *Thread* P in $P_1 \parallel P_2$ geteilt wird bedeutet dies, dass auf eine zuvor deklarierte Variable solange aus P_1 und P_2 lesend zugegriffen werden darf, bis einer der beiden *Threads* P_1, P_2 auf die Variable schreibend zugreift. Ab diesen Moment wird dem anderen *Thread* der Zugriff komplett verwehrt [4] .

2.5.2 Imperative Anweisungen

Aktionen innerhalb einer Reaktion finden unverzüglich statt. Für Signalsequenzen bedeutet dies:

```
emit S1 || emit S2
⇔ emit S2; emit S1
⇔ emit S1; emit S2
```

Für Sequenzen von Zuweisungen wie zum Beispiel `X := 1; X := X + 1` hingegen spielt die Reihenfolge trotz augenblicklicher Ausführung eine Rolle. So ist `X` erwarteter Weise in dem Beispiel immer 2.

Auch alle Durchläufe einer Schleife sollten verzögerungsfrei erfolgen [a]. Diese Idee, dass auch Schleifen in einem einzigen Augenblick zu durchlaufen sind, wird durch Endlosschleifen, die wie im Codeschnipsel 2.9 „Endlosschleife [4] “ zu sehen unendlich viele Additionen und Speicherzugriffe durchführen, ad absurdum geführt.

[a] Siehe Abschnitt 1.3 „Synchronitätshypothese“ auf Seite 4: Klassifizierung der Anweisungen

```
loop
  X := X +1;
end
```

Esterel-Code 2.9: Endlosschleife [4]

Denn dies würde bedeuten, dass Endlosschleifen, welche schließlich nicht terminieren, nur einen Augenblick an Ausführungszeit beanspruchen. Widerspruch!
Daher stellt der ESTEREL-Compiler sicher, dass Schleifen nie zu dem Zeitpunkt terminieren, in welchem ihre Ausführung begonnen hat.

ESTEREL bietet zwei Konstrukte zur bedingten Verzweigung an: **if** für den Test auf boolesche Ausdrücke und **present** zum Testen auf die Präsenz von Signalen. Nun könnte argumentiert werden, dass ein **if-then-else**-Konstrukt in Verbindung mit einer Testfunktion **present** S völlig ausreichend gewesen wäre, sodass ein Vorhandensein eines Signales mit **if present** S **then**... abgefragt werden würde. Die Notwendigkeit zweier Konstrukte liegt in der unterschiedlichen Handhabe von Signalen und Variablen begründet. Signalzustand und Signalwert bleiben über die ganze Reaktion hinweg unverändert. Boolesche Variablen hingegen können beliebig oft ihren Wert ändern.
Ein extra Konstrukt für den Test auf das Abhandensein eines Signales ist nicht erforderlich. Solch ein Test ist mit **present** S **else** *stat* **end** durchführbar.

Das folgende Beispiel zeigt das Zusammenspiel von *Traps* mit dem Paralleloperator ‖ und der **exit**-Anweisung.

```
trap T1 in
  trap T2 in
      X := 0
    ||
      Y := 0; exit T2
    ||
      Z := 0; exit T1; Z := 1
  end;
  U := 0
end
```

Esterel-Code 2.10: Parallele Anweisungen und vorzeitiger Ausstieg [4]

Allen drei Variablen X, Y und Z wird der Wert 0 zugewiesen. Nach der Zuweisung an Y wird mit **exit** T2 die innere *Trap* T2 aufgelöst. Gleichzeitig geschied dies auch mit der *Trap* T1 durch die parallele Ausführung von **exit** T1. Damit wird die Zuweisung Z := 1 nie erreicht. Desgleichen kommt es durch das vorzeitige Verlassen von T1 auch nie zur Ausführung der Zuweisung U := 0.
Traps, welche entfernt vergleichbar sind mit *Exceptions* in JAVA, werden später noch genauer behandelt.

2.5.3 Temporale Anweisungen

Lokale Signale weisen gegenüber Ein- und Ausgabesignalen ein besonderes Verhalten auf: Sie konnen gleichzeitig in verschiedenen Sichtbarkeitsbereichen emittiert werden.

```
module FOO :
input S1 (integer);
output S2 (integer), S3 (Integer);
loop
  signal S (integer) in % local signal
    emit S(0);
    await S1;
    emit S(1);
  ||
    emit S2(?S);
    await S;
    emit S3(?S);
  end % end of scope of S
end
```

Esterel-
Code 2.11: Beispiel für 2 simultan abgegebene Signale ohne Kombinationsfunktion [7]

Wie in Codeschnipsel 2.11 „Beispiel für 2 simultan abgegebene Signale ohne Kombinationsfunktion [7] “ zu sehen, wird mit Beginn des ersten Schleifendurchlaufes S mit dem Wert 0 ausgesendet. Folglich gibt S2 den Wert 0 aus. Mit dem Auftreten des Eingabesignales S1 sendet S nun den Signalwert 1 und daraufhin S3 den Wert 1. Der Schleifenrumpf wurde das erste Mal durchlaufen und die Ausführung wird mit der erneuten Deklaration der lokalen Variable S fortgesetzt. Aber dies geht mit einer erneuten Initialisierung der lokalen Variable einher, sodass nun wieder $S(v) = \perp$ ist. Dies wird deutlich, wenn man die Schleife durch eine entsprechende Sequenz von **signal** S (**integer**) **in** ... ersetzt. So entfällt in diesem Fall auch die Notwendigkeit, mehrere emittierte Signalwerte von S mittels Kombinationsfunktion zusammenfassen zu müssen.

```
do
  do halt watching S1
  X := 0
watching S2
```

Esterel-Code 2.12: Beispiel für verschachtelte *Watchdogs* [7]

Ein weiterer wesentlicher Aspekt ist die Priorisierung der Signalbehandlung in verschachtelten *Watchdog*-Konstrukten. Der Codeschnipsel 2.12 „Beispiel für verschachtelte *Watchdogs* [7] “ führt zu folgender Fallunterscheidung:

i) `S1` erscheint vor `S2`: `X` wird auf 0 gesetzt und damit terminiert auch der äußere *Watchdog*.

ii) `S2` erscheint vor `S1`: Die Abbruchbedingung für den äußeren Watchdog ist erfüllt. Damit terminiert das gesamte Konstrukt.

iii) `S1` und `S2` treten gleichzeitig auf: Wie in *ii*) wird das ganze Konstrukt sofort verlassen und somit die Zuweisung `X := 0` nicht mehr Ausgeführt.

2.6 Erweiterter ESTEREL-Befehlssatz

PLAIN ESTEREL ist eine Erweiterung von BASIC ESTEREL. Neben den in Abschnitt 2.4 „Schnittstellen“ auf Seite 6 bereits behandelten beiden Schnittstellentypen `inputoutput` und `sensor` sind in PLAIN ESTEREL auch noch einige Befehle aus Gründen der Pragmatik sowie Wiederverwendung hinzugekommen.

Kopfgesteuerte Schleife PLAIN ESTEREL führt eine vorprüfende Schleife ein, welche *exp*-Mal die Anweisung *stat* ausführt.

```
1  repeat exp times
2    stat
3  end
```

Esterel-Code 2.13: Vorprüfende Schleife [7]

Dabei wird die Schleife auf eine `loop`-Schleife sowie einem umschließenden *Trap*-Konstrukt zurückgeführt, in welchem ein `if`-Konstrukt den Zählerstand prüft und gegebenenfalls mit `exit` den Schleifenrumpf verlässt.

Timeouts in Watchdogs Der Einsatz von *Timeouts* ist dann sinnvoll, wenn Schleifen unabhängig ihrer sonstigen Abbruchbedingungen eine bestimmte Dauer nicht überschreiten sollen. Oftmals ist zudem gewünscht, im Nachhinein Anweisungen in Abhängigkeit des Abbruchgrundes auszuführen. In BASIC ESTEREL ließe sich ein solches Verhalten mit dem folgenden Programmfragment herbeiführen:

```
1  trap TERMINATE in
2    do
3      stat1;
4      exit TERMINATE
5    watching S
6    stat2
7  end
```

Esterel-Code 2.14: *Timeout*-Alternative in BASIC ESTEREL [7]

In PLAIN ESTEREL lässt sich dies mit dem `timeout`-Befehl übersichtlicher schreiben.

```
1  do
2    stat1
3  watching S
4  timeout stat2
5  end
```

Esterel-Code 2.15: *Timeout* in PLAIN ESTEREL [7]

Immediate-Watchdog Die aus BASIC ESTEREL bekannte *Watchdog*-Anweisung hat den Nachteil, dass nur Signalauftritte zum Verlassen des Konstruktes führen, die erst emittiert wurden, als das Konstrukt bereits betreten wurde. Für den Fall, dass bei einem bereits anliegendem Signal der *Watchdog* direkt passiert wird, muss diesem in BASIC ESTEREL eine Abfrage auf das Vorhandensein des Signales vorangehen:

```
1  present S else
2    do
3      stat
4    watching S
5  end
```

Esterel-Code 2.16: *Immediate Watchdog* in BASIC ESTEREL [7]

Äquivalentes Programmfragment mit `immediate` in PLAIN ESTEREL:

```
1  do
2    stat
3  watching immediate S
```

Esterel-Code 2.17: *Immediate Watchdog* in PLAIN ESTEREL [7]

Warten Statt wie in BASIC ESTEREL mit

`do halt watching S`

das Warten auf das Eintreffen eines Signales zu formulieren genügt es in PLAIN ESTEREL ein `await S` anzugeben.

Darüber hinaus wird mit PLAIN ESTEREL eine Fallunterscheidung zum Warten auf mehrere Signale eingeführt.

```
await
  case SECOND do stat_1
  case 2 METER do stat_2
  case immediate ALARM do stat_3
end
```

Esterel-Code 2.18: Multiples **await** [7]

Dabei werden die Bedingungen der Reihe nach von oben nach unten geprüft. So wird beim gleichzeitigen Eintreten mehrerer Signale stets nur der am weitesten oben gelistete Fall behandelt.

Ausnahmebehandlung PLAIN ESTEREL führt einen Mechanismus zur Behandlung von Ausnahmen in Ergänzung zu den bereits aus BASIC ESTEREL bekannten *Traps* ein.

Im Codeschnipsel 2.19 „Beispiel für Ausnahmenbehandlung [7] “ sind den beiden *Traps* ALARM und ZERO_DIVIDE jeweils $stat_1$ respektive $stat_2$ zur Ausnahmebehandlung zugewiesen. *Trap* TERMINATE verbleibt ohne *exception handler*. Wird nun innerhalb von *stat* die Anweisung **exit** ALARM(*exp*) ausgeführt, so wird die Ausnahmebehandlung durch Ausführung von $stat_1$ eingeleitet. Auf den mitgegebenen Fehlerwert kann ähnlich wie bei Signalen über ??ALARM zugegriffen werden.

```
trap ALARM (combine integer with +),
     ZERO_DIVIDE, TERMINATE in
  stat
handle ALARM do stat_1
handle ZERO_DIVIDE do stat_2
end
```

Esterel-Code 2.19: Beispiel für Ausnahmenbehandlung [7]

Modularität und Wiederverwendung Mit der Direktive **copymodule** bietet PLAIN ESTEREL die Möglichkeit ESTEREL-Programme mittels Makros begrenzt modular zu gestallten.

copymodule M

kopiert den Inhalt des Modules M an die gewünschte Stelle. Die Direktive kann im Quellcode dort stehen, wo auch Anweisungen syntaktisch korrekt sind.

3 Semantiken

ESTEREL wurde strikt anhand formaler Methoden entwickelt. Denn gerade die Komplexität reaktiver Systeme und die hohen Anforderungen kritischer Systeme an die Korrektheit machen eine exakte Beschreibung des Systems notwendig [15] . So ist die Implementierung der ESTEREL-Sprachekonstrukte bloße Umsetzung der dahinterstehenden mathematischen Semantik. In diesem Abschnitt werden die in „The ESTEREL synchronous programming language: design, semantics, implementation“[7] zu findenden ESTEREL-Semantiken auszugweise vorgestellt. BERRY und GONTHIER geben hier zunächst eine intuitive Verhaltenssemantik und anschließend zwei mathematische Definitionen: Eine formale Verhaltenssemantik sowie eine Ausführungssemantik. Die naive Semantik beschreibt das Verhalten von ESTEREL im Wesentlichen über die Ausführungsdauer einzelner Anweisungen während die beiden detaillierten mathematischen Sematiken bei der Berechnung schrittweise vorgehen, indem sie ganze Anweisungen auf einzelne Reaktionen herunterbrechen. Dabei wird ein Programm P mit Eingabe I in ein neues Programm P' überführt, welches in der Lage ist, im Weiteren die zuvor erzeugte Ausgabe O zu verarbeiten.

Für eine denotationelle Semantik, welche sich aus der naiven Semantik wie in Abschnitt 3.1 „Intuitive Verhaltenssemantik“ auf Seite 13 beschrieben ableitet, sei auf die Arbeit „Sémantiques et modèles d’exécution des langages réactifs synchrones; application à Esterel“ von GONTHIER verwiesen. Eine statische Semantik sowie die formale Verhaltenssemantik und die mathematische Ausführungssemantik werden von BERRY und COSSERAT in „The ESTEREL Synchronous Programming Language and its Mathematical Semantics“ [5] ausführlich behandelt. Ferner sei noch auf das Papier „The Constructive Semantics of Pure Esterel“ [3] verwiesen, welches neben einer logischen Semantik

Das *sharing law*

Die zentrale Rolle im Verhalten von ESTEREL-Programmen spielt das *sharing law* [7, S. 101] , dessen folgende zwei Eigenschaften für den Determinismus im Programmablauf maßgeblich sind:

1. Jedes Signal verfügt über einen Zustandsmarker welcher angibt, ob ein Signal an einer Reaktion beteiligt ist. Solange die Reaktion anhält, ist der Zustand eines Signals unveränderlich (*persistent law*). Ein Signal ist nur dann Teil einer Reaktion, sofern es
   a) entweder ein im Eingabeereignis vorhandenes Eingabesignal ist
   b) oder ein vom Programm erzeugtes Ausgabesignal oder lokales Signal ist.
2. Unabhängig vom Wert des Zustandsmarkers gilt: In jeder Reaktion ist der Wert eines Signales `?S` eindeutig. Der Wert des Signales ist
   a) der aktuelle Eingabewert, wenn das Signal ein an der Reaktion beteiligtes Eingabesignal ist.
   b) eine Kombination aller emittierten Werte, wenn das Signal ein lokales Signal oder ein Ausgabesignal ist.
   c) der Wert aus der letzten Reaktion, sofern das Signal nicht an der aktuellen Reaktion beteiligt ist.
   d) der unbestimmte Initialwert $\perp$, sofern es ein nicht an der Reaktion beteiligtes lokales Signal oder Ausgabesignal ist, welches bislang noch nie emittiert wurde.

noch eine konstruktive sowie elektrische ESTEREL-Semantik vorstellt.

3.1 Intuitive Verhaltenssemantik

Die intuitive Semantik beschreibt oberflächlich das Verhalten eines ESTEREL-Modules auf Eingabeereignisse unter Berücksichtigung deren zeitlichen Ablaufs (*timing*). Besonderes Augenmerk wird dabei auf den Moment der Ausführung einer Anweisung gelegt.

Ein ESTEREL-Programm entspricht einer Folge von Reaktionen. Eine *Reaktion* ist die unverzügliche Antwort auf das Auftreten eines Eingabesignals. Während einer Reaktion aktualisiert das Module Variablen und löst Ausgabesignale aus. Die emittierten Signalwerte ein und des selben Signals werden mittels Kombinationsfunktion zusammengefasst. Alle aus einer Reaktion stammenden Ausgabesignale werden unter dem Begriff Ausgabeereignis (*output event*) subsumiert. Dieses induzierte Ausgabeereignis wirkt in die Umwelt des Moduls. Eine Reaktion kann nur auf ein Eingabeereignis erfolgen und gemäß der Synchronitätshypothese (siehe Abschnitt 1.3 „Synchronitätshypothese“ auf Seite 4) dies augenblicklich und atomar. Die hieraus resultierenden Forderungen beschreibt das *sharing law*. Die zeitliche Abfolge der auftretenden Eingabeereignisse wird als Eingabehistorie (*input history*) bezeichnet.

Die strukturelle Beschreibung der intuitiven Verhaltenssemantik fußt auf der folgenden Festlegung des allgemeinen Ausführungsverhalten von Anweisungen:

- Der Kontext einer Anweisung bestimmt deren Ausführungsmoment.
- Der Zeitpunkt zu welchem die Ausführung einer Anweisung terminiert bestimmt allein die jeweilige Anweisung.
- Unabhänging davon können Blöcke mit **`exit`** auch vorzeitig verlassen werden.
- Die Ausführung einer jeden Anweisung kann aus jedem Zweig eines ESTEREL-Programmes mit sofortiger Wirkung abgebrochen werden.

Eine ausführliche strukturelle Beschreibung der einzelnen Befehle des im Abschnitt 2.5 „Einfacher ESTEREL-Befehlssatz“ auf Seite 8 beschriebenen BASIC ESTEREL-Befehlssatzes geben BERRY und GONTHIER in „The ESTEREL synchronous programming language: design, semantics, implementation“ [7, S. 102] .

3.2 Formale Verhaltenssemantik

Neben der in Abschnitt 3.1 „Intuitive Verhaltenssemantik“ vorgestellten naiven Verhaltenssemantik geben BERRY und GONTHIER [7] auch eine mathematische Definition der Verhaltenssemantik von BASIC ESTEREL. Diese *natural deduction semantic* basiert auf den strukturellen Regeln [24] aus PLOTKIN's Kalkül des natürlichen Schließens und weist einen stärkeren Praxisbezug auf.
Kurz gesagt übersetzt die auf das Ein- und Ausgabeverhalten fokusierte Semantik ein gegebenes Programm in Abhängigkeit eines Eingabeereignisses in ein neues Programm, auf welches dann die nachfolgenden Eingabeereignisse angewandt werden. Für den Entwurf eines ESTEREL-Interpreters eignet sich die Verhaltenssemantik allerdings nicht. Zu diesen Zweck haben BERRY und GONTHIER die weit umfangreichere Ausführungssemantik entworfen, die in Abschnitt 3.4 „Ausführungssemantik“ auf Seite 17 vorgestellt wird.

3.2.1 Ereignisse und Historien

Ein Ereignis E ist formal definiert als eine Menge gleichzeitig emittierter Signale S_i und deren zugeordneter Signalwerte v_i.

$$E = S_1(v_1) \cdot S_2(v_2) \cdot \ldots \cdot S_n(v_n), \qquad n \geq 0$$

Ein Vorkommen des Signals S im Ereignis E wird allgemein mit $S \in E$ beziehungsweise unter Berücksichtigung des Signalwertes mit $S(v) \in E$ oder $E(S) = v$ notiert. Die Abwesenheit eines Signals S in einem Ereignis E wird mit $S \notin E$ angegeben. $E = \epsilon$ ist die Schreibweise für ein Ereignis E, welchem keine Signale zugeordnet sind.

Die aus Abschnitt 2.4 „Schnittstellen“ auf Seite 6 bekannte Kombinationsfunktion $c(v_x, v_y)$ wird formal $x \star_s y$ geschrieben. Hinzu kommt eine Kombinationsoperation für Ereignisse. $E = E_1 \star E_2$ ist das sychrone Produkt zweier Ereignisse E_1 und E_2 wobei gilt:

$$\begin{aligned} S(v_1) \in E_1 \wedge S \notin E_2 &\Rightarrow S(v_1) \in E \\ S(v_2) \in E_2 \wedge S \notin E_1 &\Rightarrow S(v_2) \in E \\ S(v_1) \in E_1 \wedge S \in E_2 &\Rightarrow S(v_1 \star_s v_2) \in E \\ S(v_1) \notin E_1 \wedge S \notin E_2 &\Rightarrow S \notin E \end{aligned}$$

Sei $\mathscr{S}$ eine Sortierung aller emittierter $S^+(v)$ oder inaktiver $S^-(v)$ mit Werten v behafteter Signale gemäß dem *persistence law* (siehe „*sharing law*“ auf Seite 13). Während *Ereignisse* E lediglich emittierte Signale $S(v)$ mit Momentanwert v beinhalten, ist ein *vollständiges Ereignis* $\hat{E}$ die Vereinigung aller emittierten Signale $S^+(v) \in \mathscr{S}$ mit der Menge aller nicht emittierten Signale $S^-(v) \in \mathscr{S}$.

Eine *Historie* H ist eine Folge von Ereignissen.

$$H = E_0, E_1, \ldots, E_i, \ldots$$

Dabei bezeichnet $H[n]$ eine abgeschlossene *Historie* $E_0, E_1, \ldots, E_n$. Eine *vollständige Historie* $\hat{H}$ ist eine Sequenz vollständiger Ereignisse.

$$\hat{H} = \hat{E}_0, \hat{E}_1, \ldots, \hat{E}_n$$

Gibt es in dem ersten Ereignis $\hat{E}_0$ einer vollständigen Historie ein nicht aktives Signal S so ist dessen Wert unbestimmt.

$$S^- \in \hat{E}_0 \quad \Rightarrow \quad \hat{E}_0(S) = \perp$$

Für jedes folgende Ereignis $\hat{E}_i$ in dieser Historie ist der Wert eines Signales, welches zum Zeitpunkt i nicht Teil einer Reaktion ist, gleich dem zuletzt in einer früheren Reaktion zugewiesenen Signalwert.

$$S^-(v) \in \hat{E}_i \quad \Rightarrow \quad \hat{E}_{i-1}(S) = v$$

Für eine Sortierung $\mathscr{S} = \{S1, S2\}$ ist eine Mögliche Historie H:

	E_0	E_1	E_2	E_3
H	$S1(0)$	$S2(1)$	ϵ	$S1(2) \cdot S2(2)$

Die dazugehörige vollständige Historie $\hat{H}$ ist dann:

	$\hat{E}_0$	$\hat{E}_1$
$\hat{H}$	$S1^+(0) \cdot S2^-(\perp)$	$S1^-(0) \cdot S2^+(1)$

	$\hat{E}_2$	$\hat{E}_3$
$\hat{H}$	$S1^-(0) \cdot S2^-(1)$	$S1^+(2) \cdot S2^+(2)$

3.2.2 Modulableitung

Anhand einer gegebenen Eingabehistorie I berechnet ein Programm P eine Ausgabehistorie O.

$$P = P_0 \overset{O_0}{\underset{\hat{I}_0}{\longmapsto}} P_1 \overset{O_1}{\underset{\hat{I}_1}{\longmapsto}} \ldots \overset{O_{i-1}}{\underset{\hat{I}_{i-1}}{\longmapsto}} P_i \overset{O_i}{\underset{\hat{I}_i}{\longmapsto}} \ldots$$

Die Ausgabehistorie wird dabei schrittweise durch Überführung eines Programmes P_i in ein neues abgeleitetes Programm P_{i+1} hergeleitet. P_{i+1} wird Ableitung von P_i bezüglich der vollständigen Eingabehistorie $\hat{I}$ genannt und unterscheidet sich von P_i in Hinsicht auf den Modulkörper.

Zur Verdeutlichung ein kleines Beispiel das einen 100-Meter-Lauf starten soll:

```
1  module COUNTDOWN :
2    input DEC;
3    output GO;
4    await 3 DEC do emit GO; halt; end;
5  end
```

Esterel-Code 3.1: Initiales Programm P_0

Mit dem Auftreten des Signals DEC wird Programm P_0 in P_1 überführt. Dabei ist das vollständige Eingabeereignis $\hat{I}_0 = \text{DEC}^+$

```
1  module COUNTDOWN :
2    input DEC;
3    output GO;
4    await 2 DEC do emit GO; halt; end;
5  end
```

Esterel-Code 3.2: Programm im Ableitungsschritt P_1

Für die zwei nächsten Vorkommen des Signals DEC wird das Programm P_1 weiterüberführt in P_3 und die Anweisung

```
await 2 DEC do emit GO; halt; end;
```

wird reduziert zu

```
halt
```

wobei das Ausgabeereignis $O_2 = \text{GO}$. Das abgeleitete Programm P_3 akzeptiert zwar noch Eingaben, reagiert aber nicht mehr.

3.2.3 Induktionsregeln

Die zur Programmentwicklung entlang der Eingabehistorie verwendete Relation $\longmapsto$ ist eine vereinfachte Form der allgemeineren Relation $\longrightarrow$ für Ableitungen auf Anweisungsebene.

$$\langle stat, \rho \rangle \xrightarrow[\hat{E}]{E', b, T} \langle stat', \rho' \rangle$$

Die Relation $\longrightarrow$ stellt die Überführung des Befehls *stat* mit aktuellem Speicher ρ in *stat'* mit modifizierten Speicher ρ' dar. Ist der Speicher leer, d.h. enthält keine Variablen, so wird dieser anstelle von ρ mit ϕ gekennzeichnet. Das vollständige Ereignis $\hat{E}$ beinhaltet die gesamte Signalumgebung, unter welcher *stat* ausgeführt wurde. Das Ausgabeereignis E' ist eine Ansammlung aller während der Ausführung von *stat* emittierten Signale.

$$b = \begin{cases} tt, & \textit{stat} \text{ terminiert} \\ ff, & \text{sonst} \end{cases}$$

Die Komponente T speichert alle *Trap*-Marken der *Traps*, welche mit **exit** während der Ausführung von *stat* verlassen wurden.

Für die Programmüberführung eines Programmes P in ein abgeleitetes Programm P wird in ESTEREL aus technischen Gründen angenommen, dass das Programm niemals terminiert.

$$P \overset{E'}{\underset{\hat{E}}{\longmapsto}} P' \quad \Leftrightarrow \quad \langle stat, \phi \rangle \xrightarrow[\hat{E}]{E', ff, \emptyset} \langle stat', \phi \rangle$$

Hierbei entspricht *stat'* dem Körper des Programmes P'.

Die Auswertung von Ausdrücken *exp* liefert einen Wert v und hat die Form

$$\langle exp, \rho \rangle \underset{\hat{E}}{\longrightarrow} \langle v \rangle$$

Nachfolgend seien auszugsweise einige Induktionsregeln aus [7, S. 118 ff] für BASIC ESTEREL Anweisungen gegeben.

nothing-Axiom

$$\langle \text{nothing}, \rho \rangle \xrightarrow[\hat{E}]{\epsilon, tt, \emptyset} \langle \text{nothing}, \rho \rangle$$

Die Ausführung von **nothing** terminiert ohne dabei Speicher ρ und Programmfragment *stat* zu verändern. Zudem werden keinerlei Signale emittiert.

`halt`-Axiom

$$\langle \text{halt}, \rho \rangle \xrightarrow[\hat{E}]{\epsilon, ff, \emptyset} \langle \text{halt}, \rho \rangle$$

`halt` unterscheidet sich von `nothing` lediglich darin, dass es mit $b = ff$ nicht terminiert.

`emit`-Regel

$$\frac{\langle exp, \rho \rangle \xrightarrow[\hat{E}]{} v}{\langle \text{emit S}(exp), \rho \rangle \xrightarrow[\hat{E}]{S(v), tt, \emptyset} \langle \text{nothing}, \rho \rangle}$$

`emit` emittiert den Wert v des Signals S in die Umwelt ($E' = S(v)$) und terminiert ($b = tt$). Folglich ist $stat' =$ `nothing`.

`present`-Regel

Ist das Signal `S` in der Reaktion vorhanden, so wählt `present` den `then`-Zweig und führt $stat_1$ aus:

$$\frac{S^+ \in \hat{E} \qquad \langle stat_1, \rho \rangle \xrightarrow[\hat{E}]{E_1', b_1, T_1} \langle stat_1', \rho_1' \rangle}{\langle \text{present S then } stat_1 \text{ else } stat_2, \rho \rangle \xrightarrow[\hat{E}]{E_1', b_1, T_1} \langle stat_1', \rho_1' \rangle}$$

Ist das Signal `S` nicht Teil der Reaktion, so wählt `present` den `else`-Zweig und führt $stat_2$ aus:

$$\frac{S^- \in \hat{E} \qquad \langle stat_2, \rho \rangle \xrightarrow[\hat{E}]{E_2', b_2, T_2} \langle stat_2', \rho_2' \rangle}{\langle \text{present S then } stat_1 \text{ else } stat_2, \rho \rangle \xrightarrow[\hat{E}]{E_2', b_2, T_2} \langle stat_2', \rho_2' \rangle}$$

3.3 Determinismus der Verhaltenssemantik

Wie bereits zu Anfang erwähnt war es Ziel der Entwickler, mit ESTEREL eine deterministische Programmiersprache für reaktive Systeme zu entwickeln.

In „The Constructive Semantics of Pure Esterel“ [3] nennt BERRY eine Definition für *logisch korrekte* Programme. Demnach ist ein Programm bezüglich einer Eingabe *logisch reaktiv*, wenn es zu dieser Eingabe mindestens eine Ausgabe produziert; liefert es höchstens eine Ausgabe, so ist es *logisch deterministisch*. Ist ein Programm sowohl *logisch reaktiv* als auch *logisch deterministisch*, so ist es *logisch korrekt*.

In diesem Abschnitt wird betrachtet, ob die Verhaltenssemantik dem Anspruch an Determinismus genügt. Hierzu seien zwei kleine Programme aus [7] angeführt, welche von ihrer Umwelt isoliert betrachtet werden.

Mehrfache Semantik

```
signal S in
  present S then emit S end
end
```

Esterel-Code 3.3: Programm P_1 [7]

Die logische Kohärenzregel (*logical coherence law*) [3, S. 27] besagt: Ein Signal `S` ist zu einer Zeit genau dann vorhanden, wenn es im Sichtbarkeitsbereich ein `emit S` gibt, welches `S` zu genau dieser Zeit sendet.

Demzufolge ist es in diesem Fall gleich, ob das Signal `S` im Rumpf des `signal`-Konstruktes als vorhanden oder abwesend angenommen wird, die Ableitung des Programmes führt in beiden Fällen zu

```
signal S in
  nothing
end
```

Esterel-Code 3.4: Abgeleitetes Programm P_1'

Für das Programm P gibt es also 2 Semantiken.

Ohne Semantik

```
signal S in
  present S else emit S end
end
```

Esterel-Code 3.5: Programm P_2 [7]

Das Programm P_2 hingegen hat keine Verhaltenssemantik. Denn angenommen, S ist innerhalb der lokalen Signalumgebung present. Dann muss es für diesen Fall ein erreichbares `emit S` geben. Widerspruch! Angenommen, S ist innerhalb der lokalen Signalumgebung nicht present. Dann dürfte keine `emit`-Anweisung zur Ausführung kommen. Aber in diesem Fall wird der `else`-Zweig durchlaufen. Widerspruch! Das Signal S würde also genau dann vorhanden sein, wenn es nicht gesendet wird.

Offensichtlich ist das *sharing law* („*sharing law*“ auf Seite 13) nicht hinreichend, um Determinismus zu garantieren.

3.4 Ausführungssemantik

Im Gegensatz zur Verhaltenssemantik garantiert die die Ausführungssemantik die Korrektheit der einzelnen Reaktionen. Die Semantik ist für die Ausführung von ESTEREL-Programmen auf einem gewöhnlichen sequentiell arbeitenden Rechner konzipiert. Jede Reaktion (*macrostep*) wird als Sequenz atomarer Aktionen (*microsteps*) aufgefasst, welche mit einer Vorbereitung (*expansion step*) auf die nächste Reaktion abgeschlossen wird. Ein Programm wird bezüglich einer Eingabe als korrekt bezeichnet, wenn es eine Ausführung gibt, in der es anhält. Reaktionen korrekter Programme sind daher deterministisch; auch dann, wenn die Ausführung des gesamten Programmes aufgrund der Verzahnung parallel ausgeführter Anweisungssequenzen nichtdeterministisch ist.

3.4.1 Signale

Die Implementierung der Signale in ESTEREL erfolgt in Form zugriffsgesteuerter geteilter Variablen (*controlled shared variables*). Der kontrollierte Zugriff auf die Speicherstellen der Signale erfolgt über zusätzlich mitgespeicherte Statusinformationen. Jedes Speicherzelle eines Signals wird mit einem Wert aus der Menge der Statusinformationen $\{\bot, \dagger, +, -\}$ markiert, welcher den aktuellen Zustand der Zelle angibt.

Dabei bedeutet:

- $S^{\bot} = v$: Die Zelle ist in der aktuellen Reaktion bislang unberührt geblieben. Der Speicherinhalt ist v und hat sich seit der letzten Reaktion nicht verändert.
- $S^{\dagger} = v$: Die Zelle ist bereits einmal in der aktuellen Reaktion geschrieben worden. Der momentane Speicherinhalt ist v. Da noch weitere Schreibaktionen in dieser Reaktion folgen können, kann das Signal S in dieser Reaktion nicht mehr gelesen werden.
- $S^{+} = v$: Der Zellinhalt wurde in dieser Reaktion bereits manipuliert. Die Schreiboperationen sind jedoch abgeschlossen. Dass heißt, da keine weiteren Schreibaktionen mehr zugelassen sind, kann auf den Zellinhalt mit $?S$ wieder lesend zugegriffen werden.
- $S^{-} = v$: Der Zellinhalt wurde in dieser Reaktion bislang unberührt belassen und die Möglichkeit in die Zelle zu schreiben besteht für den Rest der Reaktion nicht weiter. Der Zellinhalt des abwesenden Signals S kann mit $?S$ gelesen werden. Und dies ist der Signalwert aus der vorherigen Reaktion.

Der Signalspeicher θ leitet sich aus dem leeren Speicher ϕ ab und verfügt neben Speicherplatz für den Signalwert auch über ein Feld für den Statusindikator $\theta.(S^x = v), x \in \{\bot, \dagger, +, -\}$. Die Leseoperation lässt sich auf die Speichezelle $\theta.(S^x = v)$ jedoch nur dann anwenden, sofern $x \in \{+, -\}$. Das Schreiben in eine Signalzelle wird mit $\theta.[S \leftarrow v]$ notiert. Geschrieben werden kann nur in Zellen $\theta.(S^x = v)$ mit $x \in \{\bot, \dagger\}$.

Die Ausführungssemantik wird mit der Verhaltenssemantik verknüpft, indem die soeben behandelten Signalspeicher mit den Eingabe- und Ausgabehistorien aus der Verhaltenssemantik in Beziehung gesetzt werden.

3.4.2 Atomare Aktionen

Die Ausführungssemantik besteht aus einer Menge von Aktionsregeln. Diesen stehen noch zusätzliche Hilfsregel zur Verfügung: Regeln zur Ausdrucksauswertung, Terminationsregeln und Potentialregeln.

Aktionsregeln Die Aktionsregeln orientieren sich im Aufbau an den Induktionsregeln der Verhaltensemantik, führen jedoch noch als weitere Komponente den Speicher mit auf.

$$\langle stat, \rho, \theta\rangle \to \langle stat', \rho', \theta'\rangle$$

Als Beispiele für Aktionsregeln folgen die Regel für das Aussenden von Signalen sowie die Regel zur Abfrage der Existenz eines Signales.

`emit`-Regel

$$\frac{\langle exp, \rho, \theta\rangle \to v}{\langle \text{emit S}(exp), \rho, \theta\rangle \to \langle \text{nothing}, \rho, \theta[S \leftarrow v]\rangle}$$

Das Aussenden eines Signales kann erfolgen, sofern der Ausdruck exp berechenbar ist.

`present`-Regel

$$\frac{S^+ \in \theta}{\langle \text{present S then } stat_1 \text{ else } stat_2, \rho, \theta\rangle \to \langle stat_1, \rho, \theta\rangle}$$

$$\frac{S^- \in \theta}{\langle \text{present S then } stat_1 \text{ else } stat_2, \rho\rangle \to \langle stat_2, \rho, \theta\rangle}$$

Eine `present`-Anweisung wählt in Abhängigkeit der Statusinformation eines Signals S den entsprechenden Zweig. Hier wird offensichtlich, dass `present` nicht für Ausgabesignale definiert ist.

Ausdrucksauswertung Regeln zur Ausdrucksauswertung handhaben Regeln der Form

$$\langle exp, \rho, \theta\rangle \to v$$

Die Regeln für den Signalspeicher erzwingen, dass der Zugriff mittels `?S` auf den Signalspeicher θ nur dann zur Auswertung des Signalwertes v führt, wenn für den Signalstatus gilt: $\theta.(S^x = v),\ x \in \{+, -\}$.

$$\frac{S^- \in \theta \quad \vee \quad S^+ \in \theta}{\langle ?S, \rho, \theta\rangle \to \theta(S)}$$

Terminationsregeln Terminationsregeln berechnen eine partielle Funktion $\mathscr{T}(stat) = \langle b, T\rangle$ wobei der Rückgabewert b angibt, ob $stat$ terminiert, und der Rückgabewert T alle angesprungenen Trap-Marken enthält. Die Funktion $\mathscr{T}$ ist erst dann ausführbar, wenn sonst keine weitere Ausführung mehr möglich ist.

`nothing`-Regel

$$\mathscr{T}(nothing) = \langle tt, \emptyset\rangle$$

Die Anweisung `nothing` terminiert. Und wegen $b = tt$ ist auch $T = \emptyset$.

`halt`-Regel

$$\mathscr{T}(nothing) = \langle ff, \emptyset\rangle$$

Die Anweisung `halt` terminiert nicht ($b = ff$) und wird auch nicht verlassen ($T = \emptyset$).

Potentialregeln Das Potential $\pi(stat)$ einer Anweisung $stat$ ist die Menge aller möglichen Signalausgaben innerhalb eines Berechnungsschrittes. Potentialregel werden zur Manipulation der Signalstatus sowie der Signalspeicher eingesetzt.

4 Fazit

Im Verlauf dieser Seminararbeit wurden Umfeld, Grundlagen, Syntax und Semantiken der synchronen, imperativen und nebenläufigen Programmiersprache ESTEREL beleuchtet. Es wurde die für synchrone Sprachen wesentliche Synchronitätshypothese behandelt sowie im Rahmen der intuitiven und formalen Semantiken Regeln wie das *sharing law*, *persistence law* und *logical coherence law* erwähnt. Ferner wurde überprüft, ob mit ESTEREL gemäß der Zielsetzung eine synchrone Sprache entwickelt wurde, die auch deterministisch ist. Die Betrachtungen im Einzelnen erfolgten mittels der Befehlssätze BASIC ESTEREL und PLAIN ESTEREL.

Die synchrone Sprache ESTEREL scheint eher ein Nischendasein mit rein industriellem Einsatz zu plegen. Dennoch ließen sich im Laufe der Recherche genügend Quellen mit ergänzendem und interessantem Material finden; mit zumeist rein theoretischem Charakter aber dennoch praktischer Relevanz. Auf eine Vielzahl an ergänzender Literatur verwiesen wir in dieser Seminararbeit an entsprechender Stelle.

Mit seinen Eigenschaften ist ESTEREL auf ein sehr spezifisches Einsatzgebiet wie Luftfahrt, Schiene und Kernenergiesektor ausgerichtet, sodass es unserer Ansicht nach bei der eher marginalen Verbreitung bleiben wird.

Literatur

[1] C. André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. Technical report, Laboratoire Informatique, Signaux, Universite de Nice-Sophia Antipolis, 1996.

[2] J. G. P. Barnes. The standardization of RTL/2. *Software: Practice and Experience*, 10:707–719, 1980.

[3] G. Berry. The Constructive Semantics of Pure Esterel Draft Version 3. 1999.

[4] G. Berry. The Esterel v5 Language Primer. Technical report, 2000.

[5] G. Berry and L. Cosserat. *The ESTEREL Synchronous Programming Language and its Mathematical Semantics*. Institut national de recherche en informatique et en automatique, 1984.

[6] G. Berry and Esterel Team. The Esterel v5 21 System Manual. 1999.

[7] G. Berry and G. Gonthier. The ESTEREL synchronous programming language : design , semantics , implementation. *Science of Computer Programming*, 19:87–152, 1992.

[8] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[9] M. W. S. Dayaratne, P. S. Roop, and Z. Salcic. Direct Execution of Esterel Using Reactive Microprocessors. pages 1–22, 2005.

[10] S. A. Edwards, C. Soviani, and J. Zeng. CEC: The Columbia Esterel Compiler, 2006.

[11] V. K. Garg. Modeling of Distributed Systems by Concurrent Regular Expressions.

[12] T. Gautier, P. le Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. Technical report, IRISA / INRIA, Campus de Beaulieu„ 1987.

[13] U. Goltz. Prozeßalgebren. Technical Report November, Technische Universität Braunschweig, Institut für Programmierung und Reaktive Systeme, 2007.

[14] U. Goltz. Vorlesung Reaktive Systeme II. Technical report, Technische Universität Braunschweig, Institut für Programmierung und Reaktive Systeme, 2008.

[15] U. Goltz, D. Maciuszek, and W. Struckmann. Vorlesungsskript: Reaktive Systeme. Technical report, Technische Universität Carolo-Wilhelmina zu Braunschweig, 2002.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. Technical Report 9, 1991.

[17] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477 – 498. Springer-Verlag New York, 1989.

[18] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[19] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*, volume 32. Addison Wesley, 2000.

[20] M. Jantzen, M. Kudlek, and G. Zetzsche. Concurrent Finite Automata. 2009.

[21] F. Maraninchi. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1-3):61–92, 2001.

[22] D. May. CSP, occam and Transputers. In *Communicating Sequential Processes*, pages 75–84. Springer, 2005.

[23] G. K. Palshikar. An introduction to Esterel. *Embedded Systems Programming*, (November), 2001.

[24] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, University of Aarhus, Denmark, 1981.

[25] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin. The Synchronous Hypothesis and Synchronous Languages. pages 1–21, 2004.

[26] S. Sergio, S. Terrasa, V. Lorente, and A. Crespo. Implementing Reactive Systems with UML State Machines and Ada 2005. *Lecture Notes in Computer Science*, pages 149–163, 2009.

[27] C. Siemers. *Handbuch Embedded Systems Engineering*. TU Clausthal, FH Nordhausen, 2011.

[28] E. Technologies. The Esterel v7 Reference Manual Version v7 30. Technical Report November, Esterel Technologies, 2005.

[29] N. Wirth. The Programming Language Pascal. Technical report, ETH Zürich, 1973.

[30] N. Wirth. Modula-2 and Oberon. Technical Report June 2005, 2006.

Software

Esterel Web Webpräsenz des „Syncronous Reactive Team“ des Forschungsinstituts INRIA und der Hochschule Ecole des Mines de Paris. Hier wird unter anderem der ESTEREL-Compiler *v*5.92 für die Betriebsysteme LINUX, SUN und WINDOWS sowie eine umfangreiche Dokumentation angeboten. Darüber hinaus gibt es etwas über die Entwicklungsgeschichte von ESTEREL zu erfahren.
http://www-sop.inria.fr/esterel.org/filesv5_92/

The Columbia Esterel Compiler (CEC) Ein quelloffener Hard- und Softwarecompiler. Unterstützt eine Untermenge des ESTEREL V5 Befehlssatzes.
http://www.cs.columbia.edu/~sedwards/cec/

KIES - Kieler Esterel Ein Projekt der Christian-Albrechts-Universität zu Kiel zur Transformation von ESTEREL-Programmen in SYNCCHARTS. Hierzu gehört auch ein auf XTEXT basierender Editor für Eclipse. KIES ist in der Weiterentwicklung KIELER — Kiel Integrated Environment for Layout Eclipse RichClient aufgegangen.
http://trac.rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/Esterel
http://www.informatik.uni-kiel.de/rtsys/kieler/

Esterel Technologies ESTEREL TECHNOLOGIES bietet mit dem Produkt SCADE eine umfangreiche kommerzielle Entwicklungsumgebung für den professionellen Einsatz in der Industrie; insbesondere Luftfahrtbranche, Schiene sowie Kernenergiesektor.
http://www.esterel-technologies.com/

Bildquellen

Logo auf der Titelseite: Esterel Web, http://www-sop.inria.fr/esterel.org/filesv5_92/home.htm, retuschiert

A Beispiel

Als Beispiel eine einfache Armbanduhr aus [6, S. 103] , welche die Zeit in Stunden, Minuten und Sekunden anzeigt. Um die Uhr einstellen zu können, verfügt sie über zwei Taster zum Setzen der Stunden und der Minuten. Der Einfachheit halber wird das gleichzeitige Drücken der Taster SET_HOUR und SET_MINUTE sowie das Zusammenfallen mit dem Signal SECOND des externen periodischen Taktgebers durch das Setzen in Relation (Z. 26) ausgeschlossen. Die Zeit ist als benutzerdefinierter Datentyp TIME deklariert (Z. 3).
Zu Beginn der Ausführung (Z. 29) wird eine lokale Variable TIME mit dem Initialwert INITIAL_TIME definiert. Daraufhin wird die Zeit ein erstes Mal über das Ausgabesignal TIME (Z. 23) ausgegeben (Z. 32). In der folgenden Schleife werden die Eingabesignale SECOND, SET_MINUTE und SET_HOUR über das multiple **await**-Konstrukt (Z. 34) abgepasst und entsprechend behandelt. Bei jedem Auftreten des Taktes SECOND wird die Inkrementprozedur (Z 36) aufgerufen. Der Prozedur INCREMENT (Z. 12) wird die zu manipulierende Zeit als Variablenparameter und die Konstante ONE_SECOND (Z. 6) als Werteparameter übergeben. Wird dagegen das Signal SET_MINUTE (Z. 37) oder SET_HOUR (Z. 40) registriert, so ist der Werteparameter der Inkrementroutine entsprechend ONE_MINUTE (Z. 7) oder ONE_HOUR (Z. 8). Im Fall des Minutensetzens werden die Sekunden zuvor noch zurückgesetzt (Z. 38). Die aktuelle Zeit TIME wird fortwährend über das wertbehaftete Signales TIME ausgegeben (Z. 45).

```
1  module WATCH:
2
3    type          TIME;
4
5    constant      INITIAL_TIME : TIME;
6    constant      ONE_SECOND,
7                  ONE_MINUTE,
8                  ONE_HOUR : TIME;
9
10   % The INCREMENT procedure is used for time arithmetics
11   % The RESET_SECONDS procedure sets seconds to zero.
12   procedure     INCREMENT (TIME) (TIME),
13                 RESET_SECONDS (TIME) ();
14
15   % SECOND is the watch internal quartz
16   % SET_HOUR is the hour update button
17   % SET_MINUTE is the minute update button
18   input         SECOND,
19                 SET_HOUR,
20                 SET_MINUTE;
21
22   % Broadcasts the updated time value.
23   output        TIME: TIME;
24
25   % All external events are supposed to be exclusive
26   relation      SECOND # SET_HOUR # SET_MINUTE;
27
28   % We keep the current time value in the TIME variable
29   var TIME := INITIAL_TIME : TIME in
30
31     % Emission of the initial time value
32     emit TIME (TIME);
33     loop
34       await
35         case SECOND do
36           call INCREMENT (TIME) (ONE_SECOND)
37         case SET_MINUTE do
38           call RESET_SECONDS (TIME) ();
39           call INCREMENT (TIME) (ONE_MINUTE)
40         case SET_HOUR do
41           call INCREMENT (TIME) (ONE_HOUR)
42       end; % await
43
44       % Emission of the updated value of TIME
45       emit TIME (TIME)
46     end % loop
47   end % var
48 end. % module
```

Esterel-Code 1.1: Eine einfache Armbanduhr [6]

Das Beispiel demonstriert die Verwendung eines benutzerdefinierten Typs sowie die Deklaration von Konstanten, das Setzen von Relationen, das Deklarieren und Aufrufen von Prozeduren sowie das Übergeben von Variablen und Werteparametern und die Verwendung des multiplen `await`-Befehls.

Die Schnittstellendefinition zu diesem Beispiel ist zum Teil im Exkurs „Schnittstelle zwischen ESTEREL und C “ auf Seite 7 gegeben.

B Compiler

Für ESTEREL wurden diverse Compiler geschrieben. Unter diesen kann zwischen Software- und Hardwarecompiler unterschieden werden. Die Softwarecompiler kategorisieren DAYARATNE, ROOP und SALCIC in „Direct Execution of Esterel Using Reactive Microprocessors“ [9, S. 5 f.] anhand der verwendeten Algorithmen und Datenstrukturen nach vier Typen:

Automaten-basierte Compiler Diese Compiler übersetzen ESTEREL-Quellcode zunächst in einen Zustandsautomaten, welcher den gesamten Zustandsraum des Programms abdeckt. In der Exkursion „Werkzeuge zur Entwicklung von reaktiven Echtzeitsystemen “ auf Seite 3 sind wesentliche Vor- und Nachteile automatenbasierter Systeme genannt. In dem dieser Seminararbeit zugrunde liegendem Papier „The ESTEREL synchronous programming language: design, semantics, implementation“[7, S. 140] beschreiben BERRY und GONTHIER die Übersetzung von ESTEREL-Programmen mit dem ESTEREL-Compiler V3 in deterministische Automaten.

Logikgatter-basierte Compiler Diese Compiler übersetzen ESTEREL-Quellcode zunächst in ein Schaltnetz aus logischen Gattern und generieren dann daraus einen gestuften Code-Simulator. Dieser generierte Code verwendet einen minimalen Satz an Befehlen, um den Originalcode abzubilden. Der Vorteil dieses Ansatzes gegenüber dem umfassenden Zustandsautomaten ist, dass keine Codeduplizierung erfolgt und der resultierende Code kompakter ist. BERRY verfolgte diese Strategie mit den späteren Compilerversionen V4 und V5.

Diskrete Event-basierte Compiler . Diese Compiler brechen den ESTEREL-Quellcode in kleinere funktionale Blöcke auf, deren Ausführung von einem Scheduler organisiert wird.

Fork-Join-Compiler Compiler diesen Typs produzieren den effizientesten Code von allen der bislang betrachteten Ansätze. Ein Beispiel für einen Compiler dieser Kategorie ist der „Columbia Esterel Compiler“ (CEC) [10] . EDWARDS, SOVIANI und ZENG entwarfen für den Compiler einen Algorithmus, welcher ein nebenläufiges ESTEREL-Programm in ein sequentielles Äquivalent durch den Einsatz der C-Befehle `fork()` und `join()` überführt.

Trotz aller Anstrengungen hinsichtlich der Ausführungsoptimierung haben alle dieser Ansätzen einen Nachteil gemein: Der Versuch, die Nebenläufigkeit von ESTEREL indirekt zu simulieren. Sei es durch Scheduling, Abbildungen von Sensoren und Signalen auf Ports eines Mikrocontrollers oder weitere Mechanismen wie das indirekte Abbilden präemptiver ESTEREL-Konstrukte mittles *Interrupts* oder *Polling*.

Hier wird deutlich, dass das theoretische ESTEREL-Modell uns zwar „echte“ nebenläufige Synchronität und Reativität vorleben will, diese aber durch Beschränkungen der real existierenden Systeme nur simuliert werden können.