Kompilieren schizophrener synchroner Programme

Christian Dohnert

18. Februar 2013

Synchrone Programmiersprachen weisen deutliche Unterschiede Sprachen auf, deren Befehle sequentiell abgearbeitet werden. Beim Übersetzen solcher Sprache in ausführbare Programme stößt man auf sogenannte schizophrene Probleme. Diese aufzulösen ist sehr aufwendig und führt oft dazu, dass bereits der Zwischencode exponentiell viele Zeilen enthält. Schneider und Wenz haben in „A New Method for Compiling Schizophrenic Synchronous Programs“ für zwei häufige schizophrene Probleme eine neue Strategie entwickelt, diese aufzulösen. Sofern nicht anders angegeben, stammen alle Informationen aus dieser Quelle.

1 Einleitung

Ohne eine klare Vorstellung zu den Adjektiven *synchron* und *schizophren*, kann man kaum über schizophrene, synchrone Programme reden/schreiben. Deshalb möchte ich zur Einleitung diese zwei wesentlichen Begriffe klären.

Synchron

Definition 1 (*synchron*)

> *Das Wort synchron vereint zwei altgriechische Wortstämme: συν syn (mit, gemeinsam) und χρονοζ chronos (Zeit). Es bedeutet im ursprünglichen Sinn gleichzeitig oder zeitlich übereinstimmend, weiter auch zu einem Zeitpunkt auf mehrere Räume.[Wikipedia]*

Der ursprüngliche Sinn *zeitgleich* ist genau der, den wir hier brauchen.

Schizophren

Definition 2 (*Schizophrenie*)

> *A mental illness in which a person becomes unable to link thought, emotion and behaviour, leading to withdrawal from reality and personal relationships. [OALD]*

Laut dieser recht kurzen Definition aus dem Oxford Advanced Learners Dictionary, ist die Schizophrenie also eines Geisteskrankheit, bei der *Personen* nicht in der Lage sind Gedanken, Emotionen und Handeln in Einklang zu bringen. Diese Definition ist also auf dem Feld der Programmiersprachen unbrauchbar. Näher kommt man der Sache, wenn man in den Duden schaut. Dort gibt es 2 Einträge für das Adjektiv *schizophren*. Neben der offensichtlichen Erklärung än Schizophrenie erkrankt"[Duden] gibt es auch folgende bildungssprachliche Erklärung:

Definition 3 (*schizophren*)

> *(bildungssprachlich) in sich widersprüchlich, in hohem Grade inkonsequent[Duden]*

Diese Erklärung bringt uns schon weiter. Es geht also um Programme, die sich inkonsequent bzw. widersprüchlich verhalten.

2 Synchrone Programmiersprachen

Synchrone Programmiersprachen dienen dazu Programme zu schreiben, in denen mehrere Anweisungen zeitgleich ablaufen. Um echte Synchronität zu erreichen, darf eine einzelne Anweisung keine Zeit benötigen. Benötigen Anweisungen keine Zeit, starten sie alle sofort und enden alle ebenfalls sofort. Dieses Konzept erscheint zunächst schon einmal schwierig, da jede Anweisung eigentlich Zeit benötigt. In synchronen Sprachen wie Esterel [1] werden

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

Anweisungen in Mikroschritten ausgeführt. Diese benötigen keine Zeit. Zeitverbrauch muss explizit durch das Schlüsselwort **pause** markiert werden. Ist das Programm bei einem **pause**-Statement angekommen, wird die Ausführung bis zu nächsten logischen Zeiteinheit pausiert. Die kleinste logische Zeiteinheit groß genug zu wählen, ist nur die die halbe Miete. Programme werden in Automaten umgewandelt in denen alles gleichzeitig passiert.

Quartz

Quartz ist eine Variante von Esterel und erweitert die Sprache um einige Funktionen.

Syntax

Vorausgesetzt S, S_1 und S_2 sind gültige Anweisungen, x ist eine *Eventvariable*, y ist eine *Zustandsvariable*, σ ist ein boolescher Ausdruck und l ist eine *Positionsvariable*, stellt folgende Liste die Menge der gültigen Anweisung in Quartz dar.

- **nothing**
- **emit** x und **emit delayed** x
- $y := \tau$ und $y :=$ **delayed** x
- l : **pause**
- **if** σ **then** S_1 **else** S_2 **end**
- $S_1; S_2$
- $S_1 \| S_2$
- $S_1 \| \| S_2$
- **choose** $S_1 [] S_2$
- **do** S **while** σ
- **suspend** S **when** σ
- **weak suspend** S **when** σ
- **abort** S **when** σ
- **weak abort** S **when** σ
- **local** x **in** S **end**
- **now** σ
- **during** S **holds** σ

Die Bedeutung der meisten Befehle und Strukturen, sind ersichtlich. Die informelle Semantik wird in [ScWe02] umfassend aufgelistet. *Zustandsvariablen* unterscheiden sich durch *Eventvariablen* durch einen angegebenen Typ. *Eventvariablen* werden als boolesche Variablen angesehen und können somit die Werte **true** oder **false** annehmen. Außerdem werden *Eventvariablen* am Anfang jedes Makroschritts automatisch auf 0 gesetzt, während *Zustandsvariablen* einen einmal gesetzten Wert behalten. Ein Quartz-Modul hat folgende allgemeine Form:

```
1   module M
2     input a1,...,an,b1 : A1,...,bm : Am;
3     output x1,...xn,y1 : B1,...,ym : Bm;
4     S
5   end module
```

Semantik

Eine umfassende formale Beschreibung der Semantik von Quartz geben Schneider und Wenz in [ScWe02]. Folgende Funktionen sind für das Auflösen der schizophrenen Probleme essentiell und bedürfen einer Erläuterung.

gcmd(φ, S) ist eine Menge von Tupeln der Form (γ, C), wobei C eine Anweisung ist, die Daten manipuliert, z.B. eine Anweisung ist und γ eine Vorbedingung. Wenn die Vorbedingung γ wahr ist, wird C sofort ausgeführt.

inst(S) ist wahr, gdw. der Kontrollfluss nicht in S bleibt, wenn S gestartet wird. S würde also unmittelbar zu diesem Zeitpunkt ausgeführt werden.

in(S) Disjunktion der **pause**-Anweisungen in S. Deshalb ist in(S) an zu einem beliebigem Zeitpunkt wahr, gdw. der Kontrollfluss an einer Stelle innerhalb von S ist.

term(S) beschreibt alle möglichen Bedingungen an denen der Kontrollfluss in S ist und S verlassen kann. Der Kontrollfluss kann sich zum nächsten Zeitpunkt trotzdem wieder in S befinden, wenn durch z.B. äußere Schleifen S wieder angesprungen wird.

3 Schizophrene Programme

Schneider und Wenz [ScWe01] stellen zwei verschiedene schizophrene Programmbeispiele vor. Im ersten wird gezeigt, wie Abbruchbedingungen zu schizophrenen Situationen führen können. Im zweiten Beispiel wird aufgezeigt, wie lokale Variablen zu inkonsequentem Verhalten führen können. Für beide problematischen Situationen gibt es eine Auflösung, die von Schneider und Wenz mittels einer Software zur Beweisführung als korrekt nachgewiesen wurde.

Schizophrene Abbruchanweisung

Wie Abbruchanweisungen zu schizophrenen Programmen führen, zeigen Schneider und Wenz mit folgendem Beispiel:

```
1  module AbortReincarnation
2    input i;
3    output a, b;
4    do
5      abort
6        emit a;
7        l: pause;
8        emit b
9      when i
```

```
10    while 1
11  end
```

Listing 1: AbortReincarnation

Die Schizophrenie kommt hier durch das das sofortige Verlassen und Wiedereintreten in die Schleife zustande. Dadurch entstehen in einem Makroschritt zwei Eintritte in die Schleife. Da sich beide Ausprägungen anders Verhalten muss dazwischen unterschieden werden. Da aber beide zur gleichen Zeit existieren und auf unterschiedliche Mikroschritte im selben Makroschritt verweisen, ist diese Unterscheidung mit Prädikaten, die allein auf Makroschritten beruhen, nicht mehr zu leisten.

Surface und Depth einer Anweisung

Es reicht nicht die Semantik von Makroschritten zu betrachten, um schizophrene Probleme in synchronen Programmen aufzulösen. Um die Semantik von Mikroschritten zu betrachten, führen Schneider und Wenz zwei Funktionen ein. Die erste ist *surface* und die zweite *depth* . Die Begriffe sind aus [Berry99] entliehen, aber anders definiert. Während bei Berry *surface* und *depth* Teile von Schaltkreisen beschreiben, dienen sie hier dazu Quartz-Anweisungen in andere Quartz-Anweisung zu überführen. Dabei ist emph-surface der Teil einer Quartz-Anweisung S, der keine Zeit verbraucht und somit sofort nach dem Eintritt in S ausgeführt wird. Der verbleibende Teil der Quartz-Anweisung ist *depth* . Unter den gleichen Voraussetzungen wie in 2 ist für jede beliebige Quartz-Anweisung S surface(S) wie folgt definiert:

- surface(**nothing**) :≡ **nothing**
- surface(**emit** x) :≡ **emit** x
- surface(**emit delayed** x) :≡ **emit delayed** x
- surface($y := \tau$) :≡ $y := \tau$
- surface($y :=$ **delayed** τ) :≡ $y :=$ **delayed** τ
- surface(l : **pause**) :≡ **nothing**
- surface(**if** σ **then** S_1 **else** S_2 **end**) :≡ **if** σ **then** surface(S_1) **else** surface(S_2) **end**
- surface($S_1; S_2$) :≡ surface(S1); **if** inst(S_1) **then** surface(S_2) **else nothing end**
- surface(S_1 || S_2) :≡ surface(S_1) || surface(S_2)
- surface(**do** S **while** σ) :≡ surface(S)
- surface(**suspend** S **when** σ) :≡ surface(**weak suspend** S **when** σ) :≡ surface(S)
- surface(**abort** S **when** σ) :≡ surface(**weak abort** S **when** σ) :≡ surface(S)
- surface(**local** x **in** S **end**) :≡ surface(S)
- surface(**local** $x : \alpha$ **in** S **end**) :≡ surface(S)
- surface(**now** σ) :≡ **now** σ

- surface(**during** S **holds** σ) :≡ surface(S)

Da alle Datenmanipulation in surface(S) bereits enthalten sind, sollte sofort klar sein, dass depth(S) keine Daten unmittelbar nach dem Start verändert. Der restliche Daten- und Kontrollfluss muss jedoch erhalten bleiben. Gelten wieder die gleichen Bedingungen wie in 2, so gilt für jede gültige Quartz-Anweisung S folgende rekursive Definition von depth(S):

- depth(**nothing**) :≡ **nothing**
- depth(**emit** x) :≡ **nothing**
- depth(**emit delayed** x) :≡ **nothing**
- depth($y := \tau$) :≡ $y := \tau$
- depth($y :=$ **delayed** τ) :≡ **nothing**
- depth(l : **pause**) :≡ l : **pause**
- depth(**if** σ **then** S_1 **else** S_2 **end**) :≡ **if** σ **then** depth(S_1) **else** depth(S_2) **end**
- depth($S_1; S_2$) :≡

$$\left(\begin{array}{c} \text{depth}(S_1); \\ \textbf{if } \text{in}(S_1) \textbf{ then } \text{surface}(S_2) \\ \text{depth}(S_2) \end{array}\right)$$

- depth(S_1 || S_2) :≡ depth(S_1) || depth(S_2)
- depth(**do** S **while** σ) :≡

$$\left(\begin{array}{c} \textbf{do} \\ \text{depth}(S); \\ \textbf{if } \sigma \textbf{ then } \text{surface}(S) \textbf{ end}; \\ \textbf{while } \sigma \end{array}\right)$$

- depth(**suspend** S **when** σ) :≡ **suspend** S **when** σ
- depth(**weak suspend** S **when** σ) :≡ **weak suspend** S **when** σ
- depth(**abort** S **when** σ) :≡ **abort** depth(S) **when** σ
- depth(**weak abort** S **when** σ) :≡ **weak abort** depth(S) **when** σ
- depth(**local** x **in** S **end**) :≡ depth(S)
- depth(**local** $x : \alpha$ **in** S **end**) :≡ depth(S)
- depth(**now** σ) :≡ **nothing**
- depth(**during** S **holds** σ) :≡ depth(**during** depth(S) **holds** σ)

Die Krux hierbei ist, dass der Kontrollfluss von S und depth(S) gleich ist. Mit Hilfe des HOL-Systems haben sie folgenden Satz bewiesen:

Satz 1 (*Kontrollfluss von surface und depth*)

- *surface(S)*
- *S und depth(S) haben den gleichen Kontrollfluss*
- *S und surface(S);depth(S) haben den gleichen Kontrollfluss*

Daraus folgt, dass sogar S, surface(S);depth(S) und depth(S) den gleichen Kontrollfluss aufweisen. Der Datenfluss bleibt jedoch nicht immer erhalten. Das erkennt man wenn man *surface* und *depth* auf das Beispiel *AbortReincarnation*1 anwendet.

- surface(S):

```
1       emit a
```

- depth(S):

```
1  do
2    abort
3      l:pause;
4      emit b
5    when i;
6    emit a
7  while 1
```

- gcmd(st, surface(S)): {(st,**emit** a)}
- gcmd(st, depth(S)): {(l, **emit** a), ($l \wedge \neg i$, **emit** b)}

Der Unterschied entsteht aber nur durch die schizophrene Abbruchanweisung. Deshalb beschränken Schneider und Wenz ihre Betrachtung erst einmal auf Abbruchanweisungen, deren *surface* leer ist. Dann ist der Datenfluss von S der gleiche wie surface(S);depth(S). Bewiesen wurde das wieder mit HOL.

Satz 2 (*Datenfluss von surface und depth*)

Für jedes Quartz-Programm S, in dem für jede Anweisungsfolge der Form **abort** P **when** σ *gilt, dass die Menge gcmd(φ, surface(P)) für jede Vorbedingung φ leer ist, gilt: Der Datenfluss von S ist gleich dem Datenfluss von surface(S);depth(S)*

Die Mengen gcmd(φ, surface(S);depth(S)) und gcmd(S) müssen nicht notwendiger Weise identisch sein. Beide Mengen beschreiben aber, unter den gegebenen Annahmen den selben Datenfluss. Daraus ergibt sich dann die folgende Transformationsregel für strikte Abbruchanweisungen, die potentielle schizophrene Situationen vermeidet.
abort S **when** $\sigma \equiv surface(S)$; **abort** $depth(S)$**when**σ

Wie man leicht sieht, erfüllen die dadurch produzierten Anweisungen die Anforderungen des obigen Satzes. Mit Hilfe der Semantik der Mikroschritte von *surface* und *depth* kann auch leicht nachgewiesen werden, dass die Gleichheit wirklich gilt. Schneider und Wenz definieren diese Transformation als die Semantik der **abort**-Anweisung um die Semantik von Quartz der von Esterel anzugleichen.

Schizophrene lokale Deklarationen

Komplexer wird es bei lokalen Deklarationen. Ein Problem, welches von Schneider und Wenz [ScWe01] LocalReincarnation genannt wird veranschau-

licht die *Schizophrenie* die durch lokale Variablen in Schleifen bei synchronen Programmen entstehen kann.

```
module LocalReincarnation:
  output xOn, xOff;
    do
      local x in
        if x then emit xOn else emit xOff end;
        l: pause;
        emit x;
        if x then emit xOn else emit xOff end;
      end local
    while 1
end
```

Listing 2: LocalReincarnation

Das Programm verhält sich wie folgt:

1. $xOff$ wird sofort gesendet[2].
2. Es wird bis zum nächsten Makroschritt gewartet.
3. x wird gesendet.
4. xOn wird gesendet.
5. Die Schleife wird beendet und sofort wird ein neues Exemplar der Variablen x angelegt.
6. Da diese neue Ausprägung wieder den Wert **false** hat, wird auch xOff sofort wieder ausgegeben.
7. Es wird auf den nächsten Makroschritt gewartet

Nachdem das erste Mal xOn gesendet wird, werden in jedem weiteren Makroschritt beide Signale gesendet. Dieses Verhalten ist durchaus unproblematisch, da xOn und $xOff$ voneinander unabhängige Signale sind. Problematisch ist aber, dass ab dem zweiten Schleifendurchlauf immer zwei Exemplare der Variablen x aktuell sind. Diese haben auch immer unterschiedliche Werte, da beim Starten des Geltungsbereichs x den Wert **false** hat, das Exemplar aus dem vorhergehenden Schleifendurchlauf aber den Wert true **besitzt**. Um diese Situation aufzulösen würde es genügen, den Bereich, in dem x Gültigkeit hat in *surface* und *depth* aufzuteilen und dabei x durch x_0 im *surface*-Abschnitt zu ersetzen. Das Ergebnis sieht wie folgt aus:

```
module LocalReincarnationSolved:
  output xOn, xOff;
    do
      local x0 in
        if x0 then emit xOn else emit xOff end;
      end locale;
      local x in
        l: pause;
        emit x;
        if x then emit xOn else emit xOff end;
      end local
    while 1
```

[2]Ich verwende hier das Verb *senden*, da ich keine geeignetere Übersetzung für das englische Verb *emit* gefunden habe.

```
13 end
```

Listing 3: LocalReincarnation

Das Verhalten dieses Programms entspricht genau dem, des oberen. Es treten aber keine schizophrenen Situationen mehr auf. Da dieses Problem auch tiefer geschachtelt werden kann, wie man in Abbildung 4 in [ScWe01] sieht, ist diese sehr einfache Lösung nicht der Weisheit letzter Schluss. Schneider und Wenz legen dar, dass auch für mehrfach verschachtelte lokale Variablen ein Algorithmus existiert, mit dem die Schizophrenie aufgelöst werden kann. Der Nachteil am Algorithmus von Berry aus „The constructive semantics of pure Esterel"ist, dass für jede Ebene der Verschachtelung eine Kopie der lokalen Variablen in der nächst höheren Ebene angelegt werden muss. Von der innersten Variable müssen also n Kopien angelegt werden, von der nächst höheren $n-1$, bis zur lokalen Variable in der obersten Ebene, von der nur noch eine Kopie angelegt werden muss. Das Zusammenspiel dieser Kopien ist nicht einfach zu handhaben.
Schneider und Wenz haben eine Methode gefunden, die das Kopieren der Variablen umgeht. Dabei ist es im Allgemeinen nicht möglich das Duplizieren der *surface* zu umgehen.
Nur die lokale Variable im *surface*-Abschnitt zu ersetzen reicht nicht immer aus. Müssen sofort nach Beginn des *depth*-Abschnitts Bedingungen ausgewertet werden, die auf x verweisen, muss statt x_0 stattdessen genutzt werden. Die Umformung wird durch die Funktion $\text{surfcond}_s^{s_0}(t,\ \varphi,\ \text{depth}(S)$ vervollständigt.

Definition 4 (*Korrigierende* surface-*Bedingungen*)

Für jede gültige Quartz-Anweisung S ohne lokale Signaldeklarationen, wird die Funktion $surfcond_s^{s_0}(t,\ \varphi,\ S)$ mit t als Abbruchbedingung und φ als Vorbedingung für S definiert.

- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{nothing}) :\equiv \textbf{nothing}$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{emit}\ x) :\equiv \textbf{emit}\ x$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{emit delayed}\ x) :\equiv$

$$\begin{cases} \textbf{if}\ \neg t\ \textbf{then emit delayed}\ s\ \textbf{end} & : if\ x \equiv s \\ \textbf{emit delayed}\ x & : if\ x \not\equiv s \end{cases}$$

- $surfcond_s^{s_0}(t,\ \varphi,\ y := \tau) :\equiv y := \tau$
- $surfcond_s^{s_0}(t,\ \varphi,\ y :=\textbf{delayed}\ \tau) :\equiv$

$$\begin{cases} \textbf{if}\ \neg t\ s := \textbf{delayed}\ \tau & : if\ y \equiv s \\ y := \textbf{delayed}\ \tau & : if\ y \not\equiv s \end{cases}$$

- $surfcond_s^{s_0}(t,\ \varphi,\ l :\textbf{pause}) :\equiv l :\textbf{pause}$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{if}\ \sigma\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}) :\equiv \textbf{if}\ \ \varphi \wedge [\sigma]_s^{s_0} \vee \neg\varphi \wedge \sigma$
  $\textbf{then}\ surfcond_s^{s_0}(t,\ \varphi \wedge [\sigma]_s^{s_0},\ S_1)$
  $\textbf{else}\ surfcond_s^{s_0}(t,\ \varphi \wedge \neg[\sigma]_s^{s_0},\ S_2)$
  $\textbf{end}$

- $surfcond_s^{s_0}(t,\ \varphi,\ S_1;S_2) :\equiv surfcond_s^{s_0}(t,\ \varphi,\ S_1);\ surfcond_s^{s_0}(t,\ \varphi \wedge [inst(S_1)]_s^{s_0},\ S_2)$
- $surfcond_s^{s_0}(t,\ \varphi,\ S_1 \ //\ S_2) :\equiv$
  $surfcond_s^{s_0}(t,\ \varphi,\ S_1) \ //\ surfcond_s^{s_0}(t,\ \varphi,\ S_2)$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{do}\ S\ \textbf{while}\ \sigma) :\equiv$
  $\textbf{do}\ surfcond_s^{s_0}(t,\ \varphi,\ S)\ \textbf{while}\ \sigma$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{suspend}\ S\ \textbf{when}\ \sigma) :\equiv$
  $\textbf{suspend}\ surfcond_s^{s_0}(t,\ \varphi,\ S)\ \textbf{when}\ \sigma$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{weak suspend}\ S\ \textbf{when}\ \sigma) :\equiv$
  $\textbf{weak suspend}\ surfcond_s^{s_0}(t,\ \varphi,\ S)\ \textbf{when}\ \sigma$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{abort}\ S\ \textbf{when}\ \sigma) :\equiv$
  $\textbf{abort}\ surfcond_s^{s_0}(t,\ \varphi,\ S)\ \textbf{when}\ \sigma$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{weak abort}\ S\ \textbf{when}\ \sigma) :\equiv$
  $\textbf{weak abort}\ surfcond_s^{s_0}(t,\ \varphi,\ S)\ \textbf{when}\ \sigma$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{local}\ x\ \textbf{in}\ S\ \textbf{end}) :\equiv$
  $\textbf{local}\ x\ \textbf{in}\ surfcond_s^{s_0}(t,\ \varphi,S)\ \textbf{end}$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{local}\ y:\alpha\ \textbf{in}\ S\ \textbf{end}) :\equiv$
  $\textbf{local}\ y:alpha\ \textbf{in}\ surfcond_s^{s_0}(t,\ \varphi,S)\ \textbf{end}$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{now}\ \sigma) :\equiv \textbf{now}\ \sigma$
- $surfcond_s^{s_0}(t,\ \varphi,\ \textbf{during}\ S\ \textbf{holds}\ \sigma) :\equiv$
  $\textbf{during}\ surfcond_s^{s_0}(t,\ \varphi,\ S)\ \textbf{holds}\ \sigma$

Die Vorbedingung φ beschreibt dabei alle möglichen Bedingungen unter denen S keine Zeit zur Ausführung benötigt. Man beachte, dass sich bei Zuweisungen jeglicher Art die Variable nie ersetzt werden müssen. Stehen Zuweisungen im *depth*-Abschnitt, werden diese nie sofort ausgeführt, da diese sonst noch zum *surface*-Abschnitt gehören würden. Um das Problem der wiederkehrenden lokalen Variable zu lösen müssen die Ergebnisse der folgenden zwei Funktionen den Block mit der lokalen Variablen ersetzen.

1. subst_immed(surface(S),x,x_0);
2. surfcond$_s^{s_0}(t,\ \varphi,$ depth(S))

Durch subst_immed werden alle Vorkommen von x durch x_0 ersetzt. Ausgenommen sind davon verzögerte Anweisungen. Es ist nun ersichtlich, dass das Ergebnis äquivalent zum Ursprünglichen Quelltext ist. Schneider und Wenz begründen in [ScWe01] die Korrektheit auch noch informell. Damit ist auch der zentrale Satz des ganzen Artikels bewiesen.

Satz 3 (*Schizophrene Probleme beseitigen*)

Jedes beliebige Quartz-Programm kann durch ein Quartz-Programm ersetzt werden, dass durch die Umformungen

- $\textbf{abort}\ S\ \textbf{when}\ \sigma \equiv surface(S);\ \textbf{abort}\ depth(S)\textbf{when}\sigma$
- $\textbf{locale}\ x\ \textbf{in}\ S\ end :\equiv$

$$\begin{bmatrix} subst\_immed(surface(S), x, x_0); \\ surfcond_s^{s_0}(t, \varphi, depth(S)) \end{bmatrix}$$

4 Abschließendes

Schneider und Wenz zeigen im vorletzten Abschnitt noch die tatsächliche Implementierung der Umformungen und analysieren Laufzeit und Größe der Ausgabe ihrer Algorithmen. Sie kommen zu dem Schluss, dass die sich die Komplexitätsklasse ihrer Algorithmen die gleiche ist, wie die der Algorithmen von Berry. Die Länge des produzierten Zwischencodes liegt immer noch in der Größenordnung von $O(2^n)$. Getestet haben sie das ganze mit dem Modul $MultipleReincarnation_n$ und haben bis zu zehn Ebenen tief Deklarationen von lokalen Variablen verschachtelt. Obwohl die Länge des Zwischencodes gemessen an der Verschachtelungstiefe in der gleichen Komplexitätsklasse liegen, unterscheiden sich die absoluten Zahlen stark. Der Zwischencode der Quartz-Programme hat nur circa 30% des Umfangs des von Esterel erzeugten Zwischencodes. Auch nach der Optimierung und Übersetzung in die Zielsprache, bleibt die Quartz-Variante weniger umfangreich.

Literatur

[OALD] `http://oald8.oxfordlearnersdictionaries.com/dictionary/schizophrenia`, Zugriff 10.02.2013

[Duden] `http://www.duden.de/rechtschreibung/schizophren`, Zugriff 10.02.2013

[Wikipedia] `http://de.wikipedia.org/w/index.php?title=Synchronit%C3%A4t&oldid=104773894`, Zugriff 10.02.2013

[ScWe01] A new method for compiling schizophrenic synchronous programs

[ScWe02] Embedding Imperative Synchronous Languages in Interactive Theorem Provers

[Berry99] G.Berry. The constructive semantics of pure Esterel, Juli 1999