FSPL - Eine funktionale synchrone Programmiersprache

Vom Modell über Syntax und Semantik zu kleinen Beispielen

Michael Wittig, Marco Ziener

Abstract: FSPL ist eine funktionale synchrone Programmiersprache zur Programmierung von eingebetteten Echtzeitsystemen. Durch die funktionale Umsetzung des mathematischen Modells wird die Verifikation der entwickelten Echtzeitprogramme wesentlich erleichert, was der Korrektheit der Echtzeitprogramme zuträglich ist. Ziel dieser Arbeit ist es eine kurze Einführung in das mathematische Modell FSPLs zu bieten und auf dieser Grundlage in Syntax und Semantik einzuführen, so dass die Vorteile des Ansatzes in der Praxis deutlich werden.

Inhaltsverzeichnis

1 Einleitung

Die Entwicklung von eingebetteten Echtzeitsystemen wird durch die fortschreitende Minituarisierung immer relevanter. Gleichzeitig werden solche Systeme immer häufiger in Umgebungen eingesetzt, in welchen ein Softwarefehler von fataler Bedeutung sein kann. Es ist wünschenswert über ein Werkzeug zu verfügen, welches den Entwickler ausgehend von der Spezifikation des Programm über die Softwareentwicklung bis hin zur Verifikation der Applikation begleitet und unterstützt. Insbesondere ist eine automatische Verifikation des Programmes hinsichtlich der ursprünglichen Spezifikation von enormer praktischen Relevanz.

In der Praxis reduziert sich Programmierung von hardwarenahen System auf die Anwendung der Programmiersprache C, was an der Verfügbarkeit der Übersetzer von C zu den verschiedenen Zielarchitekturen liegt, sowie der Verfügbarkeit von erfahrenen C-Programmierern. Problematisch ist jedoch, dass die in der Informatik verbreitete Auffassung eines Programmes - als Definition einer Berechnungsvorschrift - den Zeitbegriff nicht fasst: Man folgt der Annahme, dass die Berechnung augenblicklich erfolgt und das Ergebnis sofort vorliegt. Diese Auffassung und die Verwendung der Programmiersprache C bei hardwarenaher Programmierung führen zu erschwerten Bedingungen bei der Programmierung von Echtzeitsystemen und ereignisbasierenden Systemen: Die Überprüfung der Korrektheit, die Einhaltung der funktionalen und nichtfunktionalen Anforderungen, erfordern einen hohen Aufwand, da man sich nicht den Mechanismen und Garantien höherer Programmiersprachen der aktuellen Generation bedienen kann. Vor allem das Fehlen eines Zeitbegriffes kompliziert die Situation: Die Dauer einer Funktion ist konkret von der Geschwindigkeit des Prozessors abhängig. Wechselt man die Architektur, so müssen alle bisher gegebenen Garantien erneut überprüft werden. Eine unbefriedigende Situation.

Die Functional Synchronous Programming Language[1], soll Konzepte höherer Progammiersprachen - daher statische Typsysteme, funktionale bzw. prozedurale Abstraktion, Kapselung von Abstraktion und Daten sowie Polymorphie - mit der Programmierung von hardwarenaher Entwicklung verbinden, so dass die obig beschriebenen Probleme gelöst werden. Problematisch ist jedoch die Verfügbarkeit: Während andere synchrone Programmiersprachen langgediegen sind und somit über Übersetzer verfügen, ist FSPL eine recht junge Sprache, welche mehr als konzeptionelle Idee existiert als dass sie einen benutzbaren Übersetzer hätte. Gleichwohl ist es problematisch, dass bis auf die formale Spezifikation, kein öffentlicher Code - Compiler oder Interpreter - existiert, sodass sich unsere Arbeit ausschließlich auf[2] stützt. Dies hat zur Folge, dass wir ohne es explizit angeben uns auf die dort enstandenen Definitionen und Beispiele beziehen. Aus diesem Grunde konzentrieren wir uns in dieser Arbeit auf das Beschreibungsmodell der Sprache und demonstrieren die Nützlichkeit und Eleganz an kleinen Beispielen.

[1] Im folgenden als FSPL bezeichnet

[2] Frick: Eine funktionale, synchrone Programmiersprache für eingebettete Echtzeitsysteme,Forschungszentrum Informatik Karlsruhe, 2006

2 Anforderungen an eine Echtzeitprogrammiersprache

Die Programmierung von Echtzeitsystemen stellt neue Herausforderung an die klassische Informatik: Echtzeitprogramme sollen in der Regel nicht terminieren und widersprechen somit dem klassischen Algorithmusbegriff. Die Ausgabe an verschiedenen diskreten Zeitpunkten bei fortschreitender Zeit stellt mit langer Laufzeit und Koppelung an andere Hardware neue Herausforderungen an die Verifizierbarkeit und Fehlertoleranz. Dabei muss man unterscheiden zwischen 3 verschiedenen Echtzeitbedingungen:

hart Wenn ein Überschreiten der Zeitgrenze für die Berechnung katastrophale Konsequenzen haben kann.

weich Wenn ein Überschreiten der Zeitgrenze für die Berechnung, nicht die Nützlichkeit der Berechnung beeinflusst.

fest Wenn ein Überschreiten der Zeitgrenze die Berechnung wertlos macht, da sich zum Beispiel Randparameter signifikant verändert haben.

Möchte man diese Echtzeitbedingungen innerhalb eines Programmes umsetzen, so kann man zwischen 3 verschiedenen Modellen der Echtzeit wählen. In der Realität möchte man sich mit einer Programmiersprache aller dieser Modelle bedienen können um die maximale Flexibilität bei der Modellierung zu erhalten.

Quasi-kontinuierliches Verhalten Man betrachtet das System als eine Funktion, welche Signale auf Signale abbildet. Der aktuelle Zustand eines Signals wird zu einem bestimmtem Clocksignal aus den aktuellen und vorherigen Zuständen anderer Signale bestimmt. Mittels Differenzengleichung kann man ein Signalflußgraphen erstellen und so abhängige Signale/Variablen ermitteln.

Reaktives Verhalten Änderungen im Zustand werden durch Ereignisse ausgelöst. Man betrachtet Prozesse als Menge diskreter Zustände und Transitionen zwischen den Zuständen. Das Verhalten eines Prozesses wird durch den Zustandsübergang modelliert, wobei innerhalb einer Transition auch gerechnet werden kann.

Sequentielles Verhalten Das Programm ist eine Folge von Aktionen. Jede Terminierung einer Aktion bewirkt den Start der nächsten Aktion. Zusätzlich führt man Kontrollstrukturen und Sprünge ein, so dass man komplexere Algorithmen modellieren kann.

Zusätzlich muss die Sprache sowohl die Domäne bedienen als auch Methoden des Softwareengineering vernünftig umsetzen. Auch müssen die bereits in der Einleitung 1 beschriebenen Anforderungen umgesetzt werden.

3 Sprachkonzeption

Im vorherigen Kapitel haben sich an die Sprache FSPL verschiedenste Anforderungen ergeben. Diese gilt es nun umzusetzen. FSPL verfügt zu diesem Zweck über ein semantisches Modell um quasi-kontinuierliches, reaktives und sequentielles Verhalten zu modellieren. Zu diesem Zwecke werden Prozesse, Signale und Variablen als Funktionen[3] $f : \mathbb{T} \mapsto \mathbb{D}$ betrachtet, wobei $\mathbb{D}$ eine beliebige semantische Domäne darstellt. Man kann mittels dieser Betrachtung Prozess, Signal und Variable als isomorph betrachten. Eine notwendige Unterscheidung muss nur zwischen Prozess, Signal, Variable und Ereignis gemacht werden. Prozesse können als Ausgabe keinerlei Ereignisse generieren, jedoch solche als Eingabe annehmen. Ein Ereignis kann zu jeweiligen Ticks des Systems auftreten. Ein Prozess kann unterteilt werden in verschiedene Phasen: Eine Phase beschreibt ein Prozessverhalten ab einem Zeitpunkt. Eine Transition ist ein Übergang zwischen verschiedenen Phasen eines Programmes. Es ist möglich die Transition rekursiv zu gestalten. Eine Aktion ist ein Paar aus einer Phase und einem Ereignis. Das Starten einer Aktion startet eine Phase. Beendet wird sie durch das Eintreten des Ereignisses. Mittels dieser Mechanismen ist es möglich alle 3 Verhaltensmodelle zu beschreiben.

4 Spezifikation & Semantik

4.1 Zeit

In der Regel wird Zeit als ein beliebig fein aufgelöstes Kontinuum verstanden. Diese Betrachtung stellt den Programmierer eines Echtzeitsystems vor große Probleme: Man muss diskretisieren und die Zeit mittels Ticks versehen um weiterhin eine Modellierung angeben zu können. In der Folge kann man nicht mehr genau einen Zeitpunkt für ein Ereignis angeben, da eine Unschärfe eingeführt wird. Stattdessen fällt ein Ereignis nun in ein durch den Tick des Systems spezifiziertes Fenster.

Definition 4.1 (Zeit):
Sei $\mathbb{T}$ ein Modell für die Zeit.

$$\begin{aligned}
\mathbf{time} &: (\mathbb{T} \mapsto \mathbb{R}_+) \\
\mathbf{step} &:= (\mathbb{R}_+) \\
\mathbf{tick} &: (\mathbb{T} \mapsto \mathbb{N}) \\
\mathbf{toTime} &: (\mathbb{N} \mapsto \mathbb{T}) \\
\mathbf{time}\ t &= \mathbf{step} * (\mathbf{tick}\ t) \\
\mathbf{toTime}(\mathbf{tick}\ t) &= t \\
\mathbf{tick}(\mathbf{toTime}\ n) &= n
\end{aligned}$$

[3] $\mathbb{T}$ stellt die Zeit dar.

Nun kann man $\mathbb{T} = \mathbb{N}$ wählen und erhält ein valides Modell.

Definition 4.2 (Rechnen mit Zeit):
Sei $t_1, t_2 \in \mathbb{T}$ und $n \in \mathbb{N}$:

$$\begin{aligned} t_1 + t_2 &:= \mathbf{toTime}(\mathbf{tick}\ t_1 + \mathbf{tick}\ t_2) \\ t_1 - t_2 &:= \mathbf{toTime}(max(0, \mathbf{tick}\ t_1 - \mathbf{tick}\ t_2)) \\ n * t_1 &:= \mathbf{toTime}(n * \mathbf{tick}\ t) \\ t\ n &:= \mathbf{toTime}((\mathbf{tick}\ t)\ div\ n) \end{aligned}$$

Die Vergleichsoperatoren für die Zeit ergeben sich durch das natürliche Verständnis. Aufgrund der vorherigen Betrachtungen gilt es nun den Zustandsraum/Zustandsmenge zu definieren.

Definition 4.3 (Zustandsraum/Zustandsmenge):
Der Zustandsmenge $\mathbb{Z}$ besteht aus Tupeln der Form (x, e), wobei x Prozesse/Signale/Variablen darstellt und e ein mögliches Ereignis ist.

Ein Programm kann aus Prozessen bestehen, auch diese müssen ausreichend beschrieben werden.

Definition 4.4 (Prozess):
Sei $\mathbb{P}$ die Menge aller Prozesse. Ein Prozess $p \in \mathbb{P}$ ist eine Abbildung der Zeit in den Zustandsraum $\mathbb{Z}$:

$$p : \mathbb{T} \mapsto \mathbb{Z}$$

Nun kann man die Sicht des Programmes auf seine Teilprozesse definieren:

Definition 4.5 (Zustand eines Prozesses aus Sicht des Programmes):
Sei $p \in \mathbb{P}$.

$$\mathbf{state}_{\mathbb{Z}} : \mathbb{P} \mapsto \mathbb{T} \mapsto \mathbb{Z}$$

Man kann nun durch den Ausdruck $\mathbf{state}_{\mathbb{Z}}\ p\ t$ den Zustand $z \in \mathbb{Z}$ des Prozesses p zum Zeitpunkt t bestimmen.

Auf dieser Basis ergibt sich der Vergleich von Prozessen:

Definition 4.6 (Vergleich von Prozessen):
Seien $p_1, p_2 \in \mathbb{P}$ und $t \in \mathbb{T}$.

$$\begin{aligned} p_1 &= p_2 \Leftrightarrow \mathbf{state}_{\mathbb{Z}}\ p_1\ t = \mathbf{state}_{\mathbb{Z}}\ p_2\ t \\ p_1 &\overset{t_0}{|=} p_2 \Leftrightarrow \exists t_o \in \mathbb{T} : (t \geq t_0 \mapsto \mathbf{state}_{\mathbb{Z}}\ p_1\ t = \mathbf{state}_{\mathbb{Z}}\ p_2\ t) \\ p_1 &\overset{t_0}{=|} p_2 \Leftrightarrow \exists t_o \in \mathbb{T} : (t < t_0 \mapsto \mathbf{state}_{\mathbb{Z}}\ p_1\ t = \mathbf{state}_{\mathbb{Z}}\ p_2\ t) \\ p_1 &\overset{t_0}{|}\overset{t_1}{=|} p_2 \Leftrightarrow \exists t_o, t_1 \in \mathbb{T} : (t_0 \leq t < t_1 \mapsto \mathbf{state}_{\mathbb{Z}}\ p_1\ t = \mathbf{state}_{\mathbb{Z}}\ p_2\ t) \end{aligned}$$

Definition 4.7 (Ereignis):
Ein Ereignis kann zu mehreren Zeitpunkten auftreten und wird daher durch die endliche oder unendliche Folge von Zeitpunkten definiert. Dabei haben sie Einfluss auf den Zustandsverlauf eines oder meherer Prozesse. Die Folge der Zeitpunkte muss monoton steigend sein. In der formalen Definition wird ein Ereignis e durch eine Funktion **happened** beschrieben, die das nächste Auftreten des Ereignisses nach einem gegebenen Zeitpunkt t_0 beschreibt.

$$\mathbf{happened} : \mathbb{E} \mapsto \mathbb{T} \mapsto \mathbb{T} \mapsto \mathbb{B}$$
$$t \leq t_0 \mapsto \neg(\mathbf{happened}\ e\ t_0\ t)$$
$$(\mathbf{happened}\ e\ t_0\ t) \wedge (t \leq t') \mapsto \mathbf{happened}\ e\ t_0\ t'$$

Definition 4.8 (Vergleich von Ereignissen):
Sei $e_1, e_2 \in \mathbb{E},\ \forall t_0, t \in \mathbb{T},\ \mathbf{never} \in \mathbb{E}$:

$$e_1 = e_2 \Leftrightarrow \mathbf{happened}\ e_1\ t_0\ t = \mathbf{happened}\ e_2\ t_0\ t$$
$$e_1 \leq e_2 \Leftrightarrow \mathbf{happened}\ e_2\ t_0\ t \Rightarrow \mathbf{happened}\ e_1\ t_0\ t$$
$$\mathbf{happened\ never}\ t\ t_0 = false$$

$\leq$ kann als "findet stets früher oder gleichzeitig statt"gelesen werden.

4.2 Ereignisgeneratoren

Definition 4.9 (Timeout):
Für eine Funktion $\mathbf{after} : \mathbb{T} \mapsto \mathbb{E}$:

$$\mathbf{happened}\ (\mathbf{after}\ \Delta t)\ t_0\ t = t \geq (t_0 + \Delta t)$$

Definition 4.10 (Takt):
Ein Takt ist definiert durch eine Periodendauer ΔT und eine Phasenverschiebung t_1.
Für eine Funktion $\mathbf{clock} : \mathbb{T} \to \mathbb{T}_+ \to \mathbb{E}$:

$$\mathbf{happened}\ (\mathbf{clock}\ t_1\ \Delta t) t_0\ t = \begin{cases} t \geq t_1 & , falls\ t_1 > t \\ \mathbf{happened}(\mathbf{clock}\ (t_1 + \Delta t)\ \Delta t)\ t_0\ t & , sonst \end{cases}$$

(**clock** $t_1 \Delta t$) wirft nach $t_1 + \Delta t$ ein Event.

Definition 4.11 (Trigger):
Die trigger-Funktion erzeugt ein Event, wenn der Zustand eines booleschen Prozesses von false auf true wechselt [4].

Für eine Funktion $\mathbf{trigger} : \mathbb{P} \to \mathbb{E}$:

$$\mathbf{happened}\ (\mathbf{trigger}\ p)\ t_0\ t = \begin{cases} false,\ falls\ t \leq t_0 \\ true,\ falls\ \mathbf{happened}\ (\mathbf{trigger}\ p)\ t_0 (t-1) \\ (\mathbf{state}_{\mathbb{B}}\ p\ t) \wedge \neg(\mathbf{state}_{\mathbb{B}}\ p\ (t-1)),\ sonst \end{cases}$$

[4] eine steigenende Flanke

Definition 4.12 (Ereignissequenz):
$e_0 + e_1$ bezeichnet das Ereignis e_1, das auf das Ereignis e_0 folgt.
Für eine Funktion $+ : \mathbb{E} \times \mathbb{E} \to \mathbb{E}$ in infix-Notation gilt:

$$\textbf{happened}\ (e_0 + e_1)\ t_0\ t = \begin{cases} false,\ falls\ \neg(\textbf{happened}\ e_0\ t_0\ t) \\ \textbf{happened}\ e_1\ t_1\ t,\ sonst \end{cases}$$

wobei $t_1 = min\{t \in \mathbb{T} \mid \textbf{happened}\ e_0\ t_0\ t\}$

Damit lässt sich die Ereigniswiederholung sehr einfach definieren:

Definition 4.13 (Ereigniswiederholung):

$$n * e = \underbrace{e + \cdots + e}_{n}$$

Definition 4.14 (Ereignisausschluss):
Für eine Funktion $- : \mathbb{E} \mapsto \mathbb{E} \mapsto \mathbb{E}$ in Infix-Notation gilt:

$$\textbf{happened}\ (e_0 - e_1)\ t_0\ t = \textbf{happened}\ e_0\ t_0\ t\ \wedge\ \neg\textbf{happened}\ e_1\ t_0\ t$$

Ein bewachtes Ereignis ist ein Ereignis, das nur Auftritt wenn die Bedingung [5] gleichzeitig wahr ist.

Definition 4.15 (bewachtes Ereignis):
Für eine Funktion $\textbf{guard} : \mathbb{P} \mapsto \mathbb{E} \mapsto \mathbb{E}$ gilt:

$$\textbf{happened}\ (\textbf{guard}\ p\ e)\ t_0\ t = \begin{cases} false,\ falls\ \neg(\textbf{happened}\ e_0\ t_0\ t) \\ true,\ falls\ \textbf{state}_{\mathbb{B}}\ p\ t_1 = true \\ \textbf{happened}\ (\textbf{guard}\ p\ e)\ t_1\ t & ,sonst \end{cases}$$

, wobei $t_1 = min\ \{t \in \mathbb{T} \mid \textbf{happened}\ e\ t_0\ t\}$

Definition 4.16 (bedingtes Ereignis):
Für eine Funktion $\textbf{ifElseEvent} : \mathbb{P} \mapsto \mathbb{E} \mapsto \mathbb{E} \mapsto \mathbb{E}$ gilt:

$$\textbf{happened}\ (\textbf{ifElseEvent}\ p\ e_0\ e_1)\ t_0 = \begin{cases} \textbf{happened}\ e_0\ t_0 & ,\ \text{falls}\ \textbf{state}\ p\ t_0 = true \\ \textbf{happened}\ e_1\ t_0 & ,\ \text{sonst} \end{cases}$$

Definition 4.17 (Watchdog-Timeout):
Für eine Funktion $\textbf{watchdog} : \mathbb{E} \mapsto \mathbb{T}_+ \mapsto \mathbb{E}$ gilt:

$$\textbf{happened}\ (\textbf{watchdog}\ e\ \Delta t)\ t_0\ t = \begin{cases} false,\ \text{falls}\ t < (t_0 + \Delta t) \\ true,\ \text{falls}\ \neg(\textbf{happened}\ e\ t_0\ (t_0 + \Delta t)) \\ \textbf{happened}\ (\textbf{watchdog}\ e\ \Delta t)\ t_1\ t,\ sonst \end{cases}$$

[5] hier: Boolscher Prozess

, wobei $t_1 = min\{t \in \mathbb{T} \mid \mathbf{happened}\ e\ t_0\ t\}$

Wenn also ein Ereignis e nicht nach $t_0 + \Delta t$ eintritt, so wird ein Timeout Ereigniss geworfen.

4.3 Signalflüsse

Definition 4.18 (Konstante):
Für eine Funktion $\mathbf{const}_X : X \mapsto \mathbb{P}_X$ zu jeder Zustandsmenge X gilt:

$$\mathbf{state}_X\ (\mathbf{const}_X k)\ t = k$$

Nachdem die Zeit als ADT eingeführt wurde, kann sie auch als Signal selber interpretiert werden. Dazu dient die Identitätsfunktion zur Zeit!

Definition 4.19 (Zeit als Signal):
Für ein Objekt $\mathbf{timeProcess} \in \mathbb{P}_{\mathbb{T}}$ mit $\mathbf{timeProcess} = id_{\mathbb{T}}$ gilt:

$$\mathbf{state}_{\mathbb{T}}\ \mathbf{timeProcess}\ t = t$$

4.3.1 Statische Systeme

Ein statisches System bildet zeitgleich Eingabewerte auf Ausgabewerte ab, unabhängig vom Zeitpunkt oder Werten zu anderen Zeitpunkten. Ein statisches System hat daher eine Systemfunktion $f : \mathbb{P}_X \mapsto \mathbb{P}_Y$ und ist charakterisiert durch eine Funktion $f' : X \mapsto Y$ mit:

$$\mathbf{state}_Y(f\ p)\ t = f'(\mathbf{state}_X\ p\ t),\ \forall p \in \mathbb{P}_X \forall t \in \mathbb{T}$$

f' wird damit auf die Signalebene angehoben. Die Verallgemeinerung dieses Funktionsliftings ist der Konstruktor **apply**.

Definition 4.20 (Funktionslifting):
Für eine Funktion $\mathbf{apply}_{X \mapsto Y} : (X \mapsto Y) \mapsto \mathbb{P}_X \mapsto \mathbb{P}_Y$ gilt:

$$\mathbf{state}_Y(\mathbf{apply}_{X \mapsto Y}\ f\ p)\ t = f\ (\mathbf{state}_X\ p\ t)$$

4.3.2 Dynamische Systeme

Definition 4.21 (Zeitverschiebung):
Für eine Funktion $\mathbf{delay}_X : X \mapsto \mathbb{P}_X \mapsto \mathbb{P}_X$ gilt:

$$\mathbf{state}_X(\mathbf{delay}_X\ x_0\ p)0 = x_0$$
$$\mathbf{state}_X(\mathbf{delay}_X\ x_0\ p)(t+1) = \mathbf{state}_X\ p\ t$$

Nach der Verschiebung beträgt der Wert des Signals den Wert von vorher. Für den Zeitpunkt 0 muss eine Startwert x_0 angegeben werden.

4.4 Reaktive Prozesse

4.4.1 Phasen und Transitionen

Definition 4.22 (Phase):

$$\mathbf{phase}_X : \mathbb{P}_X \mapsto Phase_X$$
$$\mathbf{valueInPhase}_{Y,X} : \mathbb{P}_Y \mapsto (Y \mapsto Phase_X) \mapsto Phase_X$$
$$\mathbf{switch}_X : Phase_X \mapsto Transition_X \mapsto Phase_X$$
$$\mathbf{start}_X : Phase_X \mapsto Time \mapsto \mathbb{P}_X$$
$$\mathbf{goto}_X : Phase_X \mapsto Continuation_X$$
$$\mathbf{wait}_X \in Continuation_X$$
$$\mathbf{wait}_X : \mathbb{E} \mapsto Continuation_X \mapsto Transition_X$$
$$\mathbf{start}_X\ (\mathbf{phase}_X\ p)\ t_0\ \overset{t_0}{|=}\ p$$
$$\mathbf{start}_X\ (\mathbf{valueInPhase}_{Y,X}\ p\ \varphi_p)\ t_0 \overset{t_0}{|=} \mathbf{start}_X(\varphi_p(state_Y\ p\ t_0))\ t_0$$
$$t \geq t_0 \Rightarrow \mathbf{state}_X(\mathbf{start}_X(\mathbf{switch}_X\ \varphi\ \mathbf{wait}_X\ e_0\ \psi_0, \ldots, \mathbf{wait}_X\ e_n\ \psi_n])\ t_0)\ t =$$
$$\begin{cases} \mathbf{state}_X(\mathbf{start}_X\ \varphi\ t_0)t & , falls \neg(\mathbf{happened}(e_0 \vee \cdots \vee e_n)t_0\ t) \\ \mathbf{state}_X(\mathbf{start}_X\ \varphi_k\ t_1)\ t & , sonst \end{cases}$$

mit

$$t_1 = min\{t \in \mathbb{T} \mid \mathbf{happened}\ (e_0 \vee \cdots \vee e_n)t_0\ t\}$$
$$x_1 = \mathbf{state}_X(\mathbf{start}_X\ \varphi\ t_0)\ t_1$$
$$k = min\{i \in 0 \ldots n \mid \mathbf{happened}\ e_i\ t_0\ t_1\}$$
$$\varphi_i = \begin{cases} \phi_i & , \text{ falls } \psi_i = \mathbf{goto}_X\ \phi_i \\ \mathbf{phase}_X(\mathbf{const}_X\ x_1)\ t & , \text{falls } \psi_i = \mathbf{wait}_X \end{cases}$$

Ein Prozess ist damit eine unendliche Folge von Phasen. Eine Phase beschreibt das Prozessverhalten in einem Zeitinterval. Dabei übergeht man mit einer Transition (also einem Event!) in eine nächste Phase, oder sie endet nie.

Der Konstruktor $\mathbf{phase}_X$ erzeugt eine atomare Phase aus einem gegebenen Prozess, unabhängig vom Startzeitpunkt.

$\mathbf{switch}_X$ erzeugt eine zusammengesetzte Phase aus einer Ausgangsphase und n konkurrierenden Transitionen.

Eine Transition wird durch ein Event und eine Fortsetzung definiert. Dabei kann eine Fortsetzung nur aus einem $\mathbf{wait}_X$, das in der Phase bleibt, oder einem $\mathbf{goto}_X$-Konstrukt zu einer Zielphase bestehen.

$\mathbf{start}_X$ definiert hingegen welches Prozessverhalten die Phase ab einem Startzeitpunkt hat. $\mathbf{start}_X\ \varphi\ 0$ überführt die Phase φ in einen Prozess und ist damit ein Prozesskonstruktor.

Definition 4.23 (ifElsePhase$_X$):
Für eine Funktion $\mathbf{ifElsePhase}_X : \mathbb{P}_{\mathbb{B}} \mapsto Phase_X \mapsto Phase_X$ und für jede Zustandsmange X gilt:

$$\mathbf{ifElsePhase}_X\ c\ \varphi_0\ \varphi_1 = \mathbf{valueInPhase}_X\ c \left(\lambda c_0 \in \mathbb{B}. \begin{cases} \varphi_0 & \text{, falls } c_0 = true \\ \varphi_1 & \text{, sonst} \end{cases} \right)$$

Mittels dieses Konstrukts kann man effektiv zwischen Phasen wechseln und somit Transitionen definieren.

Definition 4.24 (bedingter Phasenwechsel):

$$\mathbf{switch}_X\ \varphi[\mathbf{when}_X\ e\ (\mathbf{goto}_X(\mathbf{ifElsePhase}_X\ c\ \varphi_0\ \varphi_1))]$$

Definition 4.25 (Phasenparallelisierung):
Sei $\lambda i \in I.X_i$ eine Mengenabbildung die jedem Index i aus einer Indexmenge I eine Zustandsmenge X_i zuordnet und

$$\mathbf{orthogonalize}_{\Pi(\lambda i \in I.X_i)} : \Pi(\lambda i \in I.Phase_{X_i}) \mapsto Phase_{\Pi(\lambda i \in I.X_i)}$$
$$\mathbf{start}_X(\mathbf{orthogonalize}_{\Pi(\lambda i \in I.X_i)}(\lambda i \in I.\varphi_i))\ t_0 \overset{t_0}{\mid=} \mathbf{zip}_{\Pi(\lambda i \in I.X_i)}(\lambda i \in I.\mathbf{start}_X\ \varphi_i\ t_0)$$

Nebenläufige Phasen setzen sich damit zusammen zu einer neuen Phase. Der neue Zustandsraum setzt sich multiplikativ zusammen aus den der einzelnen Phasen.

Definition 4.26 (Parallelisierung von Aktionen):
Für Funktionen $\mathbf{parallelizeTerminating}_{\Pi(\lambda i \in I.X_i)} : \Pi(\lambda i \in I.Action_{X_i}) \mapsto Action_{\Pi(\lambda i \in I.X_i)}$

und **parallelizeWaiting** mit dem gleichen Funktionskopf gilt:

$$\begin{aligned}\mathbf{behaviour}_X(\mathbf{parallelizeTerminating}_{\Pi(\lambda i \in I.X_i)}(\lambda i \in I.a_i)) = \\ \mathbf{orthogonalize}_{\Pi(\lambda i \in I.X_i)}(\lambda i \in I.(\mathbf{behaviour}_{X_i}\ a_i)) \\ \mathbf{termination}_X(\mathbf{parallelizeTerminating}_{\Pi(\lambda i \in I.X_i)}(\lambda i \in I.a_i)) = \\ \bigvee(\lambda i \in I.(\mathbf{termination}_{X_i}\ a_i))\end{aligned}$$

$$\begin{aligned}\mathbf{behaviour}_X(\mathbf{parallelizeWaiting}_{\Pi(\lambda i \in I.X_i)}(\lambda i \in I.a_i)) = \\ \mathbf{orthogonalize}_{\Pi(\lambda i \in I.X_i)}(\lambda i \in I.(\mathbf{continue}_{X_i}\ a_i\ \mathbf{wait}_X)) \\ \mathbf{termination}_X(\mathbf{parallelizeWaiting}_{\Pi(\lambda i \in I.X_i)}(\lambda i \in I.a_i)) = \\ \bigwedge(\lambda i \in I.(\mathbf{termination}_{X_i}\ a_i))\end{aligned}$$

Definition 4.27 (lokale Variablen):
Für eine Funktion $\mathbf{variableInPhase}_{Y,X} : Phase_Y \mapsto (\mathbb{P}_Y \mapsto Phase_X)Phase_X$ gilt:

$$\mathbf{start}_X(\mathbf{variableInPhase}_{Y,X}\ p\ \varphi_p)\ t_0 \mid^{t_0} = \mathbf{start}_X\ (\varphi_p\ (\mathbf{start}_Y\ p\ t_0))t_0$$

Lokale Variablen bilden eine Art Unterordner Nebenläufigkeit. In einer Phase eines Prozesses wird temporär ein weiterer Unterprozess definiert. Dieser dient als Hilfsdefinition des Oberprozesses.

Definition 4.28 (Ereignislokale Auswertung):
Für eine Funktion $\mathbf{valueInEvent}_X : \mathbb{P}_X \mapsto (X \mapsto \mathbb{E}) \mapsto \mathbb{E}$ und $\mathbf{variableInEvent} : Phase_X \mapsto (\mathbb{P}_X \mapsto \mathbb{E}) \mapsto \mathbb{E}$ gilt:

$$\begin{aligned}\mathbf{happened}\ (\mathbf{valueInEvent}_X\ p\ e_p)t_0\ t = \mathbf{happened}(e_p(\mathbf{state}_X p\ t_0))\ t_0\ t \\ \mathbf{happened}\ (\mathbf{variableInEvent}_X\ \varphi\ e_\varphi)t_0\ t = \mathbf{happened}(e_\varphi(\mathbf{start}_X \varphi\ t_0))\ t_0\ t\end{aligned}$$

Dies ist die gleiche Idee wie zuvor, nur auf Events anstatt auf Phasen.

4.5 Sequentielle Prozesse

Definition 4.29 (Aktion $\mathbb{A}$):
Für Funktionen :

$$\begin{aligned}
\mathbf{behaviour}_X &: \mathbb{A}_X \mapsto Phase_X \\
\mathbf{termination}_X &: \mathbb{A}_X \mapsto \mathbb{E} \\
\mathbf{until}_X &: \mathbb{E} \mapsto Phase_X \mapsto \mathbb{A}_X \\
\mathbf{continue}_X &: \mathbb{A}_X \mapsto Continuation_X \mapsto Phase_X \\
\mathbf{loop}_X &: \mathbb{A}_X \mapsto Phase_X \\
\mathbf{sequence}_X &: \mathbb{A}_X \mapsto \mathbb{A}_X \mapsto \mathbb{A}_X \\
\mathbf{ifElse}_X &: \mathbb{P}_{\mathbb{B}}\mathbb{A}_X \mapsto \mathbb{A}_X \mapsto \mathbb{A}_X \\
\mathbf{repeatUntil}_X &: \mathbb{A}_X \mapsto \mathbb{P}_{\mathbb{B}} \mapsto \mathbb{A}_X \\
\mathbf{valueInAction}_X &: \mathbb{P}_Y \mapsto (Y \mapsto \mathbb{A}_X) \mapsto \mathbb{A}_X \\
\mathbf{variableInAction}_X &: Phase_Y \mapsto (\mathbb{P}_Y \mapsto \mathbb{A}_X) \mapsto \mathbb{A}_X
\end{aligned}$$

gilt:

$$\begin{aligned}
&\mathbf{behaviour}_X(\mathbf{until}_X\ e\ \varphi) = \varphi \\
&\mathbf{termination}_X(\mathbf{until}_X\ e\ \varphi) = e \\
&\mathbf{continue}_X\ a\ \psi = \mathbf{switch}_X(\mathbf{behaviour}_X a)[\mathbf{wait}_X(\mathbf{termination}_X\ a)\psi] \\
&\mathbf{loop}_X\ a = \mathbf{continue}_X\ a(\mathbf{goto}_X(\mathbf{loop}_X\ a)) \\
&\mathbf{behaviour}_X(\mathbf{sequence}_X\ a\ b) = (\mathbf{termination}_X\ a) + (\mathbf{termination}_X\ b) \\
&\mathbf{behaviour}_X(\mathbf{ifElse}_X\ c\ a\ b) = \mathbf{ifElsePhase}_X\ c\ (\mathbf{behaviour}_X\ a)(\mathbf{behaviour}_X\ b) \\
&\mathbf{termination}_X(\mathbf{ifElse}_X\ c\ a\ b) = \mathbf{ifElseEvent}\ c(\mathbf{termination}_X\ a)(\mathbf{termination}_X\ b) \\
&\mathbf{behaviour}_X(\mathbf{repeatUntil}_X\ a\ c) = \mathbf{loop}_X\ a \\
&\mathbf{termination}_X(\mathbf{repeatUntil}_X\ a\ c) = \mathbf{guard}\ c(\mathbf{termination}_X\ a) \\
&\mathbf{start}_X(\mathbf{behaviour}_X(\mathbf{valueInAction}_{Y,X}\ y\ a_y))\ t_0 \mid\overset{t_0}{=} \mathbf{start}_X(\mathbf{behaviour}_X(a_y(\mathbf{state}_Y\ y\ t_0)))t_0 \\
&\mathbf{happened}_X(\mathbf{termination}_X(\mathbf{valueInAction}_{Y,X}\ y\ a_y))\ t_0\ t = \\
&\mathbf{happened}_X(\mathbf{termination}_X(a_y(\mathbf{state}_Y\ y\ t_0)))t_0\ t \\
&\mathbf{start}_X(\mathbf{behaviour}_X(\mathbf{variableInAction}_{Y,X}\ \varphi\ a_\varphi))\ t_0 \mid\overset{t_0}{=} \\
&\mathbf{start}_X(\mathbf{behaviour}_X(a_\varphi(\mathbf{start}_Y\ \varphi y\ t_0)))t_0 \\
&\mathbf{happened}(\mathbf{termination}_X(\mathbf{variableInAction}_{Y,X}\ \varphi\ a_\varphi))\ t_0\ t = \\
&\mathbf{happened}(\mathbf{termination}_X(a_\varphi(\mathbf{start}_Y\ \varphi\ t_0)))t_0\ t
\end{aligned}$$

Until ist der neue Operator für die Menge der Aktionen. continue und loop führen beide aus der Menge wieder heraus. Die übrigen Operatoren verknüpfen Aktionen und erschließen

sich aus dem natürlichem Verständnis. Eine Aktion $\mathbf{until}_X\ e\ \varphi$ heißt atomar. Alle anders konstruierten Aktionen heißen zusammengesetzt.

Definition 4.30 (Vergleich von Aktionen):

$$a_1 = a_2 \Leftrightarrow (\mathbf{behaviour}_X\ a_1) = (\mathbf{behaviour}_X\ a_2)$$

Für die Sequenzierung gilt das Assoziativgesetz:

$$\mathbf{sequence}_X\ a_0(\mathbf{sequence}_X\ a_1\ a_2) = \mathbf{sequence}_X(\mathbf{sequence}_X\ a_0\ a_1)a_2$$

Definition 4.31 (Aktionswiederholung):

$$1 * a = a(n+1) * a = \mathbf{sequence}(n * a)\ a$$

4.6 Exception Handling

Definition 4.32 (Ausnahmebehandlung):
Für die Funktion $\mathbf{continueWithExceptions}_X : \mathbb{A}_X \mapsto Continuation_X \mapsto Transition^*_X \mapsto Phase_X$ gilt:

$$\mathbf{continueWithExceptions}_X\ a\ \psi[\tau_0, \ldots, \tau_{n-1}] =$$
$$\mathbf{switch}_X(\mathbf{behaviour}_X\ a)[\mathbf{wait}_X(\mathbf{termination}_X\ a)\psi, \tau_0, \ldots, \tau_{n-1}]$$

5 Die Programmiersprache FSPL

5.1 Architektur der Sprache

Die Programmiersprache FSPL teilt sich in ihrer Architektur in 2 Teile auf: die Objekt- und die Metasprache.

Die Objektsprache ist das Mittel um die Objekte der Anwenungsdomäne zu beschreiben, während die die Metasprache dazu dient mit den Definition in der Objektsprache umzugehen. Daher sie erlaubt es dem Programmierer Namen und Klassifikationen zu vergeben, so wie Relationen zwischen den Objekten der Domäne zu formulieren: Ausdrücke der Objektsprache stellen die primitiven Elemente, der Metasprache dar. Die für die Metasprache zugrunde liegende Objektsprache ist austauschbar, auch wenn durch sie einen bestimmten Modellbildungsansatz umgesetzt wird. Das Funktionale der Programmiersprache FSPL lässt sich in der Metasprache festmachen: Sie enthält λ-Kalkül, starkes Typsystem und ein an den funktionalen Programmiersprachenn orientiertes Modulsystem. Erst durch die Kombination von Objektsprache und Metasprache wird die Programmiersprache vollständig.

5.2 Datentypen in Objektsprache

Da FSPL eine recht spezielle Anwendungsdomäne hat, benötigt man nur eine hanvoll von primitiven Datentypen: Booleans, Zahlen, Characters und Zeichenketten.

Die boolschen Operationen ergeben sich analog zur Programmiersprache C. Eine Einschränkung gibt es bei *if* b *then* x *else* y-Anweisung: x, y müssen gleich typisiert sein.

Bei Zahlen kann man Bereiche angeben

```
m as Integer n U oder m#n U
o as Integer k U oder o#k S
```

Hierdurch wird m als eine n-stelliger unsigned Integer definiert; o hingegen als k-stelliger signed Integer. Will man zwischen kompatiblen Typen umwandeln, so muss man dies explizit angeben:

```
d as Type
```

So kann man beispielsweise aus einem Integer einen Real leicht erzeugen:

```
1#8u as Real -> m = 1.0
```

Um sich die Betrachtung von Charactern zu erleichtern sind diese durch ihren ASCII-Code definiert. Dadurch ergibt sich auch der Vergleich von Charactern durch die Ordnungsrelation ihrer ASCII-Codes.

Strings sind analog zu anderen funktionalen Programmiersprachen Arrays von Chararactern.

Allerdings reichen diese Begrifflichkeiten nicht aus, um das Problem der Echtzeitprogrammierung behandeln zu können: Man muss Zeit als einen Datentyp einführen.

Dabei gilt die im vorherigen Abschnitt beschriebene Semantik von Zeit. Die kleinste verfügbare Zeiteinheit muss festgelegt werden. Dies kann beliebig je nach Zielplattform und unterstützer Zeitgenauigkeit geschehen, da sich die Semantik abseits des kleinsten verfügbaren Ticks nicht ändert. Allerdings muss ein Zeitoperator eingefügt werden, welcher aus natürlichen Zahlen eine Zeitkonstante erzeugt. Man greift auf die in der Semantik beschriebene Definition zurueck und ergänzt diese um

$$\mathbf{step} := 0.001 \in \mathbb{R}_+$$
$$\mathbf{step}_{Double} := 0.001 \in \textbf{Double}$$

5.3 Komplexe Datentypen in der Objektsprache

Prozesse designiert man mit **Process** X, wobei X eine Zustandsmenge des Prozesses darstellt. Die in der Sematik verwendeteten Beschreibung eines Prozess als Sammlung von **state**s diente nur zur Modellierung. Prozesse sind erstklassige Objekte in FSPL, daher

können sie - wie man später sehen wird - als Möglichkeit der Parametrisierung und Interprozesskommunikation genutzt werden.

Die Ereignisse müssen ebenfalls eingeführt werden: Sie werden mit **Event** deklariert.

Dabei werden die semantischen Ereignisgeneratoren an die folgende Syntax in 1 gebunden.

Zusätzlich werden die typischen Arithmetikoperatoren überladen, so dass sie für die neuen Typ benutzt werden können(siehe 2).

Da man nun ereignisbasierendes Verhalten beschreiben kann, muss man - um den in der Einleitung beschriebenen Anforderungen genüge zu tun - auch die restlichen Verhaltensmodelle umsetzen. Eines davon ist das Verhalten bezüglich von Signalen.

Primitive Signale werden als
$mathbf const_X x$, wobei X die Zustandsmenge selbst darstellt. Nun kann man typische Operationen auf diesem Datentyp definieren.

Syntax	Semantik
never	**never**
after t	**after** t
clock s offset t	**clock** t s
trigger x	**trigger** x
if x **then** e **else** f	**ifElseEvent** x e f
no e **within** t	**watchdog** e t

Tabelle 1: Ereignisgeneratoren

Syntax	Semantik	Beschreibung
$\mathbf{const}\ k$	$\mathbf{const}_x\ k$	konstantes Signal
$\mathbf{time}$		Zeitsignal
$\mathbf{apply}\ f\ \mathbf{to}\ x$	$\mathbf{apply}_{X\mapsto Y}\ f\ x$	Lifting von Funktionen und Anwendung
$\mathbf{zip}\ x$		Produkttransformation
$\mathbf{unzip}\ x$		Produkttransformation
$\mathbf{previous}\ x\ \mathbf{initially}\ x_o$	$\mathbf{delay}_X\ x_0\ x$	Signalverschiebung um eine Zeiteinheit
f	f	signalwertige Funktion

Tabelle 3: Beschreibungsmittel fuer den Signalfluß

Als interessant sind hier die zip- und $unzip$-Operationen hervorzuheben: Sie verbinden Elemente der Objektsprache(Prozesse) mit den strukturierten Typen der Metasprache. Somit befinden sie sich im Grenzbereich beider Sprachen. Semantisch stellen sie nur eine Variante der in Sprachen wie Haskell vertretenden Funktionen zip und $unzip$ dar: Gegeben von n Eingabeströmen[6] liefert zip einen Eingabestrom wieder, welcher die vorherigen *vereint*. Wie der Name es bereits andeutet ist $unzip$ die inverse Funktion zu zip und stellt die entsprechende Funktionalität bereit.

Syntax	Erklärung
$e\|\|f$	Ereignis e oder Ereignis f
$e\&\&f$	Beliebige Reinfolge von e und f
$e+f$	Sequentielle Komposition
$e-f$	Ereignis e ohne dass Ereignis f
$x*e$	x mal das Ereignis e
$e\%x$	e unter der Bedinung x

Tabelle 2: Ereignisoperatoren

Um weitere Funktionalität zu gewinnen, überlädt man bereits vorhandene Operatoren so, dass sie in der Lage sind mit den Signalen umzugehen. Für viele der Operationen ergibt sich dies aus dem

[6]Daher Listen, Arrays oder Records

Verständis[7]; Einige Besondererheiten sollen aber nicht unerwähnt bleiben: Bei gemischten Ausdrücken, wird der jeweilge nichtsignalige Ausdruck auf die Signalebene geliftet. Hierdurch ist es möglich zum Beispiel ein Signal mit einem Schwellenwert zu vergleichen. Ebenfalls anders ist die Verwendung des $if - then - else$: Es wird zum Multiplexer, da er zu jedem Zeitpunkt nun - abhängig vom Wert des boolschen Ausdrucks - zwischen den 2 verschiedenen Signalen umstellen kann.

Nun gilt es das semantische Modell bezüglich Phasenübergangssystemen und reaktiven Prozessen zu modellieren. Analog zur bisherigen Modellierung exisistiert hierfür ein neuer Typ: **Phase X** für jeden Datentyp X. Es existiert eine Unterscheidung zwischen dem in der Programmiersprache verwendeten **start** und dem im semantischen Modell definierten **start**$_X$: Der Startzeitpunkt im Programm ist stets der Zeitpunkt 0.

Die Das Konzept der lokalen Variable beziehungsweise der lokalen Auswertung von Ausdrücken, wird auch verwendet wenn man statt mit einer Phase mit einem Ereignis arbeitet. In diesem Fall sind die relevanten Abschnitte im semantischen Modell **valueInEvent**$_X$ beziehungsweise **variableinEvent**$_X$, welche praktisch nur im Bezeichner unterscheiden.

Nun geht es daran sequentielle Prozesse in die Programmiersprache einzuführen - erneut auf Basis des vorher bestimmten semantischen Modells. Analog zum bisherigen Vorgehen wirdfür jeden Datentyp den Aktionstyp **Action X** eingeführt. Auf Basis dieses Typs werden alle weiteren Aktionen dann definiert.

Analog zum **orthogonalize** gibt es ein **parallelize** für den Aktionstyp. Erneut ist es mit den 3 möglichen iterierbaren Typen kompatibel. Mögliche Unterscheidung zwischen folgenden Aktionen: Endpunkt oder Möglichkeit, dass es weiter geht.

5.4 Deklarationen und Definition in der Metasprache

Die Art und Weise wie man Bezeichner in FSPL definiert, erinniert an funktionale Programmiersprachen mit einem kleinen Schuß C:

```
1 value id : T [= x]
```

Diese Aussage vereinbart den Bezeichner id vom Typ T. In eckigen Klammern ist der optionale Teil angegeben: Mittels dieses Teiles kann man id den Wert x zuweisen. Da FSPL statisch und stark typisiert ist, kann man den Typ des Ausdrucks x ermitteln und er wird nur dann an id gebunden, wenn er mit dem Typ von id kompatibel ist. Um die Zuweisung zu vereinfachen, gleichzeitig die Umsetzung des Domänenmodels semantisch zu unterstützen und die Dokumentation zu vereinfachen bietet FSPL für die Deklaration der grundlegenden Typen eine Vereinfachung[8] an:

```
1 <process|signal|variable|event|phase|action> id:T[=x]
```

Statt zum Beispiel die vollständige Deklaration für einen Prozess:

[7] Arithmetische Operationen etc. werden einfach per *apply* auf den neuen Wertebereich geliftet.
[8] Eines aus den spitzen Klammern muss gewählt werden

Syntax	Semantik	Erklärung
keep x	**phase**$_X$ x	atomare Phase
do ψ **when** e_o **then** ... $\vdots$ **when** e_n **then** ...	**switch**$_X$ ψ[**when**$_x$ $e_o(\dots)$ $\vdots$ **when**$_x$ $e_n(\dots)$]	Transitionen ausgehend von einem Zustand
when e **then** ψ	**when**$_X$ $e(\mathbf{goto}_X \psi)$	Transition zu einer Phase. Alternativ kann man das $\mathbf{goto}_X \dots$ auch durch $\mathbf{wait}_X$ ersetzen: In diesem Fall ist es eine Transition in einen Endzustand.
local **value** $x_o : X := x$ **in** ψ	**valueinPhase**$_{X,Y}$ $x(\lambda x_0 \in X.\psi)$	Signalauswertung zu Phasenbeginn
local **variable** $x_o : X := \psi x$ **in** ψ	**variableinPhase**$_{X,Y}$ $\psi x(\lambda x_0 \in X.\psi)$	Signalauswertung zu Phasenbeginn
if x **then** $\psi_0 else \psi_1$	**ifElsePhase**$_X$ ψ_0 ψ_1	Konditionaler Ausdruck auf Phasenbasis
orthogonalize ψ	**orthogonalize**$_X$ ψ	Phasenparallelisierung. Wir gehen nicht auf die verschiedene Semantik auf Listen, Tupel und Records von Phasen ein. Es ist anzumerken, da alle aus Sicht von Duck-Typing verhältnismäßig isomorph zueinerander sind, die Unterschiede sich nur minimal in der Semantik wiederspiegeln.
start ψ	**start**$_X$ $\psi 0_{\mathbb{T}}$	Start eines Prozesses

Tabelle 4: Beschreibungsmittel reaktiver Prozesse

Syntax	Semantik	Erklärung
do ψ **until** e	$\mathbf{until}_X\ e\ \psi$	elementare Aktion
complete a **then** ψ	$\mathbf{continue}_X\ a\ (\mathbf{goto}_X\ \psi)$	Aktionsvervollständigung. Analog zu der Definition bei den reaktiven Modell, kann man hier das goto auch durch ein $\mathbf{wait}_X$ ersetzen, so dass man in einen Endzustand gehen kann.
complete a **then** ... **except** **when** e_o **then** ... ⋮ **when** e_m **then** ...	$\mathbf{continueWithExceptions}_X a \ldots$ [**when**$_X$ e_o (...), ⋮ **when**$_X$ e_n (...)]	Ausnahmebehandlung
loop a	$\mathbf{loop}_x\ a$	Endlosschleife
$a_o; a_1$	$\mathbf{sequence}_X a_0\ a_1$	Sequenzielle Komposition
if x**then** a_0**else** a_1	$\mathbf{ifElse}_x\ x\ a_0\ a_1$	Konditional für Aktionen
repeat a **until** x	$\mathbf{repeatUntil}_x\ a\ x$	Nachprüfende Schleife
$x * a$	$x \underset{\mathbf{Action}_X}{*} a$	x-fache Wiederholung

Tabelle 5: Beschreibungsmittel sequentieller Prozesse

```
value id:Process(T)=x
```

Die Schlüsselwörter **process**,**signal**,**variable** haben keine semantische Unterscheidung bei dieser Definition.

Recht unüberaschend - wenn man sich die Anforderungen and die Programmiersprache in Erinnerung ruft - ist die Existenz von Funktionsapplikation und Abstraktion.

Analog zu Sprachen wie Ocaml oder StandardML lassen sich lokale Definitionen anlegen:

```
let[rec]
   def1
   ....
   defn
in
   expr
```

Syntax	Semantik	Erklärung
$(\mathbf{value}\ id : T)x$	$\lambda id \in T.x$	Abstraktion
$f\ x$	$f\ x$	Funktionsapplikation

Tabelle 6: Funktionsabstraktion und Funktionsapplikation

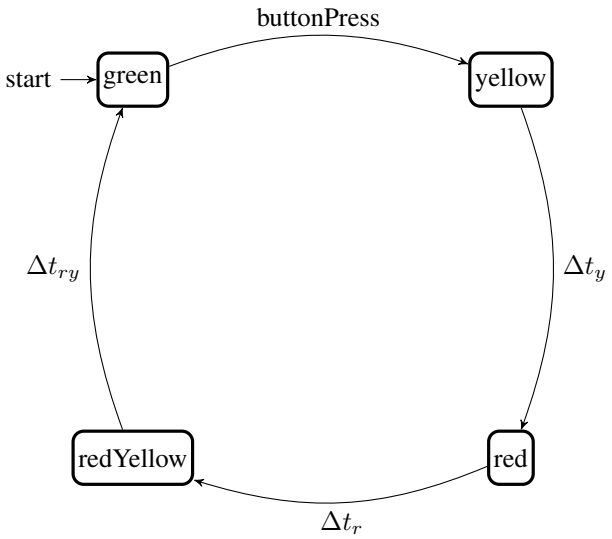
Hiermit schafft man mehrere parallele lokale Definitonen. Im Namensraum ist die Funktion selber allerdings nicht verfügbar. Möchte man also die Funktion rekursiv in sich selbst aufrufen, so muss statt *let letrec* verwendet werden, da nur so die Funktion als Symbol im eigenen Namensraum verfügbar ist. Das typische Beispiel einer rekursiven Funktion ist die bekannte Fakultätsfunktion $f : \mathbb{N}_+ \mapsto \mathbb{N}$ mit:

$$f(n) \begin{cases} n * f(n-1), n < 2 \\ 1, \text{ sonst} \end{cases}$$

Dies kann man leicht in eine **letrec** Beschreibung übertragen:

```
1 letrec
2  value fac(value: Integer16U):Integer16U =
3    if n == 0#16U then 1#16U else n*fac(n-1#16U)
4  in fac 12#16U
```

Auf dieser Basis kann man nun ein Transitionssystem in ein Phasensysten ummünzen. Das folgende Transitionsystem modelliert eine Ampelschaltung. Dabei stellt Δt_X mit $X \in \{ry, r, y\}$ ein Timeout für den Uebergang dar. *buttonPress* ist ein Event, dessen Implementierung wir nicht weiter ausführen.



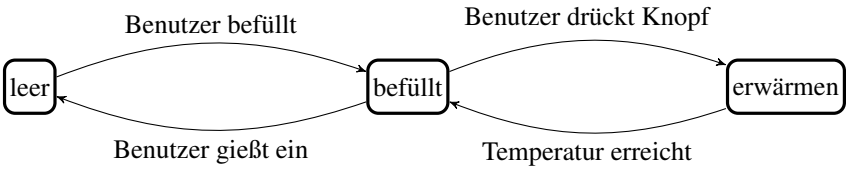
In der Implementierung nutzen wir die Triggerfunktion aus und modellieren ihn als boolwertigen Prozess: Springt seine Flanke von false nach true um, so erhalten wir das Ereig-

nis.

```
event buttonPressed = trigger(button)

letrec
   t_y = 2000
   t_r = 30000
   t_ry = 1000
   phase green:Real =
     do keep (const 0)
     when buttonPresed then yellow
   phase yellow:Real =
     do keep(const 1)
     when (after t_y) then red
   phase red:Real =
     do keep(const 2)
     when (after t_r) then red_yellow
   phase red_yellow:Real =
     do keep(const 2.5)
     when (after t_ry) then green
in
 start(green)
```

Wir werden nun stufenweise mit einen Wasserkocher die verschiedenen Konzepte praktisch demonstrieren. Dabei wird ebenfalls stufenweise die Abstraktion sinken. Ein Wasserkocher lässt sich vereinfacht durch folgenden endlichen Automaten ausdrücken:



Diese Darstellung lässt sich sehr leicht durch den *letrec*-Ausdruck in FSPL modellieren. Natürlich sind die Beschriftung der Transitionen und andere Details noch nicht ausreichend ausgefüllt; Dies stellt einen Makel dar, welchen wir später beseitigen werden, nachdem die relavanten Syntaxteile beschrieben wurden.

```
letrec
    phase leer:X =
       do wait
       when Benutzer_befuellt then befuellt

    phase befuellt:X    =
       do wait
       when Knopf_gedrueckt then erwaermen
       when Benutzer_eingegossen then leer

    phase erwaermen:X    =
        complete (heat) then befuellt
   in leer
```

Das verwendete X stellt den Typ der Phase dar. Die bisher im Text beschriebenen Typen sind nicht in der Lage die Zustände des Transitionssystem ausreichend zu beschreiben. Für diesen Fall bietet FSPL eine Möglichkeit neue Typen aus den Basistypen zu definieren und sich so für eine Problemstellung adequate Datenstrukturen zu schaffen.

FSPL trifft hierbei eine Unterscheidung zwischen Summentypen und Produkttypen: Summentypen benutzen Records um die Daten zu strukturieren und ihnen Bezeichner zuzuordnen; Bei einem Produkttyp handelt es sich um ein Tupel. Letztere lassen sich zu Potenztypen machen. Die detaillierten Unterschiede sind in der Dissertation nachzulesen.

Mit diesem neuem Wissen bewaffnet, ist Implementierung des Wasserkochers fortzusetzen.

Zunächst ist ein Typ für die Zustände zu definieren:

```
type State =
{
    full: Boolean,
    heated: Boolean
}
```

Intuitiverweise bilden diese beiden Parameter den Zustandsraum auf die im Transitionssystem beschriebenen Zustände ab. Jedoch ist ein Zustand nicht enthalten: Ein leerer, aber dennoch erwärmter Wasserkocher. In der Praxis ist die Belegung der Felder nicht einfach ein Boolean, sondern Prozesse, welche den Typ Boolean haben und auf der Hardwareplattform die entsprechenden Information sammeln. Der Wassersensor erhält als Eingabe ein realwertige Phase - welche das Signal des Wassersensors pollt und überprüft ob es einen Schwellenwert überschreitet - und liefert je nach Wasserstand entsprechend true und false wieder.

```
letrec
    phase WaterLevel (value w : WaterHeight,
                      value t:Real,
                      value b:Boolean) : Boolean =
    do
    keep const b
    when WaterLevelChanged then
       if w>t then WaterLevel(w,t,true) else WaterLevel(
           w,t,false)
    in
    start (WaterLevel(0.0,1.0, false))
```

Diese Implementierung abstrahiert von der tatsächlichen Hardware. Im Realfall müsste man die Spezifikation der Platine kennen und wie man sie anspricht. Dem Beispiel würde das jedoch mit unwesentlichen Details den didaktischen Wert nehmen. Aus diesem Grund ist w ein WaterHeight und wir benutzen es wie ein Realwert. Praktisch wäre es ein Prozess, welcher Werte des Wassersensors samplen tut. Der Heatsensor weist eine ähnliche Implementierung auf. Praktisch fehlen nun die Events, welche die Transitionen im Transitionssystem auslösen.

```
event WaterLevelChanged(value w : WaterLevelInterface) =
```

```
      let
      signal changed:Boolean = (w.change)
      in
      trigger changed

event ButtonPress(value b: ButtenInterface) =
      let signal buttonpress:Boolean = b.press
      in
      trigger buttonpress
```

Damit kann man nun leicht den Wasserkocher als gleichzeitiges System modellieren: Man nutzt **orthogonalize**. Die Leerstellen der vorherigen Beschreibung sind nun ausgefüllt. Die Funktion **heat** schreibt nun einfach einen Boolean in ein entsprechendes Hardwareregister, aus welchem der Controller der Heizspirale im Wasserkocher dann die entsprechenden Aktionen ableitet. *WaterLevelDropped* ist genau gegesätzlich zu *WaterLevelChanged* definiert, daher es wird getriggert, falls kein Wasser mehr im Wasserkocher ist.

```
letrec
    phase leer:State =
       do
         orthogonalize
         {
           full = WaterLevel,
           heated = HeatLevel,
           heatControl = HeatControl
         }
       when WaterLevelChanged then befuellt

    phase befuellt:State=
       do wait
       when buttonPress then erwaermen
       when (WaterLevelDropped) then leer

    phase erwaermen:State=
        do(
          repeat
          {
          heatControl.on = true
          }
          until heated;
          heatControl.on = false
        )
        when heatMax then befuellt
   in leer
```

5.5 Kapselung in Modulen und Paketen

FSPL unterstützt die Kapselung der Programmbestandteile nach Funktionalität mittels von Modulen. Analog zu Haskell und weiteren funktionalen Programmiersprachen, besitzt ein

Modul eine Schnittstelle, welche explizit die exportierten Funktionen des Moduls markiert und die Signaturen angibt. Beispielsweise:

```
interface {
    type t1
    ...
    type tn
    value id:T1
    ...
    value id:Tn
}
in
module    {
    type t1
    ...
    type tn
    value id:T1
    ...
    value id:Tn
}
```

Nach der Definition steht das Modul mit seinem Interface zum Import zur Verfügung:

```
import x in y
```

Hilfreich zur Anwendungsentwicklung ist es eine Bibliothek zur Verfügung zu haben. Dieses Konzept setzt FSPL mit dem Paketen um. Während Module für lokale Definitonen gedacht sind, kann man mit Paketen globale Definitonen beschreiben und somit größere Anwendungen noch stärker strukturieren und verwalten.

Die Syntax für Pakete ist dabei an die der Module angelehnt:

```
package packageID where
include packageID0
...
include packageID1
type t1
...
value id1:T1
```

Ein *Hauptprogramm* ist bei FSPL die letzte Definiton innerhalb eines Pakets.

6 Fazit

Alles in allem stellt FSPL ein interessante Sprache dar, welche aufgrund ihrer Expressivität und hierdurch vielseitigen Einsetzbarkeit gute Möglichkeiten der praktischen Anwendung bieten würde. Unglücklicherweise existert bis auf eine modellhafte[9] und nicht öffentliche zugängliche Implementierung keinerlei Compiler.

[9] Daher nicht vollständige...