Synthese von Verhaltensbeschreibungen in VHDL mittels logischer Transformationen

Ramayya Kumar und Dirk Eisenbiegler
Forschungszentrum Informatik
(Prof. Dr.-Ing. D. Schmid)
Haid-und-Neu-Straße 10-14 76131 Karlsruhe
e-mail: {kumar,eisen}@fzi.de

1. Februar 1996

Zusammenfassung

In diesem Artikel soll beschrieben werden, wie aus synchronen VHDL-Verhaltensbeschreibungen mittels logischer Transformationen Schaltungsstrukturen auf RT-Ebene abgeleitet werden können. Das Syntheseverfahren verwendet als elementare Umformungsschritte ausschließlich Regelanwendungen der Prädikatenlogik höherer Ordnung (Theorembeweiser HOL). Die Synthese erfolgt vollständig automatisch.

1 Einleitung

Syntheseprogramme für höhere Entwurfsebenen gehen oft von Verhaltensbeschreibungen in VHDL aus. Aufgrund der Komplexität der VHLD-Semantik ist es jedoch nicht einfach, aus diesen Beschreibungen eine Implementierungsstruktur zu erzeugen, die korrekt ist. Unter Korrektheit soll folgendes verstanden werden: Es wird gefordert, daß die Simulation von Verhaltensbeschreibung und Strukturbeschreibung zu gleichen Ergebnissen führt. Diese Forderung erscheint elementar, sie wird von bestehenden Werkzeugen jedoch nicht immer erfüllt.

Eine klare und formale (Simulations-)Semantik von VHDL ist die Basis dafür, um überhaupt über VHDL-Schaltungen und deren Korrektheit logisch argumentieren zu können. In diesem Bereich existieren bereits zahlreiche Aktivitäten (siehe [KlBr95]), die sich vor allem darin unterscheiden, ob sie ganz VHDL oder nur eine Teilmenge berücksichtigen, und darin, welcher Formalismus zugrunde liegt. Beispielsweise bildet der Ansatz in [BLPV95] eine Anbindung von VHDL an einen Model-Checker, d.h. Temporallogik. Andere Ansätze basieren auf Prädikatenlogik höherer Ordnung, Petri-Netz-Darstellungen etc..

Der in diesem Artikel vorgestellte Ansatz widmet sich ausschließlich rein synchronen Schaltungen. Wir werden eine VHDL-Teilmenge namens $\mathcal{ABC}$-VHDL verwenden, in der aufgrund vorgegebener Beschränkungen nur eindeutige und rein synchrone Schaltungsbeschreibungen enthalten sind [EiKM95, EiKM96]. Die Semantik von $\mathcal{ABC}$-VHDL wurde in der Prädikatenlogik höherer Ordnung definiert. Es wurde ein Übersetzungsprogramm implementiert, mit dem $\mathcal{ABC}$-VHDL-Beschreibungen in Terme des Theorembeweisers HOL [GoMe93] automatisch übersetzt werden können. Da $\mathcal{ABC}$-VHDL-Programme in der Logik dargestellt werden, kann über sie auf logischer Ebene argumentiert werden. Es ist dadurch formal definiert, was es bedeutet, daß zwei Schaltungsbeschreibungen in $\mathcal{ABC}$-VHDL äquivalent sind.

Die logische Äquivalenz kann innerhalb des Kalküls mittels logischer Regelanwendungen bewiesen werden (Verifikation). Es ist andererseits auch möglich $\mathcal{ABC}$-VHDL-Programme mittels logischer Transformationen in äquivalente, implementierungsnähere Darstellungen zu überführen (Formale Synthese).

In diesem Artikel wird beschrieben, wie $\mathcal{ABC}$-VHDL-Programme mittels logischer Transformationen synthetisiert werden können. Der Syntheseablauf beginnt mit einer Verhaltensbeschreibung in $\mathcal{ABC}$-VHDL (Spezifikation). Das Ergebnis wird eine strukturelle Implementierung auf der RT-Ebene sein (Implementierung).

Im Gegensatz zu konventionellen Syntheseverfahren, wird also nicht nur die Implementierung zu einer Spezifikation konstruiert, sondern es wird auch gleichzeitig der Beweis der logischen Korrektheit der Implementierung (Ausgabe) bzgl. der Spezifikation (Eingabe) erbracht. In klassischen Post-Synthese-Verifikationsansansätze, wird zunächst eine Implementierung ,,irgendwie" erzeugt, um danach den Beweis zu ,,erraten". ,,Erraten" wird deshalb erforderlich, da man lediglich das Syntheseergebnis nicht aber den Syntheseweg kennt. Der Beweis erfolgt in unserem Ansatz im Fluß der Synthese durch logische Schritte, wodurch die Ableitung konstruktiv erzeugt wird und später nicht mehr erraten werden muß. Die Formale Synthese bietet vor allem zwei Vorteile: Zum einen ist es durch Formale Synthese bei weitem größere Schaltungen korrekt synthetisiert werden als Schaltungen in der Post-Synthese-Verifikation automatisch verifiziert werden können. Zum anderen ist es möglich, stärkere Logiken jenseits der Entscheidbarkeit sinnvoll und effizient einzusetzen. Dies ist vor allem auf höheren Entwurfsebenen relevant, wo sich Aussagenlogik und Temporallogik als zu ausdrucksschwach erweisen.

Einer der ersten Ansätze im Bereich der formalen Synthese widmete sich der Synthese einfacher regulärer Ausdrücke [John84]. Das System Dialog [MaFo91, AHL92] ist ein Beispiel für den Einsatz von Theorembeweisern in der Synthese. Dialog ist ein graphisches Entwurfswerkzeug, in dem Implementierungen durch logische Transformationsschritte in der Prädikatenlogik höherer Ordnung (Theorembeweiser: Lambda) abgeleitet werden. Dabei werden die in der graphischen Oberfläche durchgeführten Einfügeoperationen von Bausteinen auf Reduktionsschritte bzgl. der ursprünglichen Spezifikation abgebildet. Im Gegensatz zu diesen Techniken, die für allgemeine Bausteinentwürfe auf RT-/Gatterebene konzipiert sind, beschränkt sich der Ansatz in [BoJo93] zielgerichtet auf den Mikroprozessorentwurf und setzt sowohl Techniken der Verifikation als auch der schrittweisen Verfeinerungen ein.

In unserem Ansatz sollen keine neuen Syntheseverfahren speziell für die Formale Synthese entwickelt. Vielmehr soll das Know-how bestehender Syntheseverfahren ausgenutzt und auf eine formale Basis gestellt werden, wozu es natürlich erforderlich ist, entsprechende Schaltungsrepräsentation und elementare Schaltungsumformungen in der Logik zu finden.

2 Eine kurze Einführung in $\mathcal{ABC}$-VHDL

Allgemeine VHDL-Beschreibungen stellen nicht notwendigerweise synchrone Schaltungen dar. Wenn es beispielsweise bei der Ausführung von VHDL-Programmen zu Laufzeitfehlern kommt oder wenn unendliche Schleifen ohne wait-Befehl erreicht werden, dann repräsentieren diese VHDL-Programme keine reale Schaltung, geschweige denn synchrone Schaltungen. Nur in seltenen Fällen wird durch ein VHDL-Programm auch wirklich eine eindeutige Beziehung zwischen Ein- und Ausgangssignalen definiert.

$\mathcal{ABC}$-VHDL ist eine Teilmenge von VHDL mit Einschränkungen, die gewährleisten, daß innerhalb $\mathcal{ABC}$-VHDL nur eindeutige, synchrone Schaltungsbeschreibungen möglich sind. In $\mathcal{ABC}$-VHDL beziehen sich wait-Anweisungen immer auf den Takt. Befehle[1] sind in drei Klassen aufgeteilt: $\mathcal{A}$, $\mathcal{B}$ und $\mathcal{C}$. Anweisungen vom Typ $\mathcal{A}$ enthalten keine wait-Anweisungen. Anweisungen vom Typ $\mathcal{B}$ und $\mathcal{C}$ enthalten wait-Anweisungen. Für Anweisungen vom Typ $\mathcal{C}$ wird gewährleistet, daß die Anweisung nicht in einem Takt vollständig evaluiert werden kann. Ausgehend vom Prozeßanfang immer erst eine wait-Anweisung erreicht wird, bevor der Prozess zum Ende kommt. Anweisungen vom Typ $\mathcal{B}$ müssen diese Eigenschaft nicht erfüllen. In $\mathcal{ABC}$-VHDL ist genau definiert, wie atomare Anweisungen (wait-Anweisungen, Signalzuweisungen, Variablenzuweisungen) mittels Kontrollstrukturen (Sequenzen, if-then-else-Anweisungen, while-Schleifen) in rekursiver Weise zu komplexen Anweisungen zusammengesetzt werden können, und wie sich der Typ zusammengesetzter Anweisungen aus dem Typ der Teile bestimmen läßt. In $\mathcal{ABC}$-VHDL gibt es eine wichtige Einschränkung: While-Schleifen sind nur für Schleifenrümpfe vom Typ $\mathcal{C}$ erlaubt. Mit dieser Einschränkung wird sichergestellt, daß unendliche Schleifen vermieden werden.

Die Semantik von $\mathcal{ABC}$-VHDL ist konform mit der Semantik von VHDL. Da jedoch ausschließlich synchrone Schaltungen beschrieben werden, ist das der Formalisierung zugrundeliegende Modell einfacher und damit einer logischen Argumentation leichter zugänglich. In $\mathcal{ABC}$-VHDL werden Prozesse formal durch Automaten-Beschreibungen in Form von Paaren bestehend aus einer kombinierten Aus- und Übergangsfunktion und einem Initialzustand beschrieben. Die Aus- und Übergangsfunktion bildet die aktuelle Eingabe, den aktuellen Kontrollzustand, den aktuellen Variablenzustand und den aktuellen Ausgangssignalzustand auf den neuen Ausgangssignal-, Kontroll- und Variablenzustand ab. Der gesamte Ansatz basiert auf einer in HOL formalisierten Theorie namens Automata. In dieser Theorie wird die Beziehung zwischen einer Automatenbeschreibung und ihrem Ein/Ausgabeverhalten formal definiert. Außerdem enthält diese Theorie abgeleitete Theoreme, die synthesespezifische Äquivalenzumformungen auf Automaten wie Zustandsminimierung, Zustandskodierung und Retiming in allgemeiner Form beschreiben (siehe [EiKu95]).

3 Der Ablauf der formalen Synthese

Ausgangspunkt ist eine Verhaltensbeschreibung in Form eines $\mathcal{ABC}$-VHDL-Prozesses. Als Ergebnis wird eine Schaltungsstruktur abgeleitet, die sich aus Operationseinheiten, Multiplexern und D-Flipflops zusammensetzt. Die Abbildungen 1, 2 und 3 beschreiben das Verfahren an einem Beispiel.

Die Eingabe ist ein VHDL-Program (Abbildung 1). Um logische Transformationen überhaupt durchführen zu können, muß diese Darstellung zunächst in einen logischen Ausdruck umgewandelt werden. Dies geschieht vollständig automatisch durch den $\mathcal{ABC}$-VHDL-Parser, der diesen Programmtext in logische Terme des Theorembeweisers HOL umwandelt.

Das Ergebnis dieser Umwandlung ist in der unteren Hälfte von Abbildung 2 (Spezifikation) dargestellt.[2] Das in diesem Artikel vorgestellte formale Syntheseverfahren bildet diesen Eingabeterm (Spezifikation) auf ein Theorem ab, das besagt, daß eine bestimmte Implementierung — die erst während der Synthese entsteht — die Spezifikation erfüllt (d.h.: diese impliziert) oder mit ihr äquivalent ist. Im allgemeinen stellt sich eine formale Synthese als

[1] in VHDL-Terminologie: sequential statements

[2] Die Nummern in eckigen Klammern verweisen auf die entsprechenden Zeilen des Programmtextes

```
1  entity gcd is
2    port (
3      clk : in std_logic;  a,b : in integer;  start : in std_logic;
4      ready : out std_logic := '1';  result : out integer := 0
5    );
6  end gcd;
7
8  architecture behavior of gcd is
9    begin process
10     variable x,y,z : integer;
11   begin
       ●————————————————————————— state0
12     while start /= '1' loop
13       wait until clk = '1'; ●————————— state1
14     end loop;
15     ready <= '0';
16     if (a < b) then
17       x := b;
18       y := a;
19     else
20       x := a;
21       y := b;
22     end if;
23     while (y /= 0) loop
24       z := x - y;
25       wait until clk = '1'; ●————————— state2
26       x := y;
27       y := z;
28     end loop;
29     ready <= '1';
30     result <= x;
31     wait until clk = '1'; ●————————— state3
32   end process;
33 end behavior;
```

Abbildung 1: Verhaltensbeschreibung der GCD-Schaltung in $\mathcal{ABC}$-VHDL

eine Folge von Äquivalenzumformungen und Verfeinerungsschritten dar. In unserem Beispiel wurden ausschließlich Äquivalenzumformungen durchgeführt. Abbildung 2 stellt als Ganzes betrachtet das Ergebnis der formalen Synthese dar: eine Äquivalenz zwischen einer Spezifikation (der Eingabe) und einer während der Synthese erzeugten Implementierung als ein abgeleitetes Theorem.

In Abbildung 3 ist die Implementierung der Beispielschaltung, eine strukturelle Beschreibung auf RT-Ebene, graphisch dargestellt. Die Implementierung besteht aus einer Aus- und Übergangsfunktion und einer Registereinheit. Die Registereinheit speichert die . Die Registereinheit speichert das Tripel $((ready, result), (y, z), control)$ bestehend aus Ausgangssignalen, Variablen- und Kontrollzustand. Der initiale Wert ist $(('1', 0), (inity, initz), \mathsf{state0})$. Da in der Spezifikation für die Variablen y und z keine initialen Werte definiert worden waren, wurden bei der Konvertierung zu HOL variable Werte $initx$ und $inity$ verwendet.

3.1 Fallunterscheidung über Kontrollzustände

In $\mathcal{ABC}$-VHDL gibt es zwei Arten von Verhaltensbeschreibungen: Prozesse mit zumindest einer wait-Anweisung und Prozesse ohne wait-Anweisung. Prozesse der ersten Gruppe eignen sich zur Beschreibung sequentieller Schaltungen, bei denen die wait-Anweisungen stets sensitiv bzgl. des Taktsignals sind und die Signalwerte, die auf die Ausgänge geschrieben werden, um einen Taktzyklus verzögert werden. Die Schaltungen der ersten Gruppe sind

```
⊢ automaton (
    (λ((a, b, start), (ready, result, y, z, control)).
      (
        (ready, result),
        let  e1 = a < b   in
        let  e2 = a /= 0   in
        let  e3 = start /= '1'   in
        let  e4 = b - a   in
        let  e5 = b /= 0   in
        let  e6 = a - b   in
        let  e7 = z /= 0   in
        let  e8 = y - z   in
        let  r1 = (ready, result, z, e8, state2)   in
        let  r2 = ('1', y, z, z, state3)   in
        let  r3 = ('0', result, a, e4, state2)   in
        let  r4 = ('1', b, a, z, state3)   in
        let  r5 = ('0', result, b, e6, state2)   in
        let  r6 = (ready, result, y, z, state1)   in
        let  r7 = ('1', a, b, z, state3)   in
        let  c1 = (control = state2)   in
        let  t1 = if e2 then r3 else r4    in
        let  t2 = if e5 then r5 else r7    in
        let  t3 = if e1 then t1 else t2    in
        let  t4 = if e3 then r6 else t3    in
        let  t5 = if e7 then r1 else r2    in
        let  t6 = if c1 then t5 else t4    in
          t6
      ) ),
    ('1', 0, inity, initz, state0)
  )                                                                  } Implementierung
  =
  processC  (('1', 0), (initx, inity, initz))  (
[12,13,14]  whileC  (λ((a, b, start), (x, y, z)).  (start /= '1')) wait   seqBC
[15]        sigassign(λ((a, b, start), (ready, result), (x, y, z)). ('0', result))   seqAC
[16]        ifteAA  (λ((a, b, start), (x, y, z)). a < b)
            (
[17]          varassign(λ((a, b, start), (x, y, z)). (b, y, z))   seqAA
[18]          varassign(λ((a, b, start), (x, y, z)). (x, a, z))
            )
            (
[20]          varassign(λ((a, b, start), (x, y, z)). (a, y, z))   seqAA
[21]          varassign(λ((a, b, start), (x, y, z)). (x, b, z))
            )   seqAC
[23]        whileC  (λ((a, b, start), (x, y, z)).  (y = 0))
            (
[24]          varassign(λ((a, b, start), (x, y, z)). (x, y, x − y))   seqAC
[25]          wait   seqCA
[26]          varassign(λ((a, b, start), (x, y, z)). (y, y, z))   seqAA
[27]          varassign(λ((a, b, start), (x, y, z)). (x, z, z))
            )   seqBC
[29]        sigassign(λ((a, b, start), (ready, result), (x, y, z)). ('1', result))   seqAC
[30]        sigassign(λ((a, b, start), (ready, result), (x, y, z)). (ready, x))   seqAC
[31]        wait
  )                                                                  } Spezifikation
```

Abbildung 2: Äquivalenz zwischen Implementierung und Spezifikation

deshalb immer Moore-Schaltwerke, also Schaltungen, bei denen die Eingabe nicht direkt die Ausgabe beeinflußt. Die zweite Gruppe beschreibt Schaltungen, die rein kombinatorisch sind mit der Ausnahme, daß die Ausgänge eventuell gepuffert werden müssen. Bei Prozessen der zweiten Gruppe müssen alle Eingangssignalwerte in der sensitivity-Liste aufgeführt sein. Die Ausgangssignale müssen immer dann gepuffert werden, wenn in einem Taktzyklus das

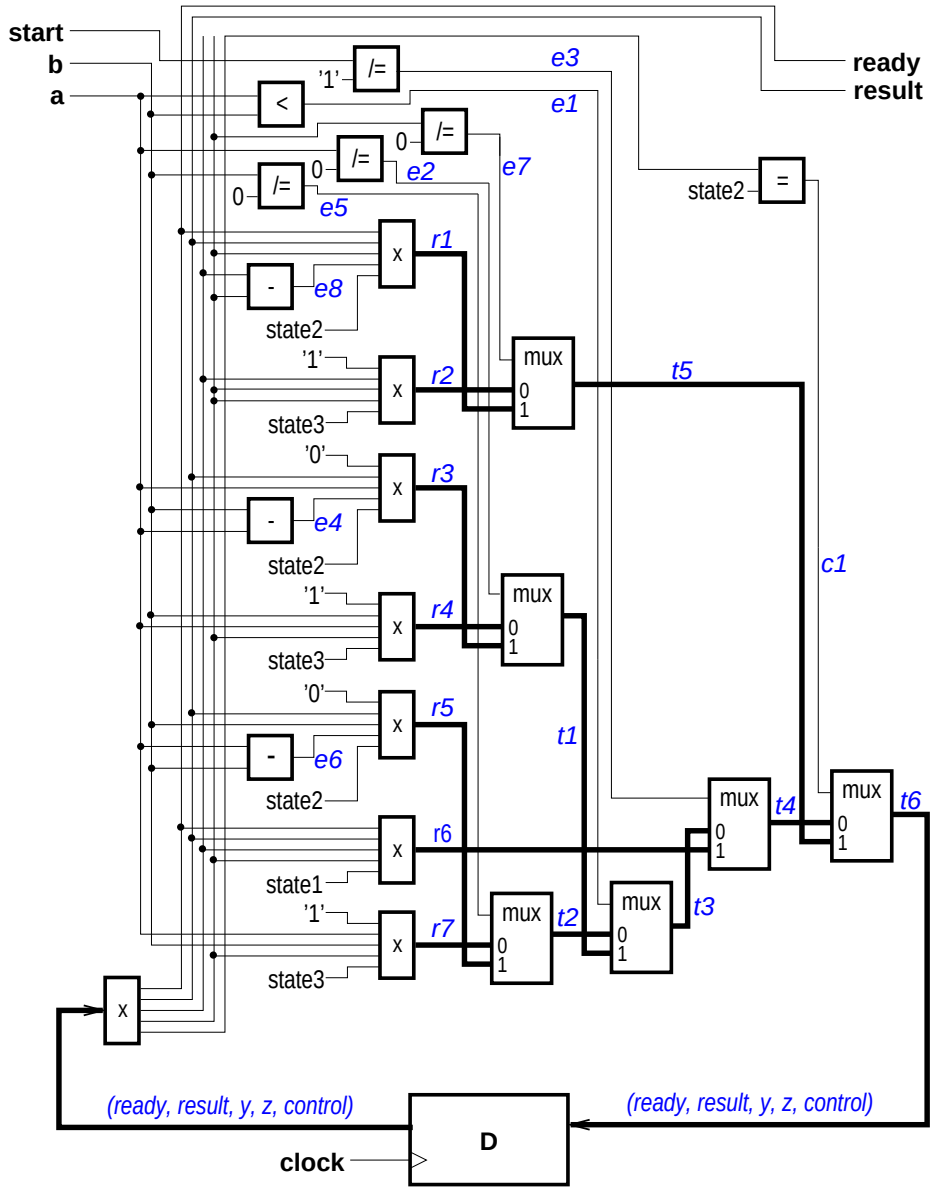

Abbildung 3: Implementierung der GCD-Schaltung

Ausgangssignal keine Signalzuweisung erhält. In diesem Fall müssen die alten Ausgangssignalwerte wiederverwendet werden.

Das Beispiel in Abbildung 1 gehört zur ersten Gruppe. Es enthält drei wait-Anweisungen, die mit `state1`, `state2` und `state3` bezeichnet wurden. Jede wait-Anweisung entspricht einem Kontrollzustand der Schaltung. Zusätzlich wird ein Kontrollzustand für den Startzustand `state0` benötigt. Die Schaltung beginnt im Takt 0 im Initialzustand `state0` und springt in den nachfolgenden Takten von einer wait-Anweisung zur nächsten. Der Initialzustand `state0` wird später nicht wieder erreicht.

Im ersten Schritt des Syntheseverfahrens wird eine Fallunterscheidung über den Kontroll-

zuständen state0, state1, state2, ... durchgeführt — vorausgesetzt natürlich, daß überhaupt Kontrollzustände existieren, d.h., daß der zu synthetisierende Prozeß zur ersten Gruppe gehört. Prozesse der zweiten Gruppe bedürfen keiner Kontrolleinheit. Es existiert lediglich ein einziger Kontrollzustand am Anfang des Programms. An dieser Position wartet der Prozeß darauf, daß sich die Eingabesignale des Programms ändern, dann durchläuft die Evaluierung das gesamte Programm, erreicht das Ende und wartet dann wieder am Anfang, bis sich wieder Eingangssignale ändern. Bei Prozessen der zweiten Gruppe wird dieser erste Schritt (Fallunterscheidung über Kontrollzustände) ausgelassen, die Synthese beginnt mit dem nächsten Schritt.

3.2 Symbolische Evaluierung

Im nächsten Syntheseschritt werden symbolische Evaluierungen durchgeführt. Dies geschieht separat für jeden Kontrollzustand state*n* als Ausgangspunkt. Das Ergebnis hängt jeweils davon ab, welchen Kontrollpfad der Prozeß ausgehend von state*n* durchläuft, bis er wieder auf eine wait-Anweisung stößt. In $\mathcal{ABC}$-VHDL gibt es nur drei verschiedene Kontrollstrukturen: Sequenzen von Anweisungen, if-then-else-Strukturen und while-Schleifen. Der Kontrollfluß wird durch die Werte der Bedingungen in den if-then-else-Strukturen und in den while-Schleifen bestimmt.

Durch die symbolische Simulation wird genau ein Taktzyklus abgearbeitet. Das Ergebnis ist ein Tripel bestehend aus Ausgangssignal, neuen Variablenwerten und neuem Kontrollzustand. Sowohl das Ergebnistripel als auch die Bedingungen, die bei der Evaluierung durchlaufen werden, hängen vom aktuellen Wert der Eingangssignale, den aktuellen Variablenwerten und den aktuellen Werten der Ausgangssignale ab. Ergebnistripel und Bedingungen sind symbolische Ausdrücke, in denen die Variablen und Eingangssignalwerte durch VHDL-Operationen wie beispielsweise `+`, `/=` und `-` miteinander verknüpft werden.

Das bisher erreichte Ergebnis ist ein Entscheidungsbaum, dessen Pfade die Kontrollpfade des $\mathcal{ABC}$-VHDL-Prozesses repräsentieren. Am Wurzelknoten erfolgt eine Fallunterscheidung über den (Ausgangs-)Kontrollzuständen. Unterhalb befinden sich Knoten, die den Verzweigungen bzgl. der passierten Bedingungen beschreiben. Die Blätter beschreiben in symbolischer Weise das Ergebnistripel. Abbildung 4 zeigt einen Kontrollpfad innerhalb des Entscheidungsbaums, der aus unserem Beispiel abgeleitet wurde. Der Kontrollpfad beginnt in state1 und durchläuft dann einen Kontrollpfad der drei Bedingungen passiert und schließlich im Kontrollzustand state2 endet.

Anmerkung zu Abbildung 4: Nur die Eingangssignale und die alten Variablen- und Ausgangswerte vor dem Taktzyklus wurden verwendet. Signal- und Variablenzuweisungen führen zu Substitutionen. Beispiel: Da der Prozeß die Zuweisung `y := a;` durchläuft, wird nachfolgend `y` durch `a` substituiert. Das bedeutet beispielsweise, daß der nachfolgende Ausdruck `y /= 0` in `a /= 0` überführt wird. Aus diesem Grunde erhält man für den beschriebenen Kontrollpfad als Ergebnis

$$(('0', result), (b, a, b - a), \mathsf{state2})$$

anstelle von

$$((ready, result), (x, y, z), \mathsf{state2})$$

Das Ergebnis, das bisher erreicht wurde, stellt bereits eine Implementierung auf RT-Ebene dar. Der Entscheidungsbaum repräsentiert eine Struktur, in der neben arithmetischen

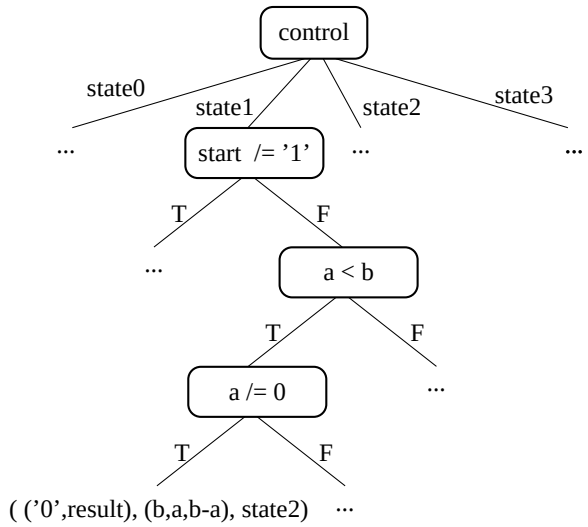

Abbildung 4: Entscheiungsbaum zur Bestimmung von ((ready,result),(x,y,z),control)

Operationseinheiten lediglich Multiplexer für die Knoten und D-Flipflops für die Speicherung der Variablen-, Ausgangssignal- und Kontrollzustände benötigt werden.

Syntheseverfahren wie diese symbolische Evaluierung sind nicht neu in dem Sinne, daß es nicht bereits Programme in C oder Pascal gäbe, die vergleichbare Techniken während der Synthese durchführen. Der grundlegende Fortschritt in unserem Ansatz besteht darin, daß unser Ansatz auf sicheren logischen Transformationen in der Logik basiert. Dadurch wird gewährleistet, daß unser Syntheseprogramm, ganz gleich wie es im Detail aufgebaut ist und wie komplex es auch ist, stets nur Implementierungen als Syntheseergebnis liefert, die korrekt bzgl. der Spezifikation sind. Hierin unterscheidet sich dieser Ansatz grundlegend von konventionellen Ansätzen, in denen die Korrektheit der Schaltungsimplementierungen von der Korrektheit großer Syntheseprogramme abhängt. Heutige Syntheseprogramme mit ausgefeilten Synthese- und Optimierungsschritten sind aufgrund ihrer Komplexität nur noch schwer zu beherrschen. Fehler bei den Umformungen und Mißverständnisse bei der Semantik der verwendeten Zwischenformate können zu Syntheseergebnissen führen, die nicht mehr die Eingabespezifikation erfüllen.

3.3 Optimierungen auf RT-Ebene

Viele der Ausdrücke und Teilausdrücke, die in den Bedingungen und Blättern des Entscheidungsbaumes verwendet werden, sind gleich, und trotzdem treten sie wiederholt auf. Die Ausdrücke und Teilausdrücke werden deshalb Schritt für Schritt durch Abkürzungen substituiert. Die entsprechenden Variablen werden mit $e1$, $e2$, ... bezeichnet (siehe Abbildung 2). Gleiche Ausdrücke werden so nur einmal berechnet und das Ergebnis einmal oder eventuell mehrfach wiederverwendet.

Bisher wurden alle Variablen, die in $\mathcal{ABC}$-VHDL-Programmen verwendet werden, auf Register abgebildet. Immer dann, wenn Teile des Speichers keinen direkten Einfluß auf den Ausgang und auf andere Speicherteile haben, können diese jedoch eingespart werden. Um

dies zu erkennen, müssen die Abhängigkeiten analysiert werden. In unserem Beispiel zeigt sich, daß das Register, das der Variablen x zugeordnet wurde, entbehrlich ist. Nachdem diese redundanten Speicherteile erkannt wurden, wird im Syntheseprogramm der Speicher zunächst in einen redundanten und einen nichtredundanten Teil aufgespalten. In einem zweiten Schritt wird dann eine Tranformation der Theorie Automata (siehe [EiKu95]) zur Eliminierung solcher redundanter Speicherteile angewandt.

Innerhalb des Entscheidungsbaumes kann es sein, daß einige der Teilbäume gleich sind. Dadurch, daß in einem weiteren Optimierungsschritt alle Blätter und Teilbäume abgekürzt werden, wird die Anzahl der benötigten Multiplexer reduziert. In unserem Beispiel wurden die verschiedenen Blätter durch $t1$, $t2$, ... und die Teilbäume durch $r1$, $r2$, ... abgekürzt. In diesem Optimierungsschritt wird ein weiteres Potential ausgenutzt: Die Reihenfolge, in der die Bedingungen im Baum erscheinen, kann durch geeignete Umformungen vertauscht werden. Den Bedingungen wird durch das Syntheseverfahren eine feste, d.h. in allen Zweigen gleiche, Reihenfolge gegeben. Dadurch, daß zum einen die Teilbäume abgekürzt und wiederverwendet werden und andererseits eine feste Reihenfolge für die Bedingungen vorgegeben wird, wandelt sich der Entscheidungsbaum in ein (symbolisches) Entscheidungsdiagramm. Äquivalente Teilgraphen, bei denen nur die Reihenfolge der Bedingungen verschieden war, werden nun echt gleich. Knoten bei denen alle Teilgraphen gleich sind werden durch einfache logische Umformungen eliminiert.

Diese Art, Teilgraphen zu eliminieren, scheitert in der Regel im Wurzelknoten (Mehrfach-Fallunterscheidung über dem Anfangskontrollzustand). Auch da können jedoch i.a. mehrere identische direkte Teilgraphen sein — i.a. sind jedoch nicht alle identisch. Verzweigung in Gruppen von Kontrollausgangszuständen mit identischen Teilgraphen überführt. In unserem Beispiel entsteht eine Zweifachverzweigung mit der Bedingung `(control = state2)`, da alle direkten Teilgraphen des Wurzelknotens außer dem für `(control = state2)` identisch sind.

3.4 Weitere Syntheseschritte

Es gibt noch zahlreiche weitere Synthese- und Optimierungsschritte, die auf diese Schaltungsbeschreibung angewandt werden könnten. Um zu rein booleschen Signalen und Zustandsspeichern in der Kontrolleinheit zu kommen, muß der Kontrollzustandsdatentyp kodiert werden. Ferner kann die Anzahl der Kontrollzustände reduziert werden, indem zum einen unerreichbare Zustände eliminiert und zum anderen Zustände mit gleichem Verhalten zusammengefaßt werden. Ein weiteres Betätigungsfeld für Optimierungen ist Retiming, wobei durch das Verschieben von Registern die Verzögerungszeit des Aus- und Übergangsschaltnetzes reduziert und damit die Taktfrequenz erhöht werden kann. Für Zustandskodierung, Zustandsminimierung und Retiming existieren in der Theorie Automata [EiKu95] bereits vorbewiesene Umformungstheoreme.

4 Zusammenfassung und Ausblick

Es wurde ein Ansatz zur *korrekten* Synthese synchroner VHDL-Schaltungen vorgestellt, bei dem sich alle Teilschritte aus logischen Umformungen zusammensetzt. Als Ergebnis wird damit nicht nur eine Implementierung erzeugt, sondern es wurde auch ein Theorem abgeleitet, das besagt, daß die Implementierung bzgl. der Spezifikation korrekt ist. Alle Syntheseschritte wurden vollständig automatisch durchgeführt.

Mit dem in diesem Artikel vorgestellten Syntheseverfahren haben wir die Spanne zwischen Verhaltensbeschreibungen in VHDL und Strukturbeschreibungen auf RT-Ebene überbrückt. Der aktuelle Stand des Synthesewerkzeugs erlaubt noch keine Ableitung von Schaltungsbeschreibungen auf boolescher Ebene. Dazu müssen zunächst noch die Datentypen der Signale und Speicher durch boolesche Werte kodiert und die entsprechenden booleschen Operationseinheiten abgeleitet werden. Dies wird das Thema weiterer Arbeiten sein.

Literatur

[AHL92] AHL. *Lambda Reference Manual*, 1989.

[BGGH92] R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experiences with Embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R. Boute, editors, *Conference on Theorem Provers in Circuit Design*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.

[BLPV95] Jörg Bergmann, Jörg Lohse, Michael Payer, and Gerd Venzel. Model checking in industrial hardware design. In *DAC '95*, 1995.

[BoJo93] B. Bose and S.D. Johnson. DDD-FM9001:derivation of a verified microprocessor. In *CHARME93*, number 683 in Lecture Notes in Computer Science, pages 191–202, Arles,France, May 1993. Springer Verlag.

[EiKM95] D. Eisenbiegler, R. Kumar, and J. Müller. Formalizing the semantics for a synchronous subset of VHDL. Technical Report FZI-Report 8/95, Forschungszentrum Informatik (FZI), 1995.

[EiKM96] D. Eisenbiegler, R. Kumar, and J. Müller. A formal model for a VHDL subset of synchronous circuits. In *APCHDL '96*, Bangalore, India, 1996.

[EiKu95] D. Eisenbiegler and R. Kumar. An automata theory dedicated towards formal circuit synthesis. In *Higher Order Logic Theorem Proving and Its Applications*, Aspen Grove, Utah, USA, September 1995. Springer.

[GoMe93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[John84] S. Johnson. Synthesis of Digital Designs from Recursion Equations. MIT Press, 1984.

[KlBr95] C.D. Kloos and P.T. Breuer, editors. *Formal Semantics for VHDL*, volume 307 of *The Kluwer international series in engineering and computer science*. Kluwer, Madrid, Spain, March 1995.

[MaFo91] E.M. Mayger and M.P. Fourman. Integration of formal methods with system design. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 59–70, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.