SIGNAL

Florian Freudenberg

Student der Informatik
Freie Universität Berlin
Takustraße 9
14195 Berlin
florian.freudenberg@fu-berlin.de

Seminararbeit für das Seminar „Synchrone Programmiersprachen“
bei Prof. Dr. Elfriede Fehr und Lilit Hakobyan

Abstract: Diese Seminarbeit behandelt die grundlegenden Konzepte der synchronen Programmiersprache SIGNAL. Dabei wird zunächst definiert, was ein Signal in SINGAL ist und welche temporalen Eigenschaften für die Basisoperatoren gelten. Schließlich wird kurz die Konstruktion komplexerer Programme erläutert und beispielhaft gezeigt, wie diese mit dem Clock-Kalkül und Conditional-Depenedency-Graphen statisch analysiert werden können.

1 Einleitung

Diese Seminararbeit gibt einen Überblick über die Grundlagen von SIGNAL, einer synchronen, datenflussbasierten Programmiersprache zur Programmierung von Real-Time-Systemen. Der Überblick baut primär auf [GLB87] auf, sodass, wenn keine andere Quelle genannt wird, die grundlegenden Fakten darauf zurückzuführen sind.

Bei der Programmierung von Real-Time-Systemen ist es wichtig, eine Aussage über die garantierte maximale Ausführungszeit treffen zu können um zu überprüfen, ob das System die Anforderung erfüllt. SIGNAL besteht nicht nur aus einer deklarativen Sprache, sondern auch aus einem dazu gehörigen Kalkül, welches ermöglicht, statische Analysen über die logischen, zeitlichen Abhängigkeiten der Operationen eines Prozesses zu machen und damit eine Aussage über die Ausführungsdauer bzw. die Berechenbarkeit treffen zu können.

2 Annahmen über die Zeit

In SIGNAL wird Zeit auf einer logischen und nicht physikalischen Ebene betrachtet. Dies bedeutet, dass alle Signale eine temporal eindeutige Ordnung besitzen. Wenn es zwei Dateneingänge x und y gibt, könnte folgende Zeitabfolge auftreten:

$$x_1, y_2, (x_3, y_3), x_4, \dots$$

Dabei gibt der Index die temporale Ordnung vor, d. h. x_1 passiert vor x_3 und y_3 synchron zu x_3. Diese relative Ordnung, unabhängig der physikalischen Zeit, soll durch einen Mechanismus, der in [GLB87] nicht näher erläutert ist, erzeugt werden, sodass es im SIGNAL-Programm eine globale Ordnung aller Dateneingänge ins Programm gibt und dadurch die prinzipiell nichtdeterministische, asynchrone Kommunikation mit der Außenwelt für das Programm abstrahiert und nicht die garantierten Abläufe innerhalb des Programms beeinflusst.

Innerhalb des Programmes wird die temporale Abfolge der Operationen untereinander in Abhängigkeit der Sequenz der Kommunikationsereignisse betrachtet. Die einzelnen Operationen werden mit einer Ausführungszeit von null angenommen.

Die Operationsausführung erfolgt, sobald alle notwendigen Daten verfügbar sind. Die Verfügbarkeit der Daten und somit die Ausführungsreihenfolge der Operationen ist abhängig von den deterministischen Abhängigkeiten der anderen Operationen und deren Daten, sodass die Parallelität der Programmausführung nur von den Datenabhängigkeiten der Operationen beschränkt ist (für Details siehe [LGB91] Kapitel 4).

3 Signale und Uhren

Das grundlegende Objekt der Sprache SIGNAL ist ein Signal. Ein Signal besteht aus einer geordneten Sequenz von getypten Werten und einer dazugehörigen Uhr.

Die Uhr definiert den relativen Zeitpunkt an welchem die Werte des Signals verfügbar sind und dient als Hilfsmittel um temporale Zusammenhänge mehrerer Signale zum Ausdruck zu bringen.

SIGNAL besitzt eine polychrone Perspektive, d. h. dass unterschiedliche Signale zwar derselben Uhr folgen können, jedoch auch vollkommen unterschiedliche Uhren besitzen dürfen, wenn sie nicht synchron verfügbar sein müssen.

4 Basisoperationen

SIGNAL basiert auf einem kleinen Satz von sechs Basisoperationen, aus welchen dann komplexere Prozesse zusammengesetzt werden. Diese Operationen werden nun eingeführt.

4.1 Elementare Operationen

Zu den elementaren Operationen gehören die grundlegenden arithmetischen und booleschen Operationen. Sie werden sofort ausgeführt, wenn alle Eingabesignale verfügbar sind. Dies bedeutet, dass all diese Eingabesignale und das Ausgangssignal dieselbe Uhr besitzen und synchron sind.

Bspw. $v \coloneqq zv + 1$ ist in Gleichungsform: $\forall t: v_t = zv_t + 1$, wobei t den jeweiligen Zeitpunkt deklariert. Konstanten sind zu jedem Zeitpunkt verfügbar.

4.2 Delay-Operation

Die Delay-Operation ermöglicht es auf den Wert eines Signals zu einem vorherigen Zeitpunkt zuzugreifen. Die Delay-Operation wird wie folgt ausgedrückt ([LGB91]):

$$y \coloneqq x\ \$k$$

wobei k die Anzahl der Zeitpunkte vor dem aktuellen Zeitpunkt darstellt für welchen der Wert von Signal x als Ausgangswert gewählt werden soll. Als Formal wäre dies:

$$\forall t: (t - k > 0: y_t = x_{t-k}) \vee (t - k \leq 0: y_t = x_0)$$

In Abbildung 1 ist ein Beispiel aus [LGB91] zum Zeitverlauf des Delay-Operators zu sehen.

```
Z := Y $1
```

Y:	2	5	1	0	4	1	3	7	9	...
Z:	0	2	5	1	0	4	1	3	7	...

Abbildung 1: Zeitliche Wertsequenz eines Delay-Signals [LGB91]

4.3 Undersampling-Operator

Der Undersampling-Operator wird folgendermaßen ausgedrückt:

$$y \coloneqq x\ when\ b$$

Der Operator erzeugt genau dann ein Ausgangssignal y, wenn die Signale x und b verfügbar sind und das boolesche Signal b den Wert true besitzt. Das bedeutet, dass die Uhr von Signal y kleiner gleich den Uhren von x und b ist. In Abbildung 2 ist ein Beispiel Zeitverlauf zu sehen.

```
Y := X when B
```

X:	1	2	⊥	3	4	⊥	5	6	9	...
B:	*t*	*f*	*t*	*f*	*t*	*f*	⊥	*f*	*t*	...
Y:	1	⊥	⊥	⊥	4	⊥	⊥	⊥	9	...

Abbildung 2: Zeitliche Wertsequenz eines Undersampling-Signals [LGB91]

4.4 Event-Operator

Die Notation des Event-Operators ist:

$$y \coloneqq event\ c$$

Der Event-Operator liefert ein boolesches Ausgangssignal genau dann, wenn das Eingangssignal c verfügbar ist. Das Ausgangssignal hat immer den Wert true. Damit entspricht der Event-Operator der Uhr von c. Ein Beispielablauf wäre:

c:	⊥	1	⊥	2	4	5	⊥	2	1	4	5	⊥	…
v:	⊥	t	⊥	t	t	t	⊥	t	t	t	t	⊥	…

4.5 Merge-Operator

Der Merge-Operator wird wie folgt notiert:

$$y \coloneqq z\ default\ x$$

Der Merge-Operator erzeugt genau dann ein Ausgangssignal y, wenn die Signale x oder z verfügbar sind. Wenn das Signal z verfügbar ist, wird der Wert von z für das Signal y verwendet. Sollte z nicht verfügbar sein, dafür aber das Signal x, dann wird ein Ausgangssignal mit dem Wert von x erzeugt. Dabei wird z, wie beschrieben, priorisiert. Die Uhr von y ist damit größer gleich der Uhren von x und z. In Abbildung 3 ist ein Beispiel für die Verwendung des Merge-Operators zu sehen.

```
Y := U default V
```

U:	1	2	⊥	3	4	⊥	5	⊥	9	…
V:	⊥	⊥	3	4	10	8	9	2	⊥	…
Y:	1	2	3	3	4	8	5	2	9	…

Abbildung 3: Zeitliche Wertsequenz eines Merge-Signals [LGB91]

4.6 Synchro-Operator

Der letzte der grundlegenden Operatoren ist der Synchro-Operator. Dieser erzwingt eine explizite Synchronisation zweier Signale indem beiden dieselbe Uhr zugewiesen wird. Damit lassen sich mehrere Operationen koppeln, die eigentlich nicht synchron aufgrund von Datenabhängigkeiten sind. Die Notation ist:

$$synchro\ x, y$$

5 Operationskomposition

Die eingeführten Operationen sind nur dann wirklich nützlich, wenn sie miteinander kombiniert werden können. Die Komposition wird wie folgt ausgedrückt:

$$(|P\ 1|\ ...\ |P\ n|)$$

wobei jedes P eine grundlegende Operation oder selbst eine Komposition sein kann. Zwei Signale sind gleich, wenn sie denselben Bezeichner tragen.

Ein grundlegendes Beispiel für die Komposition aus [GLB87] ist:

$$(\ |v \coloneqq zv + 1|zv \coloneqq v\ \$1|\ )$$

Die Komposition besitzt kein Eingangssignal, aber zwei Ausgangssignale v und zv, wobei v den Zeitpunkt der Uhr der Komposition zählt, wenn angenommen wird, dass $v_0 = 1$ gilt. Zu jedem Zeitpunkt hängt der Wert von zv um eins dem Wert von v hinterher. Die Abbildung 4 zeigt ein Beispiel für eine grafische Notation für SIGNAL.

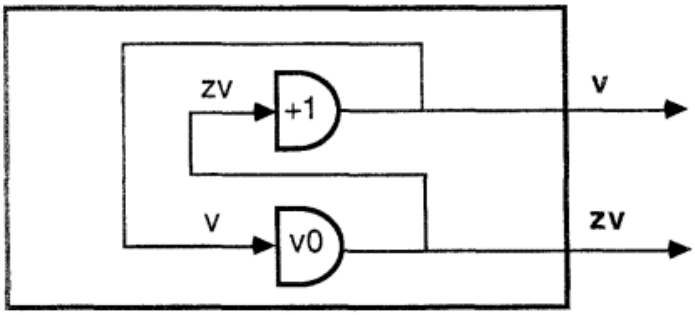

Abbildung 4: Grafische Notation der Komposition ($|v \coloneqq zv + 1|zv \coloneqq v\ \$1|$) [GLB87]

Ein nützlicher, komponierter Operator ist der Cell-Operator. Dieser ermöglicht es, den letzten Wert eines Signals über den Zeitpunkt des Auftretens hinaus verfügbar zu machen. Dabei bedeutet: $y \coloneqq x\ cell\ b$, dass das Signal y genau denn verfügbar ist, wenn x oder b verfügbar sind und dabei den letzten Wert vom Signal x trägt, wobei b ein boolesches Signal ist. Die Komposition dafür ist in [LGB91] zu finden[1]:

$$y \coloneqq x\ cell\ b$$

$$\Leftrightarrow$$

$$(|y \coloneqq x\ default\ (y\ \$1)|synchro\ a, y|a \coloneqq (event\ x)\ default\ (when\ b))$$

Dieser Operator stellt wiederum einen kleinen Prozess dar. Im Kapitel 6 wird definiert was genau ein Prozess ist.

[1] Anmerkung: Die Notation wurde an die Schreibweise aus [GLB87] angepasst.

6 Prozessmodell

Eine Komposition in Signal lässt sich wiederverwenden, wenn man ihre Schnittstelle definiert und damit einen Prozess definiert. Ein Prozess besteht aus einer Komposition und seiner Signalmaske. Dies Signalmaske gibt an, welche Signale lokale Signale sind und damit nach außen nicht sichtbar sein sollen. Für die Definition aus von Cell aus Kapitel 5 wäre das:

$$\big(|y \coloneqq x\ default\ (y\ \$1)|synchro\ a, y|a \coloneqq (event\ x)\ default\ (when\ b)\big)/a$$

Die Syntax ist abstrakt: $P/a_1, \dots, a_n$. Damit lässt sich jedoch kein Prozess ansprechen. Es fehlt eine Signatur, die Namen, Konstante und Ein- und Ausgabensingale festlegt. Die Syntax dafür ist nach [GLB87] folgende[2]:

$$Name\ \{\ Konstante\ ?\ Eingabeports\ !\ Ausgabeports\ \}\ =\ P\ /\ a_1\ , \dots , a_n$$

Wenn ein Prozess P verwendet werden soll, ist es unpraktisch, wenn man die Signalbezeichner des Prozesses verwenden muss. Daher gibt es noch folgenden Operatoren:

1. $P\ ?\ a_1 : b_1, \dots , a_n : b_n$ benennt die Eingabeports des Prozesses um.
2. $P\ !\ a_1 : b_1, \dots , a_n : b_n$ benennt die Ausgabeports des Prozesses um.
3. $P\ @\ a_1, \dots , a_n$ verbindet das Signal eines Ausgabeports mit dem gleichnamigen Eingabeport des Prozesses

Mithilfe dieses Wissens kann folgendes anschauliches Beispiel aus [GLB87] betrachtet werden. Der definierte Prozess stellt einen Zähler des booleschen Eingabesignals *iev* dar, welcher zum einen ein Zählerrücksetzsignal *hreset* und ein Schwellwertausgabesignal *oev* besitzt.

```
topmod { integer v0
        ? logical hreset,iev
        ! integer v; logical oev}
        = (     |synchro iev,zv
                |zv := v $1
                |v := (0 when hreset) default (zv+1)
                |oev := true when zv>=v0|)/zv
        where integer zv init v
end
```

Der Prozess topmod lässt sich dann zum Beispiel wie folgt aufrufen:

$$topmod(n-1)\ !\ oev : hreset\ @\ \text{hreset}$$

Abbildung 5 zeigt die grafische Darstellung des Aufrufs.

[2] Abgeleitet aus einem Beispiel.

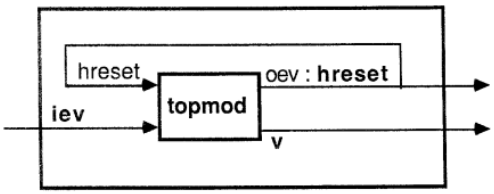

Abbildung 5: topmod-Aufruf mit mod n-1[GLB87]

7 Clock-Kalkül

Das Clock-Kalkül dient der Überprüfung eines Prozesses bzw. eines ganzen SIGNAL-Programmes auf temporale Korrektheit der Operatoren in Abhängigkeit der Signale. Die Idee hinter dem Kalkül ist, die Deklarationen auf Gleichungen zu übertragen und die möglichen Lösungen für die Signale des daraus resultierenden Gleichungssystems zu bestimmen um damit eine Aussage über die temporalen Bedingungen an die Signale zu treffen.

Dabei werden zwei Signaltypen unterschieden: boolesche Signale und allgemein Signale ohne Beachtung des Types. Die Unterscheidung ist notwendig bzw. sinnvoll, da der Undersampling-Operator explizit nur dann ein Signal erzeugt, wenn der boolesche Wert true ist. Mit Beachtung der booleschen Signale lassen sich so auch Kontradiktionen der logischen Operationen bei den Signalabhängigkeiten beachten.

Für ein Signal müssen allgemein zwei Zustände unterschieden werden: die Abwesenheit und die Anwesenheit des Signals. Diese können auf die Menge $\{0,1\}$ abgebildet werden. Für boolesche Signale muss zusätzlich zur Anwesenheit auch der Wert unterschieden werden, sodass eine Abbildung auf die Menge $\{0,1,2\}$ möglich ist. Dabei ist 0 die Abwesenheit, 1 die Anwesenheit mit Wert true und 2 die Anwesenheit mit dem Wert false.

Diese Abbildung ist im kommutativen Restklassenring $\mathbb{Z}/3\mathbb{Z}$ möglich. Gleichungen im $\mathbb{Z}/3\mathbb{Z}$ sind maximal vom Grad 2, da gilt ([GLB87]):

$$x^3 = x$$

Durch die Einschränkungen des $\mathbb{Z}/3\mathbb{Z}$ ist es effizient möglich das Gleichungssystem algorithmisch zu lösen ([LB86]). Eine formale Definition und Beweisführung zum Kalkül ist mitsamt einer formalen Definition von SIGNAL in [LB86] zu finden.

Nicht für jeden Befehl ist der Wert eines booleschen Signals wichtig, daher ist die Abbildung $x \to x^2$ für boolesche Signale sehr nützlich, da sie im $\mathbb{Z}/3\mathbb{Z}$ $0 \to 0, 1 \to 1, 2 \to 1$ abbildet.

Für die einzelnen Operationen, die im Clock-Kalkül unterschieden werden müssen, zeigt Abbildung 6 eine Tabelle mit den zugehörigen Gleichungsformen.

expressions	boolean result (values)	non boolean result (clocks)
y := *not* x	$y = -x$	$y^2 = x^2$
y := a *or* b	$y = ab\,(1 - (a + b + ab))$ $a^2 = b^2$	$y^2 = a^2 = b^2$
y := a *and* b	$y = ab\,(ab - (a + b + 1))$ $a^2 = b^2$	$y^2 = a^2 = b^2$
y := f(a_1, ..., a_n)	$y^2 = a_1{}^2 = \ldots = a_n{}^2$	$y^2 = a_1{}^2 = \ldots = a_n{}^2$
y := x $1	$y^2 = x^2$	$y^2 = x^2$
synchro a_1, ..., a_n	$a_1{}^2 = \ldots = a_n{}^2$	$a_1{}^2 = \ldots = a_n{}^2$
c := *event* a	$c = a^2$	
y := a *when* c	$y = a\,(-c - c^2)$	$y^2 = a^2\,(-c - c^2)$
y := a *default* b	$y = a + b - a^2b$	$y^2 = a^2 + b^2 - a^2b^2$

Abbildung 6: Tabelle mit Gleichungen für das Clock-Kalkül [GLB87]

Wenn sich das Gleichungssystem eines SIGNAL-Prozesses in $\mathbb{Z}/3\mathbb{Z}$ lösen lässt, sodass jede Variable, die für ein Eingangs- oder Ausgangssignal steht, nicht zwingend den Wert null annehmen muss, dann ist der Prozess temporal nicht blockiert [GLB87].

Ein grundlegendes Verständnis des Clock-Kalküls lässt sich anhand von Beispiele erlangen. Das erste Beispiel (aus [GLB87]) ist offensichtlich nicht korrekt ausführbar, da die Signale für die Addition sich ausschließen:

$$\left(|x \coloneqq a\ when\ (a > 0)|y \coloneqq a\ when\ \left(not(a > 0)\right)|z \coloneqq x + y|\right)$$

Unter Anwendung der Gleichungen aus Abbildung 6 erhält man (mit Ersetzung von (a>0) durch c zur Vereinfachung):

$$x^2 = a^2(-c - c^2)$$
$$y^2 = a^2(-(-c) - (-c)^2) = a^2(c - c^2)$$
$$z^2 = x^2 = y^2$$

Die Gleichungen lassen sich wie folgt umformen:

$$a^2(-c - c^2) = a^2(c - c^2)$$
$$-c - c^2 = c - c^2$$
$$-c = c$$

Die einzige Lösung für c ist $c = 0$ und laut Definition ist 0 die Abwesenheit des Signals. Daher gibt es für diesen Prozess keine Möglichkeit der Ausführung, da die Uhr von x, y, und z jeweils synchron mit der Uhr von c sind und damit das Signal z nie erzeugt werden kann. Dieses Bedingungsproblem ist daher von einem Compiler, der Clock-Kalkül verwendet, erkennbar.

Das folgende Beispiel besitzt keine temporalen Fehler:

$$(|x \coloneqq a\ default\ v0|z \coloneqq x + y|)$$

Wenn man sich dazu das Gleichungssystem ansieht:

$$x^2 = a^2 + v0^2 - a^2 v0^2$$
$$z^2 = x^2 = y^2$$

sieht man direkt, dass die Signale z, x und y alle verfügbar sein müssen, damit der Prozess berechenbar ist. Da v0 der Notation von Konstanten entspricht und diese immer verfügbar sind, kann geschlussfolgert werden, dass gilt:

$$x^2 = a^2 + 1 - a^2$$

Daher ist das Signal für x immer verfügbar, sodass das Signal von z nur abhängig vom Signal von y ist. Da keine weiteren Bedingungen eingehalten werden müssen, können die Signale von z und y die Werte 0 und 1 annehmen, wobei z genau dann verfügbar ist, wenn y verfügbar ist, und damit ist der Prozess temporal korrekt.

8 Conditional-Dependency-Graph

Die Gleichungen des Clock-Kalkül treffen nur eine Aussage über die temporalen Bedingungen zwischen Signalen. Die Abhängigkeiten zwischen den einzelnen Signalen werden dabei außer Acht gelassen. Für die Betrachtung der Abhängigkeiten zwischen den Signalen bietet sich ein Graph an, welcher Conditional-Dependency-Graph genannt wird.

Die Knoten dieses Graphen sind die Uhren und Signale und die Kanten des Graphen werden mit den Gleichungen aus dem Clock-Kalkül versehen, welche die Zusammenhänge zwischen den einzelnen Uhren beschreiben. Dadurch ist es möglich in dem Graphen genau nach den Kreisen zu suchen, die zyklische Abhängigkeiten zwischen den Uhren beschreiben und damit eine nichterfüllbare Signalkette darstellen.

In [GLB87] wird das Verfahren zur Detektion derartiger Kurzschlüsse in der Signalkette wie folgt beschrieben:

Wenn ein Kreis in dem Conditional-Dependency-Graph gefunden wird, muss das Produkt der Uhren-Ausdrücke aller Kanten des Kreises gebildet werden. Wenn das Produkt aller Kanten nicht null ist, dann stellt der Kreis einen Kurzschluss dar, ansonsten wird der Graph als „circuit-free“ bezeichnet und der Prozess ist berechenbar.

Wenn das Clock-Kalkül ergibt, dass der Prozess temporal korrekt ist und außerdem der Conditional-Dependency-Graph keine Kurzschlüsse enthält, kann auf Basis des Graphen und der Clock-Kalkül-Gleichungen die Codegenerierung beginnen [GLB87].

Das folgende Beispiel aus [LGB91] illustriert die Notwendigkeit des Conditional-Depenedency-Graphen:

$$(|x \coloneqq \sin\{y\} + b|y \coloneqq a\ default\ x|)$$

Die Clock-Kalkül-Gleichungen dazu sind:

$$x^2 = y^2 = b^2$$
$$x^2 = a^2 + b^2 + a^2b^2$$

Daraus folgt, dass für die gemeinsame Uhr h gilt:

$$h = x^2 = b^2 = y^2 = a^2 + (1 - a^2)b^2$$

Aus dem Gleichungssystem heraus ist keine temporale Verletzung erkennbar. Wenn man jedoch den Graphen (siehe Abbildung 7) betrachtet, existiert ein Kreis dessen Produkt für $a = 0$ einen Kurzschluss bildet und damit ist der Prozess nicht korrekt.

Wenn gilt, dass: $h = a^2$, dann wäre der Prozess äquivalent zu:

$$(\ |y \coloneqq a|x \coloneqq \sin\{a\} + b|\ )$$

und ansonsten ist er nicht berechenbar. Da der Programmierer nicht letzteren Prozess deklariert hat, liegt offensichtlich ein Fehler vor.

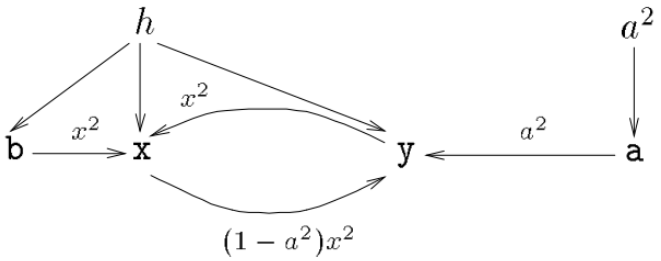

Abbildung 7: Conditional-Dependency-Graph zu $(|x \coloneqq \sin\{y\} + b|y \coloneqq a\ default\ x|)$ [LGB91]

Der Conditional-Dependency-Graph zu $(|x \coloneqq a\ default\ v0|z \coloneqq x + y|)$ ist in Abbildung 8 zu sehen und offensichtlich kreisfrei und damit auch „curcuit-free“, sodass der Prozess korrekt ist.

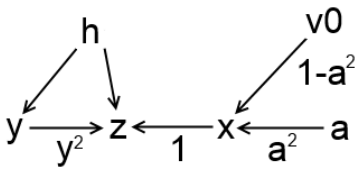

Abbildung 8: Conditional-Dependency-Graph zu($|x \coloneqq a\ default\ v0|z \coloneqq x + y|$)

9 Fazit

In dieser Seminararbeit wurden einige Grundlagen von der synchronen Programmiersprache SIGNAL präsentiert. Beginnend mit den Basisoperatoren wurden über die Operationskomposition bis hin zum Prozessmodell die notwendigen Notationen zum Entwickeln von einem Programm in SIGNAL erläutert und anschließend zwei Hilfsmittel zur statischen Analyse der Programmeigenschaften vorgestellt, mit denen sich nicht berechenbare Prozesse ausschließen lassen.

Für die formale Spezifikation und Beweisführung der Methoden für SIGNAL wird auf [LB86] verwiesen. In [LGB91] können etwas umfangreichere Prozesse mit praktischen Anwendungsfällen gefunden werden und in Form von Polychrony[3] stehen auch nutzbare Werkzeuge, wie z. B. ein Compiler und grafische Prozessdarstellungen, für SIGNAL zur Verfügung.

Literaturverzeichnis

[GLB87] Gautier, T.; Le Guernic, P.; Besnard, L.: SIGNAL: A declarative language for synchronous programming of real-time systems; In Functional programming languages and computer architecture; Springer Berlin/Heidelberg, 1987; S. 257-277.

[LB86] Le Guernic, P.; Benveniste, A.: Real-Time, Synchronous, Data-Flow Programming: the Language SIGNAL and its Mathematical Semantics; In INRIA, Rennes, Research Report n° 533, 1986.

[LGB91] Le Guernic, P; Gautier, T.; Le Borgne, M.; Le Maire, C.: Programming real time applications with SIGNAL; In Proceedings of the IEEE 79.9, 1991; S. 1321-1336.

[3] http://www.irisa.fr/espresso/Polychrony/, zuletzt abgerufen am 10.02.2013