Freie Universität Berlin
Seminar für Programmiersprachen
Wintersemester 14/15

# Die Programmiersprache Esterel

Zell, Christopher
zell.christopher@fu-berlin.de

21. November 2014

## 1 Einleitung

Die folgende Ausarbeitung behandelt die Programmiersprache Esterel und insbesondere das Paper »The ESTEREL synchronous programming language: design, semantics, implementation*« von Berry und Gonthier es wurde im Jahre 1988 Veröffentlicht und seitdem ca. 2000 mal referenziert, siehe dazu Scholar 2014. Gérard Berry wurde 1948 geboren und ist ein französischer Informatiker und derzeit Professor am *Collège de France*. Neben seinen Arbeiten an parallelen sowie *real-time programming* und dem Lambda Kalkül gilt er als einer der ursprünglichen Entwickler der Programmiersprache Esterel (vgl. Berry 2014). In dieser Ausarbeitung soll Esterel näher gebracht werden, es wird dabei nur auf den Sprachumfang eingegangen und dieser mit nicht formaler Semantik erklärt. Für weitere bzw. tiefere Informationen und formale Semantik Definitionen siehe Boussinot und Simone 1991 oder Berry und Gonthier 1988.

Esterel hat sein Beginn im Jahre 1984, als zwei Französische Forscher (Jean-Paul Marmorat and Jean-Paul Rigault) ein Robotisches Automobil zu design versuchten. Sie waren aber durch ihre Versuche frustriert die Kontrollalgorithmen auf natürlichem und kraftvollen Wege Auszudrücken, somit kreierten sie eine originelle mathematische formale Notation welche ihnen erlaubte das Auto zu kontrollieren. Diese natürliche Kontrollsprache genannt Esterel , der Name stammt von dem gleichnamigen Mittelgebirge welches in Frankreich liegt, bekam immer mehr Aufmerksamkeit von anderen Forschern und Instituten. Die Sprache Esterel begann sich zu entwickeln und bekam erste Semantiken, erste Codeerzeugungen sowie Formale Verifikationswerkzeuge welche von Gérard Berry entwickelt wurden (vgl. Esterel-Technologies 2014).

## 2 Motivation

Für Gérard Berry und anderen Forschern war die Primäre Motivation zur Entwicklung von Esterel das bessere Verständnis von *real-time programming*, insbesondere zeitliche Merkmale bzw. Funktionen wie sogenannte *watchdogs* (vgl. Boussinot und Simone 1991, S. 1293), dazu mehr in Kapitel 4. Denn diese Sprache ermöglichte bzw. eröffnete ganz neue Pfade im parallelen Programmieren bzw. im Programmieren von Echtzeitsystemen. Im folgenden Abschnitt werden kurz einige Grundlagen geklärt sodass die Sprache Esterel ohne Probleme vorgestellt werden kann.

# 3 Grundlagen

## 3.1 Synchrone Programmiersprachen

Synchrone Sprachen sind speziell zum Programmieren von Reaktiven Systemen (siehe Abschnitt 3.2) entwickelt und designed worden. D.h. eine Vielfalt von computergesteuerten Systemen die Echtzeitsysteme mit einschließt, sowie alle Arten von Kontrollautomaten. Sie zeichnen sich durch ihre Nebenläufigkeit und ihren Determinismus aus. Neben ESTEREL existieren noch weitere synchrone Sprachen wie LUSTERE, SIGNAL und STATECHARTS (vgl Berry und Gonthier 1988, S. 88) .

## 3.2 Reaktive Systeme

Um *reaktive Systeme* zu definieren, müssen wir zuerst *reaktive Programme* definieren. *Reaktive Programme* sind Programme die eine permanente Interaktion mit der Umgebung aufrechterhalten. D.h. Eingaben von der Umgebung erhalten durch Senden von Ausgaben an diese. (Interaktion) Berry und Gonthier definieren ein *reaktives System* dann wie folgt: Ein *reaktives System* ist ein System mit einem *reaktiven Programm* als Hauptkomponente.

Beispiele für *reaktive Systeme*:

1. Mikrocontroller
2. Digitaluhren
3. Videospiele
4. Kommunikationsprotokolle
5. Man-Maschinen Schnittstellentreiber

Beispiele für *Reaktive Programme*

- Betriebssystemtreiber
- Maus/Tastatur Schnittstellentreiber
- Sender und Empfänger in Kommunikationsprotokollen

*Reaktive Programme* sind in drei Schichten aufgebaut:

***interface*** für die Kommunikation mit der Umgebung (input/output),

***reactive kernel*** beinhaltet die Logik

***data handling*** führt klassische angefragte Berechnungen für den *reactive kernel* aus

Das Paper von ebd. behandelt zum Großteil den *reactive kernel* , da er das zentrale und schwerste Element eines *reaktiven Systems* darstellt. Determinismus ist ein wichtiger Charakterzug eines *reaktiven Programms*. Soll heißen ein deterministisches *reaktives Programm* Produziert eine identische Ausgabe mit einer identischen Eingabe. Dabei ist zu beachten das die Zeit meist auch als Teil der Eingabe betrachtet wird. Vorteil von deterministischen Systemen ist das diese einfacher zu spezifizieren, debuggen und zu analysieren sind, als nicht deterministische. Deterministische Nebenläufigkeit ist der Schlüssel zu modularer Entwicklungen von *reaktiven Programmen* und dies wird von Synchronen Programmiersprachen wie ESTEREL unterstützt.

# 4 Esterel

Esterel ist speziell zum Programmieren von *reaktiven Systemen* entwickelt worden. Es handelt sich dabei um eine deterministische nebenläufige Programmiersprache, sie ist die älteste und meist entwickeltste Sprache aus der Familie der Synchronen Programmiersprachen (vgl. Berry und Gonthier 1988, S. 88). Die mathematische Semantik der Sprache Esterel wurde zusammen mit der Sprache entwickelt, die Implementierung ist einfach die physikalische Realisierung der Semantik. Ohne diese erwähnte Mathematische Semantik würde es Esterel nicht geben. Ein Vorteil dieses formalen Vorgehens ist das Esterel Programme effizient in endliche Automaten übersetzt werden können sowie effizienter Code erzeugt werden kann (vgl. Boussinot und Simone 1991, S. 1293). Vielleicht ist sogar das erzeugen dieses endlichen Automatens die stärkste Charakteristik von Esterel , denn es bringt einige Vorteile mit sich. Automaten können effizient ausgeführt werden, da kein Parallelismus im Automaten existiert werden die unnötigen Prozessmanagment Aufgaben oder Synchronisation erspart. Ein weiterer Vorteil ist die Berechenbarkeit, die maximale Übergangszeit in einem Automaten ist berechenbar. Der Esterel v3 Compiler, geschrieben in C++ produziert Automaten in Code Ausgabenformat. Dieses Format kann in viele Sprachen übersetzt werden. C ist die Standard Zielsprache des Esterel v3 Compilers. D.h. beim übersetzen eines Esterel Programms wird ein C compilat erstellt ( vgl. ebd., 1298ff). Esterel ist nicht vollends entwickelt bzw. implementiert, es wird eine *Host-sprache* benötigt um alle Schichten zu implementieren.

> »In fact, Esterel is not a full-fledged programming language, but rather a programm generator used to program reactive kernels in the same way as *YACC* is used to program parses for grammers.«
>
> Berry und Gonthier 1988, S. 89

Das bedeutet Esterel kann nur für die *reactive kernel* Schicht verwendet werden, um die weiteren Schichten eines *reaktiven Systems* (siehe 3.2) zu implementieren bzw. zu spezifizieren muss eine sogenannte *Host-sprache* verwendet werden. Wie wir bereits gesehen haben in 3.2 ist deterministische Nebenläufigkeit essentiell für das Programmieren von *reaktiven Programmen*. Jedoch gibt es auch Tools für das *reaktive Programmieren* die dies nicht beachten und es somit zu Problemen kommt. Zum einen wäre da der deterministische Automat der oft benutzt wird zum Programmieren von relativ kleinen *reactive kernels*, typischerweise in Protokollen oder Controllern. Das Petri-Netz welches z.B. in dem Graflet Tool benutzt wird für Programmierbare Controller. Sequentielle Aufgaben welche unter Echtzeit-Betriebssystemen laufen. Sowie nebenläufige Programmiersprachen wie z.B. Ada oder Occam. Diese liefern oft dem Nutzer *interface* und *data manipulation* Hilfsmittel die es ermöglichen das alle Schichten in einer Sprache Implementiert werden können (siehe 3.2). Das Problem bei diesen Tools ist das der Nutzer zwischen Determinismus und Nebenläufigkeit wählen muss. Was zu weiteren Problemen führt. Diese Probleme verschwinden jedoch wenn man die sogenannte Synchrone Hypothese übernimmt. D.h. jede Reaktion wird als augenblicklich und somit atomar angenommen. Die Synchronität läuft darauf hinaus das man annimmt das die zugrunde liegende Maschine keine Zeit benötigt um eine Operation auszuführen. Es können somit mehrere Signale empfangen oder versendet bzw. mehrere Anweisung ausgeführt werden innerhalb eines Augenblicks (vgl. Boussinot und Simone 1991, S. 1294).

> »To ”take no time” has to be understood in a very strong sense.«
>
> Berry und Gonthier 1988, S. 91

Soll heißen Anweisungen verbrauchen nur dann Zeit und auch nur dann wenn sie das auch ausdrücken. Temporäre Anweisungen meinen genau das was sie sagen. Folgendes Beispiel wartet genau 30 ms.

```
await 30 MILLISECOND
```

Beispiel 1: Await

```
every 1000 MILLISECOND do emit SECOND end
```

Beispiel 2: Every

Das zweite Beispiel bedeutet das ein SECOND Signal **genau** alle 1000 MILLISECOND gesendet wird. (1000 ms = 1s) Weitere Informationen zu den Anweisungen in Abschnitt 4.1 und 4.1.5. In einem asynchronen Formalismus ist eine Sekunde niemals synchron mit eine Millisekunde. Synchronität ist vom Standpunkt des Benutzers völlig natürlich. Nehmen wir z.B. ein Benutzer einer Uhr. Er macht sich keine Gedanken über die internen Reaktionszeiten, solange seine Uhr sofort auf seine Kommandos reagiert. Synchronität ist auch vom Standpunkt des Programmierers aus natürlich, es erlaubt das wieder zusammenführen von Determinismus und Nebenläufigkeit um einfachere und gründlichere Programme zu schreiben (vgl. Berry und Gonthier 1988, 91f).

Berry und Gonthier heben hervor das ESTEREL eine Programmiersprache ist die kleinen und effizienten Objektcode liefert und nicht einfach eine idealisierte Spezifikationssprache ist, welchen den Benutzer dazu bringt sein Programm neu zu schreiben nachdem die Spezifikation vollendet ist. Die Autoren ebd. merken auch an das diese betrachtete Synchronität auch in anderen Bereichen vorkommt bzw. verwendet wird. Z.B. in der Physik ist der Begriff klassisch, sofortige Interaktionen ist die Grundlage von Netwons Mechanik (Gesetzen) oder in der Elektrik ist es die Basis von Kirchoffs Gesetzen.

Im Gegensatz zu den genannten Sprachen in Abschnitt 3.1 übernimmt ESTEREL eher ein klassischen imperativen Stil. Die ESTEREL Anweisungen behandeln entweder klassische Variablen Zuweisungen, die lokal jnebenläufig zu Anweisungen sind und die nicht geteilt werden können oder Signale die benutzt werden um mit der Umgebung und zwischen nebenläufigen Prozessen zu Kommunizieren. Signale haben einen Status und können eine Wert beliebigen Typs besitzen. Der Status ist entweder absence (engl. Abwesenheit) oder presence (engl. Anwesenheit). Signale werden sofort versendet, das besagt das sogenannte "sharing law of signals" (vgl. ebd., S. 92). Dazu später mehr in Abschnitt 4.1.2. Wie bereits zuvor kurz erwähnt gibt es zwei verschiedene Arten von Anweisungen in der ESTEREL Programmiersprache. Zum einen Standard imperative Anweisungen (wie z.B. Zuweisung, Signal Aussendung, Bedingungsanweisungen, Schleifen etc.). Bei diesen Anweisungen wird angenommen das sie auf einer unendlich schnellen Maschine ausgeführt werden. Soll heißen das z.B. die "null" Anweisung nichts tut, mit keinem Zeit verbrauch. Die andere Art von Anweisungen sind die, wie bereits gezeigten, temporären Anweisungen. Wie z.B. sogenannte *trigger* (await event do ...), *watchdogs* (do ... watching event) oder temporäre Schleifen (loop ... each event). Im folgendem Abschnitt wird auf beide Arten tiefer eingegangen.

## 4.1 Sprachumfang

Die Autoren Berry und Gonthier zeigen auf das es neben dem sogenannten BASIC ESTEREL auch ein sogenanntes PLAIN ESTEREL gibt, welches im Grunde eine Erweiterung der Basis Version darstellt. PLAIN ESTEREL soll das benutzen von ESTEREL vereinfachen. Im folgendem Abschnitt werden zuerst die Basisanweisungen aufgezeigt und evtl. mit PLAIN ESTEREL verglichen. Im Abschnitt 4.1.5 findet sich noch eine kurze Abhandlung von PLAIN ESTEREL direkt. Das heißt es werden kurz die Erweiterungen von BASIC ESTEREL vorgestellt. Die gezeigten Beispiel wurden dem ebd., 95ff. entnommen.

### 4.1.1 Module und Globale Deklarationen

In beiden BASIC ESTEREL und PLAIN ESTEREL stellt das Modul mit dem Schlüsselwort *module* eine Programmiereinheit dar, ähnlich einer Klasse oder Funktion. Ein Modul besteht aus einem Namen, Deklarationsteil und einem Körper der Anweisungen beinhaltet.

```
1  module MOD :
2    declaration part
3    body
```

Beispiel 3: Module

Innerhalb des Deklarationsteil werden externe Objekte deklariert die vom Modul benutzt werden. Wie z.B. Datenobjekte die in der *data handling* Schicht implementiert sind, Signale und Sensoren die das reaktive *interface* definieren. Diese Deklarationen sind miteinander Verflochten, da Signale und Sensoren Werte beinhalten können deren Typen in dem Deklarationsteil deklariert wurden. Alle Objekte müssen deklariert werden bevor sie benutzt werden können. ESTEREL besitzt ein paar primitive Typen jedoch

keine zusammengesetzten wie z.B. Arrays. D.h. komplexe Datenstrukturen bzw. komplexe Datenhandhabung werden abstrakt gehalten. Die abstrakten Typen und deren Funktionen die diese verändern sind nur mit ihrem Namen bekannt. Die Implementierung erfolgt dann in der *Host-sprache* .

BASIC ESTEREL besitzt drei Primitive Datentypen: *integer*, *boolean* und *triv* (mit einer einzigartigen Konstante die auch *triv* genannt wird). *Triv* wird beim übersetzen von PLAIN zu BASIC ESTEREL benutzt um die reinen Signale (d.h. Signale ohne Werte) in BASIC ESTEREL Signale mit Werten vom Typ *triv* umzuwandeln. PLAIN ESTEREL besitzt des Weiteren noch die Typen *string* und *float*. Der Nutzer kann auch seine eigenen abstrakten Datentypen mithilfe des Schlüsselworts *type* deklarieren, siehe Beispiel 4.

```
1          type DOUBLE, TIME;
```

Beispiel 4: Type Deklaration

Zudem kann man Konstanten von vordefinierten oder abstrakten Typen deklarieren, siehe Beispiel 5.

```
1          constant MEASURE_NUMBER : integer, PI : DOUBLE, NOON : TIME;
```

Beispiel 5: Konstanten Deklaration

Die Typen müssen vorher deklariert worden sein zudem werden die Werte für die Konstanten in der *Host-sprache* zugewiesen. Genauso wie bereits erwähnt die Implementierung von Funktionen. Es wird angenommen das Funktionen frei von Seiteneffekten sind. Sie werden wie folgt deklariert:

```
1          function SQRT(DOUBLE) : DOUBLE,
2                   EQUAL_TIME (TIME, TIME) : boolean;
```

Beispiel 6: Funktions Deklaration

In ESTEREL wird zwischen Prozeduren und Funktionen unterschieden. Prozeduren besitzen zwei Argumentlisten. Die erste Liste beinhaltet Variablen die per Referenz übergeben werden. Sie sind somit Subjekte von Seiteneffekten, ähnlich wie *var* Parameter in PASCAL oder *inout* Parameter in ADA. Die zweite beinhaltet normale Parameter. In der Deklaration werden nur die Argumenttypen deklariert (vgl. Berry und Gonthier 1988, S. 96).

```
1          procedure INCREMENT_TIME_BY_SECONDS (TIME) (integer)
```

Beispiel 7: Prozedur Deklaration

Das Beispiel 7 zeigt eine Deklaration einer Prozedur zum erhöhen einer Zeitvariable um eine bestimmte Anzahl von Sekunden (vgl. ebd., S. 96).

### 4.1.2 Signale

Zu allem anderen muss man auch die Sensoren und Signale deklarieren die das reaktive *interface* darstellen. Ein Sensor stellt ein degeneratives Signal im PLAIN ESTEREL dar. Zudem kann man auch Eingabebeziehungen deklarieren, die mögliche Eingabeevents beschränken und wichtig für Übersetzerprogramme sind. Variablen und Signale unterscheiden sich in dem Punkt das Signale nur geteilt werden können. Innerhalb von Anweisungen gibt es keinen Unterschied zwischen Eingabe-, Ausgabe- oder lokalen Signalen.

Signale haben unmittelbare *ticks* (z.B. Interrupts) die als Kontrollinformation für temporäre Anweisungen dienen. Wie bereits zuvor schon einmal erwähnt können Signale auch Werte eines beliebigen Typs besitzen. Diese Werte können mit dem *?* Operator abgefragt werden. D.h. für ein Signal S kann via folgenden Ausdruck "?S" der Wert des Signals S abgefragt werden. Für Eingabesignale wird folgende Beziehung angenommen: Der Wert eines Signals kann sich nur ändern wenn ein *tick* erfolgt (d.h. ein neues Signal versendet bzw. reinkommt). In diesem Fall ersetzt der neue Wert den alten sofort. Dies gilt auch automatisch für ausgehende und lokale Signale. In PLAIN ESTEREL gibt es eine spezielle Sensor Deklaration für passive externe Geräte, wie zum Beispiel Thermometer. Welche auch auf Abfrage Werte liefern aber keine Ticks generieren. Nur die Wertzugriffsoperation "?" ist für Sensoren verfügbar.

Im BASIC ESTEREL existieren nur zwei Arten von *interface* Signalen: Eingabe- und Ausgabesignale. Eingabesignale kommen von der Umgebung und können nicht intern in BASIC ESTEREL erzeugt bzw. gesendet werden. Sie werden wie folgt deklariert:

```
1         input S (type);
```

Beispiel 8: Basis Eingabesignal Deklaration

Ausgabesignale werden an die Umgebung mithilfe des Schlüsselworts *emit* abgegeben. *emit S(exp)* gibt ein Signal S mit dem Wert des Ausdrucks *exp* an die Umgebung.

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

Beispiel 9: Paralleles emit

In dem Beispiel 9 wird das Signal S gleichzeitig von zwei verschiedenen Sendern mit zwei verschiedenen Werten gleichzeitig gesendet. Wobei der Operator "||" der ESTEREL Paralleloperator ist. In diesem Fall kommt es zu einer sogenannten Kollision. Mithilfe einer Kombinationsfunktion *comb* kann der Wert für ein Signal S bei einer Kollision definiert werden. Es wird jedes Ausgabesignal mit einer *comb* Funktion verbunden. D.h. wenn Sender ein Signal mit folgenden Werten senden: $v_1, v_2, \ldots, v_n$ dann ist der Wert von S: $comb(v_1, comb(v_2, \ldots comb(v_{n-1}, v_n)\ldots))$ Ein Ausgabesignal wird wie folgt deklariert:

```
1         output S (combine type with comb);
```

Beispiel 10: Basis Ausgabesignal Deklaration

Wobei *type* und *comb* bereits deklariert sein müssen. Die Funktion *comb* wird wie eine Funktion mit zwei Parametern deklariert.

Wenn die Signale keine wirklichen Werte besitzen bzw. diese sinnlos/bedeutungslos sind, wie z.B. bei SECOND etc., dann muss im BASIC ESTEREL der *triv* Typ benutzt werden. Im PLAIN ESTEREL kann die Typangabe einfach vernachlässigt werden, siehe Beispiel 11.

```
1         input SECOND, METER;
2         output ALARM;
```

Beispiel 11: Plain Ein- und Ausgabesignal Deklaration

Falls klar ist das die Signale keine Kollision auslösen können kann die Angabe einer *comb* Funktion bei der Deklaration von Ausgabesignalen auch weggelassen werden. Der ESTEREL Compiler prüft dann ob nie eine Kollision auftauchen kann. In BASIC ESTEREL wird strikt zwischen Ein- und Ausgabe unterschieden, innerhalb von PLAIN ESTEREL gilt diese Unterscheidung nicht allzu strikt. Das erlaubt das Signale von beiden Typen sein können, siehe Beispiel 12.

```
1         inputoutput BUS_REQUEST;
```

Beispiel 12: Plain InOut-Signal Deklaration

Sensoren in PLAIN ESTEREL werden wie folgt deklariert:

```
1         sensor TEMPERATURE (FAHRENHEIT);
```

Beispiel 13: Sensor Deklaration

Der Compiler prüft das keine temporären Instruktionen wie z.B. *delays* (*watchdogs*, *presence* etc.) an Sensoren angewendet werden.

### 4.1.3 Beziehungen

Wie bereits vorher erwähnt schränken Deklarationen von Eingabeziehungen die möglichen Eingabeevents von Modulen ein. Es gibt zwei Arten von Beziehungen: Zum einen die *incompatibility* (Unverträglichkeit) Beziehung der Form $S_1 \# S_2 \# S_3$. Diese Beziehung zeigt an das die Signale $S_1, S_2$ und $S_3$ sich

gegenseitig in Eingabeevents ausschließen. Die zweite Art ist eine Synchronitätsbeziehung $S_1 => S_2$. Diese Beziehung besagt das $S_2$ in einer Eingabe vorkommt immer wenn $S_1$ vorkommt. D.h. $S_1$ impliziert $S_2$.

```
1           relation LEFT_BUTTON # RIGHT_BUTTON,
2                    SECOND => HUNDREDTH_OF_SECOND;
```

Beispiel 14: Plain Eingabebeziehung Deklaration

Mithilfe dieser Eingabebeziehungen kann man z.B. eine Spezifikationsanforderung umsetzen die besagt das Signale nicht zusammen auftauchen sollen. Zudem sind diese Beziehungen essentiell zum reduzieren der Grösse von generierten Automaten.

### 4.1.4 Anweisungen

Im folgenden Abschnitt werden die Basisanweisungen der Sprache ESTEREL gezeigt und dazu eine nicht formale Semantik gegeben (vgl. Berry und Gonthier 1988, S. 100-103).

| | |
|---|---|
| nothing | Nothing führt keine Aktion aus und endet sofort. Ähnlich einem NOP in anderen Sprachen. |
| halt | Führt keine Aktion aus, endet nie und verlässt auch keine *traps*. |
| $X := exp$ | Eine Zuweisung aktualisiert den Speicher und schließt sofort ab. |
| call P (variable list) (exp list) | Ein Prozeduraufruf aktualisiert den Speicher und schließt sofort ab. |
| emit S($exp$) | Wenn die Anweisung startet wird zuerst der Ausdruck *exp* ausgewertet. Das Signal bekommt den Wert der aus dem Ausdruck resultiert und wird versendet, danach endet die Anweisung. |
| $stat_1; stat_2$ | Stellt eine Sequenz von Anweisungen dar. Die erste Anweisung wird ausgeführt wenn die Sequenz gestartet wird. Wenn die erste Anweisung eine *trap* verlässt dann wird die Sequenz verlassen und die zweite Anweisung nicht mehr ausgeführt. Wird die erste Anweisung beendet wird die zweite Anweisung sofort gestartet. |
| loop *stat* end | Diese Anweisung stellt eine Endlosschleife dar. Der Körper einer Schleife startet wenn eine Schleife startet. Wenn der Körper terminiert dann wird die Schleife neu gestartet. Eine Schleife terminiert jedoch niemals von allein. Wenn eine Anweisung innerhalb des Körpers eine *trap* verlässt, dann wird auch die Schleife verlassen. |
| if *exp* then $stat_1$ else $stat_2$ end | Wenn eine bedingte Anweisung gestartet wird, wird die Bedingung sofort evaluiert. Je nachdem ob die Bedingung *true* oder *false* ist wird $stat_1$ bzw. $stat_2$ ausgeführt. |
| present S then $stat_1$ else $stat_2$ end | Das Verhalten gleicht sich zu der bedingten Anweisung nur das als Bedingung das Vorkommen bzw. die Anwesenheit des Signals S in der derzeitigen Reaktion geprüft wird. |
| do stat watching S | Stellt einen sogenannten *watchdog* dar. Gibt ein Zeitlimit an für die Ausführung der *stat* Anweisung. Das Limit wird dadurch dargestellt das die Bearbeitung beendet wird sobald bei einer Reaktion das Signal S vorkommt. Falls der Körper bzw. *stat* terminiert oder eine *trap* verlässt bevor S vorkommt so wird die *watching* Anweisung auch verlassen. Kommt Signal S vor wird der Körper sofort terminiert, ohne weiter ausgeführt zu werden. |

| | |
|---|---|
| $stat_1 \| \| stat_2$ | Die zwei parallelen Zweige (branches) starten wenn die Anweisung ausgeführt wird. Wenn eine der Anweisung eine *trap* verlässt so tut es die andere auch und beide Zweige sind ab diesem Zeitpunkt inaktiv. Wenn beide gleichzeitig mehrere *traps* verlassen wird nur die äußerste verlassen. Diese Anweisung terminiert wenn alle Zweige beendet wurden. |
| trap T in stat end | Der Körper wird sofort gestartet und entscheidet das Verhalten der *trap* Anweisung. Wenn der Körper terminiert oder eine *exit T* Anweisung vorkommt wird die *trap T* Anweisung sofort beendet. Falls der Körper eine *trap T'* verlässt wird diese auch Verlassen. |
| exit T | Diese Anweisung verlässt *T* und terminiert nicht. |
| var X : *type* in stat end | Eine lokale Variablen Deklaration, deklariert eine Variable lokal für die Anweisung *stat* und initialisiert diese mit $\perp$ |
| signal S (combine type with comb) in stat end | Eine lokale Signal Deklaration, deklariert ein Signal lokal für die Anweisung *stat* und initialisiert dieses mit $\perp$ |

Für BASIC ESTEREL gelten folgende Einschränkungen: Zum einen können Eingabesignale nicht intern versendet werden. Des Weiteren ist der "?" Operator für Ausgabesignale nicht erlaubt. Diese Einschränkungen werden bei PLAIN ESTEREL unterdrückt, jedoch erzeugt der Compiler trotzdem Warnungen wenn sie nicht eingehalten werden. Für Schleifen gibt es zudem auch eine Einschränkung die es ist nicht erlaubt das der Schleifenkörper sofort terminieren kann. Dies wird mithilfe des Compilers beim Übersetzen geprüft. Damit Anweisungensfolgen wie folgende Verhindert werden können:

```
1  loop
2    X := X + 1
3  end
```

Beispiel 15: Endlosschleifenzähler

Das Beispiel 15 zeigt eine Endlosschleife mit einer Variable die immer wieder um eins erhöht wird. D.h. das unendlich viele sofortige Additionen und Speicheraktualisierungen stattfinden. Dies soll durch die genannte Einschränkung verhindert werden. Wie bereits schon erwähnt verbrauchen die Anweisungen keine Zeit. Um jedoch Zeit zu verbrauchen kann die *halt* Anweisung benutzt werden, da diese Anweisung unendlich viel Zeit verbraucht da sie nie endet. Mithilfe eines *watchdogs* kann dies aber eingeschränkt werden.

```
1  do halt watching S
```

Beispiel 16: Await synonym

Das Beispiel 16 ist das einfachste Beispiel welches ermöglicht auf das nächste auftauchen eines Signals zu warten. Innerhalb von PLAIN ESTEREL wird dieses Konstrukt mit *await S* abgekürzt.

### 4.1.5 Plain Esterel

Wie bereits erwähnt ist PLAIN ESTEREL die Erweiterung und benutzerfreundlichere Variante der Basisversion. PLAIN ESTEREL zeichnet sich durch drei Erweiterungen gegenüber der Basisvariante aus. Zum einen der Signale und deren Interface, benutzerfreundlichere Anweisungen und die *copymodule* Anweisung die für modulares programmieren benutzt wird, dazu später mehr. Zu den erweiterten Anweisungen gehört z.B. das initialisieren von Variablen während der Deklaration, das deklarieren von von mehreren Signalen oder Variablen in der lokalen Deklaration. Des Weiteren eine *repeat* Schleife siehe Beispiel 17.

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

Beispiel 17: Repeat-Schleife

Diese Schleife ist im Grunde ein Konstrukt aus einer *loop* Schleife innerhalb einer *trap* Anweisung mit einer *if-then-else* Anweisung. Innerhalb der bedingten Anweisung wird die *trap* verlassen, wenn der Zähler erreicht wird, siehe Beispiel 18.

```
1  trap REPEAT_T in
2    loop
3      if COUNT == MAX then
4        exit REPEAT_T
5      else
6        stat
7    end
8  end
```

Beispiel 18: Repeat-Schleife im Detail

Um festzustellen ob eine *watching* Anweisung durch ein Signal vorkommen oder dadurch das der Körper terminierte beendet wurde, wurde die *timeout* Anweisung eingeführt. Diese ermöglicht auch eine weitere Anweisung auszuführen falls die *watching* Anweisung durch ein Timeout beendet wurde (Signal vorkommen), siehe Beispiel 19.

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

Beispiel 19: Timeout

Dieses Konstrukt ist die Abkürzung für das Beispiel 20 aus dem BASIC ESTEREL .

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

Beispiel 20: Timeout Basis Version

Die *timeout* Anweisung betrachtet nicht den Fall wenn das Signal zu Beginn/sofort vorkommt, deshalb gibt es im PLAIN ESTEREL die *immediate* Anweisung.

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

Beispiel 21: Immediate

Wie man im Beispiel 22 sieht wird zuerst geprüft ob das Signal S vorhanden ist, falls nicht wird der *watchdog* gestartet.

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

Beispiel 22: Immediate Basisversion

Eine weitere benutzerfreundlichere Anweisung ist die *upto* Anweisung. Die *upto* Anweisung ist ähnlich der *watching* Anweisung, jedoch wird sie nicht beendet wenn der Körper terminiert. Das heißt somit kann garantiert werden das die Anweisung mit ihrem Körper bis zum vorkommen des Signals S ausgeführt wird. Es sei denn der Körper terminiert eine *trap* oder ähnliches. Die Anweisung *do stat upto S* wird in der BASIC ESTEREL Version wie folgt übersetzt.

```
1  do stat;
2    halt
3  watching S
```

Beispiel 23: Upto Basisversion

Um bei jedem vorkommen eines Signals etwas auszuführen wurde in PLAIN ESTEREL die Anweisung *every S do stat end* eingeführt. Diese Anweisung wird wie folgt in BASIC ESTEREL umgesetzt.

```
1  await S;
2  loop
3    do stat upto S
4  end
```

Beispiel 24: Every Basisversion

Dazu ist zu sagen das die einzelnen Konstrukte hier wieder mit den PLAIN ESTEREL synonymen abgekürzt werden. Zur Erklärung: es wird als erstes auf das Signal S gewartet und dann der Körper der *upto* bzw. *watching* Anweisung ausgeführt. Danach wird immer nach dem vorkommen des Signals wieder der Körper Ausgeführt. Die andere Variante ist die, die den Körper ausführt und dann bei jeden vorkommen von S dies wieder tut. *loop stat each S* siehe Beispiel 25 für die Basisübersetzung.

```
1  loop
2    do stat upto S
3  end
```

Beispiel 25: Each Basisversion

PLAIN ESTEREL bringt zudem ein allgemeines *exception handling* mit, das den *trap* Mechanismus erweitert. Dieses verhält sich im Grunde wie ein *try-catch* Block in anderen Sprachen (z.B. Java). Es können *exit handlers* definiert die ausgeführt werden wenn solch ein *exit* bzw. eine *exception* auftaucht. Diese *exceptions* können ähnlich wie Signale auch Werte beinhalten. Jedoch können diese Werte nur innerhalb eines Handlers abgefragt werden und um den Unterschied zu Signalen zu verdeutlichen nur mit dem *??* Operator. Wenn mehrere *exits* bzw. *exeptions* gesendet werden dann wird bei verschachtelten *traps* nur die äusserste behandelt. Handelt es sich aber nur um eine *trap* dann werden die Handler parallel ausgeführt. Das Beispiel 26 soll eine Vorstellung vom *exception handling* bieten (vgl. Berry und Gonthier 1988, S. 110).

```
1  trap ALARM (combine integer with +)
2         ZERO_DIVIDE, TERMINATE in
3       stat
4  handle ALARM do stat1
5  handle ZERO_DIVIDE do stat2
6  end
```

Beispiel 26: Exception Handling

Wie bereits erwähnt beinhaltet die Erweiterung durch PLAIN ESTEREL auch die *copymodule* Anweisung. Diese Anweisung kann überall in dieser Form benutzt werden: *copymodule MODULE*, wobei *MODULE* für das Modul steht welches Kopiert werden soll. D.h. die Anweisung wird mit dem Text des Moduls nach Konsistenz Verifizierung des Interfaces und der Deklarationen ersetzt. Somit wird die modulare Programmierung ermöglicht.

# 5 Weiterentwicklung

In dem Paper ebd. wird ab dem Abschnitt 6 die sogenannte *behavioral* Semantik für die ESTEREL Programmiersprache definiert bzw. vorgestellt/gezeigt. Für die denotationelle Semantik wird auf andere Quellen verwiesen. Die *behavioral* Semantik dient als formale Definition der Sprache ESTEREL . Jede operationelle Semantik sollte mit dieser, laut den Autoren Berry und Gonthier, übereinstimmen (vgl. ebd., S. 114). Später präsentieren sie noch eine etwas komplexere Ausführungssemantik.

Das Paper welches als Hauptquelle für diese Ausarbeitung benutzt wurde stammt aus dem Jahre 1988. Es wurde danach weiter intensiv an der Sprache weiterentwickelt. Im Jahre 1999 wurde die Firma ESTEREL Technologies gegründet welche sich mit der Vermarktung von ESTEREL Produkten beschäftig. Sie wurde zu einem der führenden Lieferanten für kritische Systeme und Softwareentwicklungslösungen für viele Abnehmer unter anderem für die Luftfahrt-, Verteidigungs-, Zugfahrt- und auch Nuklearindustrie (Esterel-Technologies 2014). Die aktuellste Version des ESTEREL Compilers ist die Version 7,

innerhalb des Papers Berry und Gonthier 1988 wurde die Version 3 beschrieben. Daran sieht man das seither immer noch weiter versucht wurde die Sprache weiterzuentwickeln. Seit der Version 3 kamen einige neuen Features dazu, wie z.B. die Anweisungen *suspend* oder *pause*, aber auch andere neue Möglichkeiten. Soll heißen z.B. kann ein ESTEREL Programm nun auch in Hardware Spezifikationen übersetzt werden (vgl. Gamatié 2010, S. 30). Zur Entwicklung von ESTEREL Programmen existiert auch eine IDE (*integrated development environment*): ESTEREL Studio.

# 6 Fazit

Mithilfe der Programmiersprache ESTEREL ist es möglich *reaktive Systeme* in einem extrem natürlichen Programmierstil zu verfassen. Zum einen durch das Separieren eines Programms in parallele Komponenten für bessere Modularität und aber auch durch das hinzufügen von Signalen zur Synchronisation. Ein ESTEREL Programm kann als Automatenspezifikation angesehen werden, diese kann grafisch mit verschiedensten Tools dargestellt und verifiziert werden. Des Weiteren kann es in verschiedene sequentielle Sprachen übersetzt und dann mit hoher Effizienz ausgeführt werden (vgl. Boussinot und Simone 1991, S. 1303). Vor allem mit der PLAIN ESTEREL Version ist es einfach und benutzerfreundlich möglich *reactive kernel* für *reaktive Systeme* zu erstellen. Die dann auch leicht in eine *Host-sprache* eingebettet werden können, z.B. in die Sprache C. Ein weiterer Vorteil ist die Übersetzung in Hardwarebeschreibungssprachen. ESTEREL wird nicht nur in der Forschung verwendet und weiterentwickelt sondern hat auch ein großen Absatzmarkt in der Industrie, wie z.B. Nuklear-, Luftfahrt-, Verteidigungs- und Zugfahrtindustrie.

# Literatur


Berry, Gérard (2014). *Homepage*. URL: `http://www-sop.inria.fr/members/Gerard.Berry/` (besucht am 14. 11. 2014).

Berry, Gérard und Georges Gonthier (1988). »The ESTEREL synchronous programming language: design, semantics, implementation*«. In: *Science of Computer Programming 19*. Elsevier.

Boussinot, Frédéric und Robert de Simone (1991). »The Esterel Language«. In: *Proceedings of the IEEE, Vol. 79, No 9.*

Esterel-Technologies (2014). *Esterel Technologies - History*. URL: `http://www.esterel-technologies.com/about-us/history/` (besucht am 14. 11. 2014).

Gamatié, Abdoulaye (2010). »Synchronous Programming: Overview«. In: *Designing Embedded Systems with the SIGNAL Programming Language*. Springer.

Scholar, Google (2014). *Google Scholar Esterel Paper cite count*. URL: `http://scholar.google.de/scholar?cites=7055867594863660841&as_sdt=2005&sciodt=0,5&hl=de` (besucht am 14. 11. 2014).