Einbinden imperativer synchroner Programmiersprachen in interaktive Theorembeweiser

Jan Sydow Christian Tietz

1 Einleitung

Diese Arbeit gibt die Arbeit von Schneider „Embedding Imperative Synchronous Languages in Interactive Theorem Provers“ wieder[Sch01]. Alle Informationen aus dieser Arbeit stammen aus der Arbeit von Schneider sofern nicht anders angegeben.

Synchrone Programmiersprachen haben mit der zunehmenden Verbreitung von eingebetteten Systemen an Bedeutung gewonnen.

Eingebettete Systeme haben Geräte billiger, kleiner, flexibler gemacht und ihren Funktionsumfang erhöht. Sie werden mittlerweile in vielen Bereichen wie Luftfahrt, Automobilindustrie, Mobiltelefonen, Unterhaltungselektronik, Industrieanlagen und medizinischen Apparaten eingesetzt.

Ein Problem beim Design eingebetteter Systeme ist, die Vielzahl an Disziplinen, die am Design mitwirken. So ist Software Engineering, Hardware Design, Control Theory und mechanisches und elektrisches Ingenieurwesen gefragt beim Design eingebetteter Systeme.

Eingebettete Systeme haben in der Regel mehrere Komponenten mit eigenen Prozessoren die untereinander und mit der Umwelt kommunizieren. Häufig sind sie auch reaktive Systeme die nur auf Änderungen in der Umgebung reagieren. Ein weiteres Merkmal ist, dass eingebettete Systeme Echtzeitsysteme sein können, mit einer garantierten Worst-Case-Reaktionszeit. Da sie auch in sicherheitsrelevanten Gebieten eingesetzt werden, spielt formale Verifikation auch eine Rolle beim Design eingebetteter Systeme.

All diese Faktoren haben zur Entwicklung von synchronen Programmiersprachen geführt. Synchrone Programmiersprachen führen den Entwurf von Hard- und Software zusammen und sind für reaktive und Echtzeitsysteme mit unabhängigen Komponenten ausgelegt. Aus Programmen in synchronen Programmiersprachen lassen sich nämlich sowohl Schaltpläne für die Hardware als auch Software generieren.

Es gibt imperative Programmiersprachen wie Esterel oder Quartz, wobei Quartz eine Esterel-Variante ist, die hier betrachtet wird. Weiterhin gibt es deklarative Dataflow-Sprachen wie Lustre und grafische Programmiersprachen wie StateChart oder SyncCharts Varianten.

Das Prinzip von synchronen Programmiersprachen ist die perfekte Synchronie. Der Programmablauf wird an `pause`-Statements synchronisiert. Programme laufen also im lockstep. Wenn ein Programmfluss ein `pause`-Statement erreicht, so wartet er dort, bis al-

le anderen Programmflüsse ebenfalls ein `pause`-Statement erreicht haben. Konzeptuell sind `pause`-Statements, die einzigen Statements, die Zeit zur Ausführung benötigen. Sie bilden sogenannte Makrosteps. Alle anderen Statements verbrauchen 0 Zeiteinheiten, sie werden augenblicklich ausgeführt. Diese Statements laufen in Mikrosteps. Konzeptuell befindet sich der Programmfluss also zu jedem messbaren Zeitpunkt an `pause`-Statements. Man kann den Kontrollfluss von Programmen in synchronen Programmiersprachen damit als endlichen Automaten implementieren, wobei eine Menge von aktiven pause-Statements einen Zustand des Automaten darstellen. Da Programme auch parallel laufen können, können mehrere `pause`-Statements gleichzeitig aktiv sein. Deswegen ist ein Zustand durch eine Menge von aktiven `pause`-Statements charakterisiert Die

In der hier vorgestellten Arbeit „Embedding Imperative Synchronous Languages in Interactive Theorem Provers“ wird eine Möglichkeit vorgestellt imperative synchrone Programmiersprachen in Theorembeweiser wie HOL zu integrieren. Mithilfe des Theorembeweisers lassen sich dann Eigenschaften von Programmen oder der Programmiersprache an sich nachweisen.

Die Darstellung von Programmen als endliche Automaten eignet sich auch für symbolisches Model Checking. Die vorgestellte Methode zur Einbettung von Quartz in einen Theorembeweiser lässt sich auch für Model Checking verwenden.

Der von Schneider benutzte interaktive Theorembeweiser HOL wurde von Mike Gordon an der Cambridge University entwickelt [Mel88]. HOL wurde ursprünglich zur Hardware Verifikation entworfen. HOL implementiert die Logik in der funktionalen Programmiersprache ML. Propositions und Theoreme werden mithilfe der abstrakten Datentypen von ML dargestellt. Die interaktive Beweisführung wird über den Aufruf von Funktionen auf den Daten bewerkstelligt. Durch die starte Typisierung von ML und der Darstellung von Theoremen als abstrakte Datentypen ist sichergestellt, dass nur gültige logische Schlussfolgerungen möglich sind.

2 Quartz

Die synchrone Programmiersprache Quartz wurde für den Entwurf reaktiver Systeme entwickelt. Sie ist Teil des averest Frameworks[1], dass an der Universität Kaiserslautern entwickelt wurde.

Im Folgenden wird der für das Embedden in Theorembeweiser Teil der Syntax und Semantik der Programmiersprache zusammengefasst:

Da für das Einbetten der Programmiersprache in Theorembeweisern hauptsächlich die Kontrollstrukturen eine Rolle spielen, werden keine speziellen Datentypen betrachtet. Jedem Ausdruck wird also nur ein generischer Typ α zugeordnet.

Die Zeit wird als natürliche Zahl modelliert, denn Zeit vergeht ja konzeptuell nur an `pause`-Statements. Zu jedem Zeitpunkt gibt es eine Menge von `pause`-Statements, die gerade aktiv sind

[1] `http://www.averest.org`

Der Typ von Ausdrücken ist durch diese Zeitmodellierung $\mathbb{N} \to \alpha$. Es werden zwei Arten von Variablen unterschieden:

- Event-Variablen haben die Semantik $\mathbb{N} \to \mathbb{B}$, sie sind flüchtig, d. h., sie werden zum nächsten Zeitpunkt auf *false* gesetzt. Event-Variablen werden mithilfe des `emit`-Statements auf true gesetzt.
- State-Variablen sind nicht-flüchtig und können von einem beliebigen Typ α sein. Die Semantik ist hier $\mathbb{N} \to \alpha$. Sie verhalten sich so wie Variablen in anderen Programmiersprachen wie C oder Java.

Zusätzlich gibt es noch Input-Variablen. Sowohl Event- als auch state-Variablen können Input-Variablen sein. Input-Variablen werden von der Umgebung beeinflusst, z. B. von anderen Programmmodulen oder Hardware.

Die Menge der Quartz-Statements ist die kleinste Menge, die folgende Regeln erfüllt. Hierbei sind `S1`, `S2`, `S3` ebenfalls Quartz-Statements, `x` ist eine event-Variable, `y` ist eine state-Variable, `l` ist eine location-Variable und σ ist ein Boolscher Ausdruck.

- `nothing`
- `emit x`, `emit delayed x`
- `y := r`, `y := delayed r`
- `l: pause`
- `if` σ `then S1 else S2 end`
- `S1; S2`
- `S1 || S2`
- `S1 ||| s2`
- `choose S1 [] S2`
- `while` p `do S end`
- `suspend S when` σ
- `weak suspend S when` σ
- `abort S when` σ
- `weak abort S when` σ
- `local x in S end`
- `local y :` α `in S end`
- `now` σ

- `during S holds` σ
- `run m(...)`

Es folgt eine informelle Semantik der Statements:

`nothing` tut nichts. `nothing` verändert weder Daten, noch verbraucht es Zeit.

`emit x` setzt die Event-Variable `x` sofort auf true. `emit delayed x` setzt die Event-Variable für den nächsten Zeitpunkt auf true.

`y := r` setzt `y` sofort auf den Wert des Ausdrucks `r`. `y := delayed r` setzt `y` zum nächsten Zeitpunkt auf den Wert des Ausdrucks `r`. Dabei wird `r` im aktuellen Zeitschritt ausgewertet. Lediglich die Zuweisung wird auf den nächsten Zeitschritt verschoben.

`l: pause` verbraucht eine Zeiteinheit. Dies ist der einzige Befehl der Zeit verbraucht. Die Ausführung des Statements wird erst dann fortgeführt, wenn alle Programmflüsse ein `pause`-Statement erreicht haben. Deswegen muss beim Programmieren darauf geachtet werden, dass in einem Programmfluss regelmäßig `pause` aufgerufen wird und vor allem keine langen Schleifen ohne pause-Statement laufen. Die label `l` müssen programmweit eindeutig sein.

`if` σ `then S1 else S2 end` ist die klassische Verzweigung. Wenn der Ausdruck σ zu true auswertet, wird sofort `S1` ausgeführt, ansonsten wird sofort `S2` ausgeführt.

`S1; S2` ist die Hintereinanderausführung von `S1` und `S2`. `S1` wird sofort ausgeführt, sobald `S1` terminiert wird sofort `S2` ausgeführt. `S1` kann natürlich `pause`-statements enthalten. Deswegen muss `S2` nicht im selben Zeitschritt starten in dem `S1; S2` gestartet wurde.

Die Anweisungen `S1 || S2` und `S1 ||| S2` führen die Anweisungen `S1` und `S2` parallel aus. Dabei führt `S1 || S2` die Anweisungen synchron aus, d. h., beide Anweisungen werden jeden Zeitschritt ausgeführt. Bei der asynchronen Ausführung `S1 ||| S2` kann ein Programmfluss auch stehen bleiben. Zum Beispiel das Programm

```
(l1:pause; l2:pause) || (l3:pause)
```

hat als endlicher Automat die Zustände {`l1`, `l3`}und {`l2` }. Das Programm

```
(l1:pause; l2:pause) ||| (l3:pause)
```

hingegen hat die Zustände {`l1`, `l3`}, {`l1`}, {`l2`, `l3`}, {`l2`}und {`l3`}. Im folgenden wird diese Anweisung nicht betrachtet. Sie kann aber durch hinzufügen zweier weiterer Input-Eventvariablen mit den anderen Anweisungen simuliert werden.

`choose S1 [] S2` führt nichtdeterministisch entweder `S1` oder `S2` aus. Im folgenden wird diese Anweisung nicht betrachtet. Sie kann aber durch hinzufügen einer weiteren Input-Eventvariable mit den anderen Anweisungen simuliert werden.

`while` σ `do S end` ist die klassische while-Schleife.

Das Statement `suspend S when` σ und `weak suspend S when` σ wird nur ausgeführt, wenn der Kontrollfluss bereits in `S` ist und σ zum aktuellen Zeitpunkt zu falsch

auswertet. Bei der `weak`-Variante werden die Datenmanipulationen in `S` noch ausgeführt in dem Zeitschritt in dem σ wahr wird. Bei der „starken“ Variante müssen die Datenmanipulationen eventuell rückgängig gemacht werden.

Das Statement `abort S when` σ und `weak abort S when` σ bricht die Ausführung von `S` ab, wenn der Kontrollfluss bereits in `S` ist und σ gilt. Die `weak`-Variante hat die gleichen Eigenschaften wie das `weak` von `suspend S when` σ.

`local x in S end` und `local y :` α `in S end` führen lokale Event- bzw. Zustandsvarianlen in `S` ein.

`now` σ prüft, ob die Bedingung σ aktuell gilt, ansonsten bleibt der Kontrollfluss stecken. `during S holds` σ prüft zu jedem Zeitpunkt, in dem der Kontrollfluss in `S` ist, ob σ gilt.

Die Anweisung `run m(...)` führt das Modul `m` mit den in den Klammern angegebenen Parametern aus.

3 Embedden

Das zentrale Problem beim überführen einer Programmiersprache in die Logik eines Theorembeweisers sind Inkonsistenzen in den Axiomen. Treten Inkonsistenzen auf, so kann alles bewiesen werden. Die von Schneider benutzte Lösung zur Vermeidung von Inkonsistenzen ist die primitive Rekursion. Die Semantik wird mithilfe von primitiver Rekursion beschrieben. Dies hat den Vorteil, dass Theorembeweiser leicht Induktionsregeln ableiten können.

Man kann jedoch nicht einfach eine vollständige Programmiersprache mithilfe von primitiver Rekursion abbilden, da das Schleifenkonstrukt eine unendliche Ausführung erlaubt.

Der Trick, der von Schneider angewandt wird, ist die Aufteilung der Semantik in Kontrollfluss und Datenfluss. Damit lässt sich die Semantik mit primitiver Rekursion beschreiben. Der Kontrollfluss trifft Aussagen darüber, an welchen Stellen im Code sich der Programmablauf zu einem Zeitpunkt befindet und wo der Programmablauf im nächsten Zeitschritt sein wird. Da lediglich die `pause`-Statements Zeit verbrauchen, ist der Programmablauf zu jedem Zeitpunkt an irgendwelchen pause-Statements.

Der Datenfluss beschreibt die Veränderung der Daten über die Zeit hinweg. Dabei werden *guarded commands* benutzt, die aus einer Bedingung und einem Daten-verändernden Statement bestehen. Die Bedingung gibt hierbei an, wann das Statement ausgeführt wird.

Die finale Semantik besteht dann aus der Kombination von Kontroll- und Datenfluss.

Bei der nachfolgenden Betrachtung werden asynchrone Nebenläufigkeit und nichtdeterministische Auswahl erst einmal nicht betrachtet. Später wird dann erklärt, wie man diese Konstrukte auch behandeln kann

Die booleschen Aussagen sind im Folgenden für einen bestimmten Zeitpunkt zu sehen. Dabei ist der Wert von ϕ der Wert, zu dem die Aussage zum aktuellen Zeitpunkt auswertet. $X\phi$ bezeichnet den Wert der Aussage ϕ zum nächsten Zeitpunkt.

4 Der Kontrollfluss

Wie bereits erwähnt, kann sich der Programmablauf zu jedem beliebigen Zeitpunkt nur an `pause`-Statements aufhalten, da nur diese Zeit verbrauchen. Der Zustand des Kontrollflusses lässt sich also als eine Menge von gerade aktiven `pause`-Statements auffassen. Da jedes `pause`-Statement ein eindeutiges Label hat, kann man die Menge der aktiven `pause`-Statements auch als Menge von Labeln `l`, die aktiv sind, ausdrücken. Ein aktives Label wertet im Folgenden logisch zu wahr aus, ein nicht aktives zu falsch.

Sei die Menge $labels(S)$ die Menge aller in S enthaltenen Label. Zum Beispiel hat das Statement

```
if x then l1 : pause else l1 : pause end
```

die Labelmenge {`l1`, `l2`}

Der Kontrolfluss wird mithilfe der sechs Prädikate $in(S)$, $stutter(S)$, $inst(S)$, $enter(S)$, $move(S)$, $term(S)$ beschrieben. Diese Prädikate werden im Folgenden beschrieben:

4.1 in(S)

Das Prädikat $in(S)$ ist wahr, wenn der Kontrollfluss an mindestens einem pause-Statement in S ist:

$$in\,(S) :\equiv \bigvee_{l \in labels(S)} l$$

4.2 stutter(S)

Das Prädikat $stutter(S)$ beschreibt den Fall, dass der Kontrollfluss in S sich zum nächsten Zeitschritt nicht ändert. Das heißt, dass alle Label, die zum aktuellen Zeitpunkt aktiv sind, auch im nächsten Zeitschritt aktiv sind und alle aktuell inaktiven Label auch zum nächsten Zeitpunkt inaktiv sind:

$$stutter\,(S) :\equiv \bigwedge_{l \in labels(S)} (l = Xl)$$

Dabei ist es irrelevant, ob zwischen zwei Zeitschritten S verlassen wird und neu betreten oder ob der Kontrollfluss die ganze Zeit in S bleibt.

Das Hauptanwendungsgebiet für $stutter(S)$ ist das „`suspend S when` σ“-Statement.

```
suspend S when true
```

Wenn die Bedingung des `suspend`-Statements wahr ist, dann hält die Programmausführung in S an, es gilt dann

$$stutter(S) = true$$

4.3 inst(S)

Das Prädikat $inst(S)$ beschreibt alle Bedingungen, unter denen ein Statement S augenblicklich ausgeführt wird. Das heißt, dass wenn S zum Zeitpunkt t ausgeführt wird, dann wird das Statement im selben Zeitschritt wieder verlassen. Anders ausgedrückt: Bei der Ausführung von S wird kein pause-Statement erreicht. Rekursiv ist $inst(S)$ folgendermaßen definiert:

$inst$ (nothing) $:\equiv true$
$inst$ (emit x) $:\equiv inst$ (emit delayed x) $:\equiv true$
$inst$ (x := τ) $:\equiv inst$ (x := delayed τ) $:\equiv true$
$inst$ (l : pause) $:\equiv false$
$inst$ (if σ then S1 else S2 end) $:\equiv \sigma \wedge inst$ (S1) $\vee \neg\sigma \wedge inst$ (S2)
$inst$ (S1; S2) $:\equiv inst$ (S1) $\wedge inst$ (S2)
$inst$(S1 || S2) $:\equiv inst$ (S1) $\wedge inst$(S2)
$inst$ (while σ do S end) $:\equiv \neg\sigma \vee inst$ (S)
$inst$ ((weak) suspend S when σ) $:\equiv inst$ (S)
$inst$ ((weak) abort S when σ) $:\equiv inst$ (S)
$inst$ (local x in S end) $:\equiv inst$ (local y : α in S end) $:\equiv inst$ (S)
$inst$ (now σ) $:\equiv true$
$inst$ (during S holds σ) $:\equiv inst$ (S)

Wann ein Statement augenblicklich (engl.: instantaneous) ist, kann von der Ein- und Ausgabe abhängen. Zum Beispiel:

$$inst(\text{if i then l : pause; emit y else emit x end}) = \neg i$$

$$inst(\text{while b do l : pause end}) = \neg b$$

4.4 enter(S)

Die Formel $enter(S)$ beschreibt alle Fälle, in denen das Statement S zum nächsten Zeitpunkt hin betreten wird. Betreten heißt hier, dass der Kontrollfluss S betritt und zum nächsten Zeitpunkt ein `pause`-Statement innerhalb von S aktiv ist. Wenn S instantaneous ist,

dann kann $enter(S)$ also nicht gelten.

$$
\begin{aligned}
&enter\,(\text{nothing}) :\equiv false\\
&enter\,(\text{emit x}) :\equiv enter\,(\text{emit delayed x}) :\equiv false\\
&enter\,(\text{x} := \tau) :\equiv enter\,(\text{x := delayed } \tau) :\equiv false\\
&enter\,(\text{l : pause}) :\equiv Xl\\
&enter\,(\text{if } \sigma \text{ then S1 else S2 end}) :\equiv \begin{pmatrix} enter(\text{S1}) \wedge \neg Xin(\text{S2}) \wedge \sigma \vee \\ enter(\text{S2}) \wedge \neg Xin(\text{S1}) \wedge \neg\sigma \end{pmatrix}\\
&enter\,(\text{S1; S2}) :\equiv \begin{pmatrix} enter(\text{S1}) \wedge \neg Xin(\text{S2}) \vee \\ enter(\text{S2}) \wedge \neg Xin(\text{S1}) \wedge inst(\text{S1}) \end{pmatrix}\\
&enter(\text{S1} \mid\mid \text{S2}) :\equiv \begin{pmatrix} enter(\text{S2}) \wedge inst(\text{S1}) \wedge \neg Xin(\text{S1}) \\ enter(\text{S1}) \wedge inst(\text{S2}) \wedge \neg Xin(\text{S2}) \\ enter(\text{S1}) \wedge enter(\text{S2}) \end{pmatrix}\\
&enter\,(\text{while } \sigma \text{ do S end}) :\equiv \sigma \vee enter\,(\text{S})\\
&enter\,(\text{(weak) suspend S when } \sigma) :\equiv enter\,(\text{S})\\
&enter\,(\text{(weak) abort S when } \sigma) :\equiv enter\,(\text{S})\\
&enter\,(\text{local x in S end}) :\equiv enter\,(\text{local y : } \alpha \text{ in S end}) :\equiv enter\,(\text{S})\\
&enter\,(\text{now } \sigma) :\equiv false\\
&enter\,(\text{during S holds } \sigma) :\equiv enter\,(\text{S})
\end{aligned}
$$

4.5 term(S)

Die Formel $term(\text{S})$ beschreibt die Bedingungen, unter denen ein Statement zum nächsten Zeitpunkt hin verlassen wird. Damit $term(\text{S})$ gelten kann muss zum aktuellen Zeitpunkt der Kontrollfluss in S sein, d. h. es gilt $in(\text{S})$. $term(\text{S})$ ist folgendermaßen definiert:

$$
\begin{aligned}
&term\,(\text{nothing}) :\equiv false\\
&term\,(\text{emit x}) :\equiv term\,(\text{emit delayed x}) :\equiv false\\
&term\,(\text{x} := \tau) :\equiv term\,(\text{x := delayed } \tau) :\equiv false\\
&term\,(\text{l : pause}) :\equiv l\\
&term\,(\text{if } \sigma \text{ then S1 else S2 end}) :\equiv \begin{pmatrix} term(\text{S1}) \wedge \neg in(\text{S2}) \vee \\ term(\text{S2}) \wedge \neg in(\text{S1}) \end{pmatrix}\\
&term\,(\text{S1; S2}) :\equiv \begin{pmatrix} term(\text{S1}) \wedge \neg in(\text{S2}) \wedge inst(\text{S2}) \vee \\ term(\text{S2}) \wedge \neg in(\text{S1}) \end{pmatrix}\\
&term(\text{S1} \mid\mid \text{S2}) :\equiv \begin{pmatrix} term(\text{S1}) \wedge \neg in(\text{S2}) \vee \\ term(\text{S2}) \wedge \neg in(\text{S1}) \vee \\ term(\text{S1}) \wedge term(\text{S2}) \end{pmatrix}\\
&term\,(\text{while } \sigma \text{ do S end}) :\equiv \neg\sigma \wedge term\,(\text{S})\\
&term\,(\text{(weak) suspend S when } \sigma) :\equiv \neg\sigma \wedge term\,(\text{S})\\
&term\,(\text{(weak) abort S when } \sigma) :\equiv in(\text{S}) \wedge \sigma \vee term\,(\text{S})\\
&term\,(\text{local x in S end}) :\equiv term\,(\text{local y : } \alpha \text{ in S end}) :\equiv term\,(\text{S})\\
&term\,(\text{now } \sigma) :\equiv false\\
&term\,(\text{during S holds } \sigma) :\equiv term\,(\text{S})
\end{aligned}
$$

Damit $term(S)$ wahr werden kann, muss es mindestens ein aktives `pause`-Statement geben. Wo der Kontrollfluss im nächsten Zeitschritt ist, wird von $term(S)$ nicht beschrieben. Der Kontrollfluss kann außerhalb von S sein, kann S aber auch wieder betreten.

Nachdem jetzt der Eintritt und Austritt aus einem Statement beschrieben sind, fehlt noch der Statement-interne Kontrollfluss:

4.6 move(S)

Die Formel $move(S)$ beschreibt Fälle, in denen der Kontrollfluss in einem Statement S bleibt. Das heißt der Kontrollfluss ist zum aktuellen Zeitpunkt in S ($in(S)$ gilt), der Kontrollfluss ist zum nächsten Zeitpunkt in S ($Xin(S)$ gilt) und das Statement wird nicht verlassen ($term(S)$ gilt nicht).

$$
\begin{aligned}
&move\,(\text{nothing}) :\equiv false\\
&move\,(\text{emit x}) :\equiv move\,(\text{emit delayed x}) :\equiv false\\
&move\,(\text{x} := \tau) :\equiv move\,(\text{x := delayed } \tau) :\equiv false\\
&move\,(\text{l : pause}) :\equiv false\\
&move\,(\text{if } \sigma \text{ then S1 else S2 end}) :\equiv \begin{pmatrix} move(\text{S1}) \wedge \neg in(\text{S2}) \wedge \neg Xin(\text{S2}) \vee \\ move(\text{S2}) \wedge \neg in(\text{S1}) \wedge \neg Xin(\text{S1}) \end{pmatrix}\\
&move\,(\text{S1; S2}) :\equiv \begin{pmatrix} move(\text{S1}) \wedge \neg in(\text{S2}) \wedge \neg Xin(\text{S2}) \vee \\ move(\text{S2}) \wedge \neg in(\text{S1}) \wedge \neg Xin(\text{S1}) \vee \\ term(\text{S1}) \wedge \neg Xin(\text{S1}) \wedge \neg in(\text{S2}) \wedge enter(\text{S2}) \end{pmatrix}\\
&move(\text{S1} \mid\mid \text{S2}) :\equiv \begin{pmatrix} move(\text{S1}) \wedge \neg in(\text{S2}) \wedge \neg Xin(\text{S2}) \vee \\ move(\text{S2}) \wedge \neg in(\text{S1}) \wedge \neg Xin(\text{S1}) \vee \\ move(\text{S1}) \wedge move(\text{S2}) \vee \\ move(\text{S1}) \wedge term(\text{S2}) \wedge \neg Xin(\text{S2}) \vee \\ move(\text{S2}) \wedge term(\text{S1}) \wedge \neg Xin(\text{S1}) \end{pmatrix}\\
&move\,(\text{while } \sigma \text{ do S end}) :\equiv move(\text{S}) \vee term(\text{S}) \wedge \sigma \wedge enter(\text{S})\\
&move\,(\text{(weak) suspend S when } \sigma) :\equiv \begin{pmatrix} \sigma \wedge in(\text{S}) \wedge stutter(\text{S}) \vee \\ \neg\sigma \wedge move(\text{S}) \end{pmatrix}\\
&move\,(\text{(weak) abort S when } \sigma) :\equiv \neg\sigma \wedge move\,(\text{S})\\
&move\,(\text{local x in S end}) :\equiv move\,(\text{local y : } \alpha \text{ in S end}) :\equiv move\,(\text{S})\\
&move\,(\text{now } \sigma) :\equiv false\\
&move\,(\text{during S holds } \sigma) :\equiv move\,(\text{S})
\end{aligned}
$$

Anmerkung: Die Terme $\neg Xin(\text{S1}) \wedge \neg in(\text{S2})$ in der Regel für $move(\text{S1; S2})$ schließen nur aus, dass sich der Kontrollfluss unerlaubterweise zweimal im Statement `S1; S2` befindet. Es darf zum Beispiel kein zweiter Thread erzeugt werden, der auch das Statement bearbeitet.

4.7 Der Kontrollfluss

Mit den 6 Prädikaten kann der gesamte Kontrollfluss beschrieben werden. Im Folgenden sei *st* eine Startvariable für das Statement S. Der Kontrollfluss besteht dann aus der Beschreibung der Startzustände $\mathcal{I}_{cf}(st, S)$ und der Übergangsrelation $\mathcal{R}_{cf}(st, S)$. Diese beiden Formeln beschreiben den endlichen Automaten des Statements S. Jeder Zustand des Automaten steht für eine Menge von aktiven Labeln aus $labels(\mathrm{S})$. Der Startzustand ist der durch $\mathcal{I}_{cf}(st, S)$ beschriebene Zustand:

$$\mathcal{I}_{cf}(st, S) :\equiv \neg in(\mathrm{S}) \equiv \bigwedge_{l \in labels(S)} \neg l$$

Der Startzustand ist der, dass kein Label aktiv ist. Die Übergangsrelation $\mathcal{R}_{cf}(st, S)$ beschreibt dann die Zustandsübergänge des Automaten:

$$\mathcal{R}_{cf}(st, S) :\equiv \begin{pmatrix} (\neg in(\mathrm{S}) \vee term(\mathrm{S})) \wedge st \wedge inst(\mathrm{S}) \wedge \neg Xin(\mathrm{S}) \vee \\ (\neg in(\mathrm{S}) \vee term(\mathrm{S})) \wedge st \wedge enter(\mathrm{S}) \vee \\ (\neg in(\mathrm{S}) \vee term(\mathrm{S})) \wedge \neg st \wedge \neg Xin(\mathrm{S}) \vee \\ move(\mathrm{S}) in \end{pmatrix}$$

Diese Relation beschreibt alle 7 Möglichkeiten des Kontrollflusses:

- Wenn der Kontrollfluss nicht in S ist und st nicht gilt, dann bleibt der Kontrollfluss im Startzustand ($\neg in(\mathrm{S}) \wedge \neg st \wedge \neg Xin(\mathrm{S})$).
- Wenn der Kontrollfluss S verlässt und st nicht gilt, dann bleibt der Kontrollfluss ebenfalls im Startzustand ($term(\mathrm{S}) \wedge \neg st \wedge \neg Xin(\mathrm{S})$).
- Der Kontrollfluss kann in S bleiben ($move(\mathrm{S})$).
- Der Kontrollfluss kann S verlassen, aber da st gilt, wird S wieder betreten: ($term(\mathrm{S}) \wedge st \wedge enter(\mathrm{S})$).
- Das Statement S kann vom Startzustand betreten werden ($\neg in(\mathrm{S}) \wedge st \wedge enter(\mathrm{S})$).
- Der Kontrollfluss verlässt S. Da st gilt wird S sofort wieder ausgeführt, aber in dem Fall wird S augenblicklich ausgeführt und S sofort wieder verlassen ohne S neu zu betreten ($term(\mathrm{S}) \wedge st \wedge inst(\mathrm{S}) \wedge \neg Xin(\mathrm{S})$).
- Der Kontrollfluss ist im Startzustand und st gilt. S wird daraufhin augenblicklich ausgeführt und der Kontrollfluss befindet sich dann wieder im Startzustand ($\neg in(\mathrm{S}) \wedge st \wedge inst(\mathrm{S}) \wedge \neg Xin(\mathrm{S})$).

Mit dieser Beschreibung des Kontrollflusses konnten dann schon mithilfe von des Theorembeweisers HOL folgende Eigenschaften der Prädikate nachgewiesen werden:

- $enter(\mathrm{S}) \rightarrow Xin(\mathrm{S})$

- $enter(\mathrm{S}) \rightarrow \neg inst(\mathrm{S})$
- $term(\mathrm{S}) \rightarrow in(\mathrm{S})$
- $move(\mathrm{S}) \rightarrow in(\mathrm{S}) \wedge Xin(\mathrm{S})$
- $move(\mathrm{S}) \rightarrow \neg term(\mathrm{S})$
- $stutter(\mathrm{S}) \rightarrow in(\mathrm{S}) = Xin(\mathrm{S})$
- $\neg in(\mathrm{S}) \rightarrow stutter(\mathrm{S}) = \neg Xin(\mathrm{S})$

Die oben beschriebene Übergangsrelation ist in disjunktiver Form. Für einige Anwendungen ist es aber von Vorteil, wenn man eine Übergangsrelation in konjunktiver Form hat. Wenn zum Beispiel die Übergangsrelation als Beweisziel auftaucht, dann kann man zum Beispiel in HOL mit der Taktik STRIP_TAC die Übergangsrelation in kleinere Annahmen aufteilen, die man leichter beweisen kann, wenn die Übergangsrelation in konjunktiver Form vorliegt.

Es gibt auch Anwendungsgebiete, in denen die disjunktive Form von Vorteil ist. Zum Beispiel beim Modelchecking lässt sich dann eine disjunktive Partitionierung der Übergangsrelation benutzen[BCL91].

Die konjunktive Form der Zustandsübergangsrelation $\mathcal{R}'_{cf}(st, S)$ ist wie folgt definiert:

$$\mathcal{R}'_{cf}(st, S) :\equiv \begin{pmatrix} ((\neg in(\mathrm{S}) \vee term(\mathrm{S})) \wedge st \wedge inst(\mathrm{S}) \rightarrow \neg Xin(\mathrm{S})) \wedge \\ ((\neg in(\mathrm{S}) \vee term(\mathrm{S})) \wedge st \wedge \neg inst \rightarrow enter(\mathrm{S})) \wedge \\ ((\neg(\mathrm{S}) \vee term(\mathrm{S})) \wedge \neg st \rightarrow \neg Xin(\mathrm{S})) \wedge \\ ((in(\mathrm{S}) \wedge \neg term(\mathrm{S})) \rightarrow move(\mathrm{S})) \end{pmatrix}$$

Diese konjunktive Form ist aus propositionaler Sicht nicht identisch mit der disjunktiven Form der Übergangsgleichung. Trotzdem beschreiben beide Formeln das gleiche System.

Mithilfe der konjunktiven Form kann man auch eine rekursive Berechnung der Übergangsrelation über einem Statement S herleiten. Unter die Annahme, dass $st \rightarrow \neg in(\mathrm{S}) \vee term(\mathrm{S})$, kann die Zustandsübergangsrelation $\mathcal{R}_{cf}(st, S)$ folgendermaßen rekursiv über einem Quartz-Statement berechnet werden:

$\mathcal{R}_{cf}\,(st, \text{nothing}) \Leftrightarrow true$
$\mathcal{R}_{cf}\,(st, \text{emit x}) \Leftrightarrow \mathcal{R}_{cf}\,(st, \text{emit delayed x}) \Leftrightarrow true$
$\mathcal{R}_{cf}\,(st, \text{x} := \tau) \Leftrightarrow \mathcal{R}_{cf}\,(st, \text{x} := \text{delayed } \tau) \Leftrightarrow true$
$\mathcal{R}_{cf}\,(st, \text{l} : \text{pause}) \Leftrightarrow (Xl = st)$
$\mathcal{R}_{cf}\,(st, \text{if } \sigma \text{ then S1 else S2 end}) \Leftrightarrow \begin{pmatrix} \mathcal{R}_{cf}(st \wedge \sigma, \text{S1}) \wedge \\ \mathcal{R}_{cf}(st \wedge \neg\sigma, \text{S2}) \wedge \\ (in(\text{S1}) \rightarrow \neg in(\text{S2})) \end{pmatrix}$
$\mathcal{R}_{cf}\,(st, \text{S1; S2}) \Leftrightarrow \begin{pmatrix} \mathcal{R}_{cf}(st, \text{S1}) \wedge \\ \mathcal{R}_{cf}(st \wedge inst(\text{S1}) \vee term(\text{S1}), \text{S2}) \wedge \\ (in(\text{S1}) \rightarrow \neg in(\text{S2})) \end{pmatrix}$
$\mathcal{R}_{cf}\,(st, \text{S1} \,||\, \text{S2}) \Leftrightarrow (\mathcal{R}_{cf}(st, \text{S1}) \wedge \mathcal{R}_{cf}(st, \text{S2}))$
$\mathcal{R}_{cf}\,(st, \text{while } \sigma \text{ do S end}) \Leftrightarrow \begin{pmatrix} \mathcal{R}_{cf}(\sigma \wedge (st \vee term(\text{S})), \text{S}) \wedge \\ (term(\text{S}) \wedge \sigma \rightarrow \neg inst(\text{S})) \end{pmatrix}$
$\mathcal{R}_{cf}\,(st, \text{(weak) suspend S when } \sigma) \Leftrightarrow \begin{pmatrix} \mathcal{R}_{cf}(st, \text{S}) \wedge (in(\text{S}) \rightarrow \neg\sigma) \vee \\ (in(\text{S}) \wedge \sigma \wedge stutter(\text{S})) \end{pmatrix}$
$\mathcal{R}_{cf}\,(st, \text{(weak) abort S when } \sigma) \Leftrightarrow \begin{pmatrix} \mathcal{R}_{cf}(st, \text{S}) \wedge (in(\text{S}) \wedge Xin(\text{S}) \rightarrow \neg\sigma) \vee \\ in(\text{S}) \wedge \sigma \wedge st \wedge inst(\text{S}) \wedge \neg Xin(\text{S}) \vee \\ in(\text{S}) \wedge \sigma \wedge st \wedge enter(\text{S}) \vee \\ in(\text{S}) \wedge \sigma \wedge \neg st \wedge \neg Xin(\text{S}) \end{pmatrix}$
$\mathcal{R}_{cf}\,(st, \text{local x in S end}) \Leftrightarrow \mathcal{R}_{cf}\,(st, \text{local y} : \alpha \text{ in S end}) \Leftrightarrow \mathcal{R}_{cf}\,(st, \text{S})$
$\mathcal{R}_{cf}\,(st, \text{now } \sigma) \Leftrightarrow true$
$\mathcal{R}_{cf}\,(st, \text{during S holds } \sigma) \Leftrightarrow \mathcal{R}_{cf}\,(st, \text{S})$

Die Annahme $st \rightarrow \neg in(\text{S}) \vee term(\text{S})$ bedeutet lediglich, dass *st* nur gilt, wenn der S nicht aktiv ist oder gerade beendet wird, sodass S zum nächsten Zeitschritt tatsächlich auch wieder betreten werden kann. Diese Bedingung ist im Allgemeinen jedoch keine echte Einschränkung.

5 Definition des Datenflusses

Bisher wurde nur betrachtet, wie sich die Anweisungssequenzen über die Zeit hinweg verändern bzw. zu welchen Zeitpunkten man in welcher Anweisung ist. Beim Datenfluss soll es jetzt darum gehen, die Veränderungen der Inhalte aller Variablen zu erfassen. Beginnend vom Startzustand (Start des Programmes) bis zum Endzustand (Ende des Programmes). Damit kann verfolgen welchen Wert welche Variable zu einen bestimmten Zeitpunkt t hat.

Der Datenfluss basiert auf einer Menge von Wächterkommandos (*guarded commands*) zu einer Sequenz S. Ein solches Wächterkommando ist ein Tupel $(\gamma, \mathcal{C})$, wobei γ ein boolscher Ausdruck ist und wenn dieser gilt, dann wird das Kommando $\mathcal{C}$ ausgeführt. Die Kommandos, die ausgeführt werden, können nur vom Typ *emit x*, *emit delayed x*, *y* := τ, *y* := *delayed* τ und *now* σ sein.

Wenn die Bedingung gilt, dann kann der Wert einer Variable durch eine entsprechende

Anweisung verändert werden. Im Falle von *now* σ sagt man, dass wenn die Bedingung γ gilt, dann soll auch σ zu diesem Zeitpunkt gelten.

Jetzt werden Wächterkommandos der Statements (φ, S) definiert. Das φ steht hier für eine Precondition oder auch Vorbedingung, in der der aktuelle Zustand des Kontrollflusses und die aktuellen Variablen Werte enthalten sind:

$gcmd(\varphi, \text{nothing}) :\equiv \{\}$
$gcmd(\varphi, \text{emit x}) :\equiv \{(\varphi, \text{emit x})\}$
$gcmd(\varphi, \text{emit delayed x}) :\equiv \{(\varphi, \text{emit delayed x})\}$
$gcmd(\varphi, \text{y} := \tau) :\equiv \{(\varphi, \text{y} := \tau)\}$
$gcmd(\varphi, \text{y} := \text{delayed } \tau) :\equiv \{(\varphi, \text{y} := \text{delayed } \tau)\}$
$gcmd(\varphi, \text{l: pause}) :\equiv \{\}$
$gcmd(\varphi, \text{if } \sigma \text{ then S1 else S2 end}) :\equiv gcmd(\varphi \wedge \sigma, \text{S1}) \cup gcmd(\varphi \wedge \neg\sigma, \text{S2})$
$gcmd(\varphi, \text{S1;S2}) :\equiv gcmd(\varphi, \text{S1}) \cup gcmd(\varphi \wedge inst(\text{S1}) \vee term(\text{S1}), \text{S2})$
$gcmd(\varphi, \text{S1} \parallel \text{S2}) :\equiv gcmd(\varphi, \text{S1}) \cup gcmd(\varphi, \text{S2})$
$gcmd(\varphi, \text{while } \sigma \text{ do S end}) :\equiv gcmd((\varphi \vee term(\text{S})) \wedge \sigma, \text{S})$
$gcmd(\varphi, \text{suspend S when } \sigma) :\equiv \{(\gamma \wedge (in(\text{S}) \rightarrow \neg\sigma), \alpha \mid (\gamma, \alpha) \in gcmd(\varphi, \text{S})\}$
$gcmd(\varphi, \text{abort S when } \sigma) :\equiv \{(\gamma \wedge (in(\text{S}) \rightarrow \neg\sigma), \alpha \mid (\gamma, \alpha) \in gcmd(\varphi, \text{S})\}$
$gcmd(\varphi, \text{weak suspend S when } \sigma) :\equiv gcmd(\varphi, \text{weak abort S when } \sigma) :\equiv gcmd(\varphi, \text{S})$
$gcmd(\varphi, \text{now } \sigma) :\equiv \{(\varphi, now\sigma)\}$
$gcmd(\varphi, \text{during S holds } \sigma) :\equiv \{(in(\text{S}), now\sigma)\}$

Diese Definitionen sollten für die meisten Statements offensichtlich sein. Eine kurze Erklärung wird für das Statement S1;S2 gegeben. Der Kontrollfluss betritt zuerst S1 also muss $gcmd(\varphi$,S1) berechnet werden. Für den zweiten Teil S2 müssen zwei Fälle betrachtet werden. In dem ersten Fall kann S1 augenblicklich sein. Die Vorbedingung, mit der S2 betreten wird, ist damit $\varphi \wedge inst(\text{S1})$. Wenn S1 nicht augenblicklich ist, dann ist die Vorbedingung für S2, die letzte Position in S1, wo der Kontrollfluss war, bevor S1 verlassen wird. Dieser Punkt ist in $term(\text{S1})$ kodiert. Damit ergibt sich dann der Wächter für S2.

Man kann hier leicht sehen, dass der Wächter eine Kodierung der aktuellen Kontrollfussposition und der Werte der Variablen zu diesem Punkt ist.

Für die nächsten Definitionen vom Datenflusses muss man beachten, dass Event- und Zustandsvariablen unterschiedlich gehandhabt werden. Daher wird für jeden Typus eine eigene Definition formuliert.

Zuerst kommen die Eventvariablen. Angenommen es gibt nur die Wächter-Kommandos (α_1, emit x) ... (α_m, emit x) und (β_1, emit delayed x) ... (β_n, emit delayed x), dann wird folgendes definiert:

$$\mathcal{I}_{df}(st, x, S) :\equiv \left(x = \bigvee_{i=1}^{m} \alpha_i\right)$$
$$\mathcal{R}_{df}(st, x, S) :\equiv \left(\mathrm{X}x = \left(\bigvee_{i=1}^{n} \beta_i\right) \vee \mathrm{X}\left(\bigvee_{i=1}^{m} \alpha_i\right)\right)$$

Der Anfangswert einer Eventvariable x ist 1, wenn es zu Beginn eine Bedingung α_i gibt, die wahr ist.

Für den Übergang von einem Zeitpunkt t zum nächsten Punkt t+1, schaut man sich an, ob zum nächsten Zeitpunkt eine Bedingung α_i gilt. Dies wird im zweiten Teil der Disjunktion in $\mathcal{R}_{df}$ ausgedrückt. Gilt zum Zeitpunkt t eine Bedingung β_i, dann wird zum Zeitpunkt t+1 die Variable auf 1 gesetzt. Dieser Sachverhalt wird durch den ersten Teil der Disjunktion beschrieben.

Nach dem nun der Datenfluss für Eventvariablen definiert wurde, folgt die Definition des Datenflusses von Zustandsvariablen. Angenommen es gibt nur die Wächter-Kommandos (α_1, y := τ_1) ... (α_m, y : =τ_m) und (β_1, y := delayed π_1) ... (β_n, y := delayed π_n), dann wird folgendes definiert:

$$\mathcal{I}_{df}(st, y, S) :\equiv \bigwedge_{i=1}^{m} (\alpha_i \rightarrow [y = \tau_i])$$

$$\mathcal{R}_{df}(st, y, S) :\equiv \left(\begin{array}{c} (\bigwedge_{i=1}^{m} \mathrm{X}[\alpha_i \rightarrow (y = \tau_i)]) \wedge \\ (\bigwedge_{i=1}^{n} [\beta_i \rightarrow (\mathrm{X}y = \pi_i)]) \wedge \\ ([\bigwedge_{i=1}^{m} \neg \mathrm{X}\alpha_i \wedge \bigwedge_{i=1}^{n} \neg \beta_i] \rightarrow [\mathrm{X}y = y]) \end{array} \right)$$

Der Initialzustand einer Variable y ergibt sich, wenn die Bedingungen α_i gelten, dann wird nämlich τ_i ausgewertet und die Zuweisung y = τ_i durchgeführt.

Beim Übergang vom Zeitpunkt t nach t+1 werden alle Bedingungen α_i zum Zeitpunkt t+1 geprüft und bei Wahrheit die dazugehörige Zuweisung durchgeführt. Danach werden alle Bedingungen β_i abgefragt. Ist davon eine wahr, so wird π_i zu einem Wert ausgewertet (immer noch Zeitpunkt t) und dann im Zeitpunkt t+1 wird dieser Wert an y zugewiesen. Der dritte Teil der Konjunktion beschreibt den Fall, wenn keines der α_i und keines der β_i wahr ist. In diesem Fall erhält y zum Zeitpunkt t+1 den Wert, den es zum Zeitpunkt t hatte.

Die eben gezeigten Definitionen formalisieren die intuitive Benutzung der Wächterkommandos. Diese Gleichungen können aber unter Umständen, an bestimmten Positionen und für bestimmte Variablenwerte, falsch werden. Verschiedene Gleichungen können zur selben Zeit ausgeführt werden und müssen zur selben Zeit gelten. Ein Beispiel wäre die Zuweisungen von zwei Threads an eine Variable y. Die Werte, die die beiden Threads zuweisen (τ_1 und τ_2, sind unterschiedlich. Trotzdem muss nach der Zuweisung gelten : $y = \tau_1 \wedge y = \tau_2$. Diese Formel ist falsch. Wenn so eine Situation auftritt, hat man einen Schreibkonflikt (*write conflict*).

Es wird darauf hingewiesen, dass man sich bei Event Variablen keine Gedanken um einen Schreibkonflikt machen muss. Wenn zwei Threads eine (Event-) Variable aktiv machen, so ist sie auch aktiv und wenn ein Thread eine Variable aktiv macht und ein anderer nicht, so ist diese Variable trotzdem aktiv. Bei Zustandsvariablen muss man damit rechnen, dass Threads unterschiedliche Werte an eine einzige Variable zuweisen wollen. Das Problem, ob ein Programm einen Schreibkonflikt hat oder nicht, ist unentscheidbar. Trotzdem kann

man die Bedingungen für einen Schreibkonflikt angeben:

$$WC(y, st, S) := \begin{pmatrix} \bigvee_{i=1}^{m} \bigvee_{j=1}^{m} (\alpha_i \wedge \alpha_j \wedge \neg[\tau_i = \tau_j]) \vee \\ \bigvee_{i=1}^{n} \bigvee_{j=1}^{n} X(\beta_i \wedge \beta_j \wedge \neg[\pi_i = \pi_j]) \vee \\ \bigvee_{i=1}^{m} \bigvee_{j=1}^{n} (X\alpha_i \wedge \beta_j \wedge \neg[X\tau_i = \pi_j]) \end{pmatrix}$$

Damit ein Programm frei von Schreibkonflikten ist, muss zuerst Kontroll- und Datenfluss berechnet werden und danach werden die eben definierten Schreibkonflikt Situationen geprüft. Das ist aber, wie eben schon erwähnt, unentscheidbar. Eine Alternative, die einfacher, aber trotzdem noch unentscheidbar ist, ist eine statische Prüfung, in der erreichbare und unerreichbare Zustände ignoriert werden. Dies kann z.B. mit einem Theorembeweiser erster Stufe gemacht werden.

Neben dem Schreibkonflikt gibt es hier noch ein anderes Problem. Was ist, wenn keines der initialen α_i's gilt? Der Wert der Variable y ist dann nicht definiert. Eine mögliche Lösung ist es, einfach einen beliebigen Wert zu dem entsprechenden Datentypen zu wählen. Eine andere Möglichkeit, die hier verwendet wird, ist es für alle Variablen einen festen initialen Wert τ_0 zu setzen.

Jetzt folgt eine zweite äquivalente und wohldefinierte Definition für Zustandsvariablen, die einen initialen Wert für alle Variablen festlegt:

$$\mathcal{I}_{df}(st, y, S) \equiv \left(y = \begin{pmatrix} \text{if } \alpha_1 \text{ then } \tau_1 \\ . \\ . \\ . \\ \text{elsif } \alpha_m \text{ then } \tau_n \\ \text{else } \tau_0 \end{pmatrix} \right)$$

$$\mathcal{R}_{df}(st, y, S) \equiv \left(\mathrm{X}y = \begin{pmatrix} \text{if } \mathrm{X}\alpha_1 \text{ then } \mathrm{X}\tau_1 \\ \text{elsif } \mathrm{X}\alpha_2 \text{ then } \mathrm{X}\tau_2 \\ . \\ . \\ . \\ \text{elsif } \mathrm{X}\alpha_m \text{ then } \mathrm{X}\tau_m \\ \text{elsif } \beta_1 \text{ then } \pi_1 \\ . \\ . \\ . \\ \text{elsif } \beta_n \text{ then } \pi_n \\ \text{else } y \end{pmatrix} \right)$$

Nachdem der Datenfluss von Event- und Zustandsvariablen definiert wurde, können diese beiden Definitionen zusammengeführt werden, um eine Gesamtdefinition des Datenflusses

für ein Statements zu erhalten:

$$\mathcal{I}_{df}(st, S) :\equiv \bigwedge_{i=1}^{m} \mathcal{I}_{df}(st, y_i, S)$$
$$\mathcal{R}_{df}(st, S) :\equiv \bigwedge_{i=1}^{m} \mathcal{R}_{df}(st, y_i, S) \wedge \bigwedge_{i=1}^{p} (\varphi_i \rightarrow \sigma_i)$$

Der initiale Variablenzustand einer Anweisung S ergibt sich also dadurch, dass man alle Anfangszustände jeder Variable (egal ob Event oder Zustand) berechnet und diese durch eine Konjunktion verknüpft.

Die Übergangsrelationen sind ähnlich definiert, man berechnet den Übergang jeder einzelnen Variable und man muss noch zusätzlich die *now* σ Anweisungen prüfen. Wenn der Guard φ_i gilt, dann muss auch die Bedingung σ_i gelten.

Damit ist auch der Datenfluss komplett definiert. Zum Abschluss kann nun die Semantik einer Anweisung S als Kombination von Kontroll- und Datenfluss definiert werden:

$$\mathcal{I}(st, S) :\equiv \mathcal{I}_{cf}(st, S) \wedge \mathcal{I}_{df}(st, S)$$
$$\mathcal{R}(st, S) :\equiv \mathcal{R}_{cf}(st, S) \wedge \mathcal{R}_{df}(st, S)$$

6 Ergebnisse

Mit Hilfe dieses ganzen Semantik Apparates, kann jedes Quartz Programm in eine logische Formel überführt werden, die mit einem Theorem Beweiser wie HOL benutzt werden kann. Ein solches überführtes Quartz Programm kann nun über HOL's Interferenz Regeln manipuliert werden. Quartz Anweisungen können so in ihre entsprechenden endlichen Automaten überführt werden. Mit HOL's rewrite machinery wird sogar für die Überführung zum Automaten ein formaler Beweis mitgeliefert. HOL's Implementierung sichert einem sogar noch zu, dass die Code Generierung frei von Fehlern ist.

Des weiteren sind nun Formeln möglich wie $\forall P, Q : Quartz(\alpha).(P||Q) = (Q||P)$. Man kann also über *alle* Quartz Programme argumentieren und Theoreme über die Sprache selbst beweisen.

Andere Möglichkeiten, die einem zur Verfügung steht, ist das Verifizieren von Programm Schemata, Beweis von Invarianten und Induktionen über die Anzahl von Threads oder den Datentypen.

Literatur

[BCL91] J. R. Burch, E. M. Clarke und D. E. Long. Symbolic Model Checking with Partitioned Transition Relations. Seiten 49–58. North-Holland, 1991.

[Mel88] Thomas Melham. Automating Recursive Type Definitions in Higher Order Logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, Seiten 341–386. Springer-Verlag, 1988.

[Sch01] K. Schneider. Embedding Imperative Synchronous Languages in Interactive Theorem Provers. *2010 10th International Conference on Application of Concurrency to System Design*, 0:143–154, 2001.