Universität Karlsruhe
Fakultät für Informatik
Institut für Rechnerentwurf und Fehlertolearnz

Diplomarbeit

Eine synchrone generische Hardwarebeschreibungssprache für die Verifikation mit induktiven Verfahren sowie Modellierung temporaler Logik

von

Gürşad Küçük

Matrikelnummer:	673763
Fachsemester:	14
Adresse:	76131 Karlsruhe, Waldhornstr.4

30. Juni 1997

Verantwortlicher Betreuer:	Prof. Dr.–Ing. D. Schmid
Betreuer am Institut:	Dr. Klaus Schneider

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Motivation 5
1.2 Aufgabenstellung 6
1.3 Überblick über die einzelnen Kapitel 7

2 Stand der Technik **8**
2.1 LUSTRE 10
2.2 SML 12
2.3 HOL 14
2.4 VHDL 15
2.5 Vergleich der Beschreibungssprachen 19

3 GHDL **21**
3.1 Datentypen 22
3.2 Spezifikationssprache 22
3.2.1 Lineare temporale Logik 23
3.3 Implementierungsbeschreibung 25
3.3.1 BASIC 26
3.3.2 MODULE 27
3.3.3 RECURSIVE 30
3.3.4 GENERIC 37
3.3.5 INSTANCE 38

4 Benchmark-Schaltungen **42**
4.1 Island Tunnel Controller 42
4.2 Arbiter 44
4.3 Safe-Box 46

5 Ausblick **49**

A GHDL-Programme zu Kapitel 4 **54**
A.1 Island Tunnel Controller 54
A.2 Arbiter 58
A.3 Safe-Box 60

B Grammatik **62**

Abbildungsverzeichnis

2.1 Struktur eines Halbaddierers 9
2.2 1-Bit Volladdierer 10
2.3 Schaltbild eines Carry-Ripple-Addierers 11
2.4 And-Gatter mit 2 Eingängen 16
2.5 Vergleich der Beschreibungssprachen 20

3.1 Arbeitsweise von GHDL 22
3.2 Addierer mit Reset und Ready 25
3.3 Hardwaretypen im Überblick 25
3.4 Schaltbild und -symbol eines 1-Bit-Halbaddierers 27
3.5 Schaltbild eines 1-Bit-Volladdierers 29
3.6 Schaltsymbol eines 1-Bit-Volladdierers 30
3.7 Schaltbild eines Carry-Ripple-Addierers 30
3.8 Schalterstellungen von Zweierschaltern 33
3.9 Mischungspermutation für 8 Ein- und Ausgänge 34
3.10 Speichergekoppeltes Omega-Netzwerk 35
3.11 Beziehung zwischen den Hardwaretypen 39
3.12 Instantiierung 40
3.13 Beschreibungssprachen im Vergleich 41

4.1 Island Tunnel Controller 44
4.2 Implementierung eines Arbiters 45
4.3 Verkettete Arbiter-Stufen 46
4.4 Safe-Box-Zustandsdiagramm für 4 Knöpfe 48
4.5 Interne Zellendarstellung 48

Kapitel 1

Einleitung

1.1 Motivation

Durch die generische[1] Darstellung der Hardwarestrukturen kann viel Zeit und Geld gespart werden. Bei der generischen Darstellung von Hardwarestrukturen nutzt man zwei Eigenschaften aus. Die erste Eigenschaft ist, daß viele Hardwarestrukturen nur in der Anzahl der Eingänge variieren. Dabei bleibt die jeweilige Funktion gleich. Zum Beispiel realisiert ein Und-Gatter mit 2 Eingängen die gleiche Funktion wie mit 4 Eingängen. Die zweite Eigenschaft ist, daß es Hardwarestrukturen gibt, die einen regulären Aufbau haben. Zu diesen Hardwarestrukturen gehört auch der Carry-Ripple-Adder, der aus mehreren gleichen 1-Bit Volladdierern besteht, die regelmäßig miteinander verschaltet sind.

Durch die Verwendung von generischen Hardwarebeschreibungen wird erreicht, daß nur einmal entworfen und einmal verifiziert wird. Indem man die generische Hardwarebeschreibung durch Festlegen der Parameter spezialisiert, umgeht man das Dilemma, für jede Neuerung vieles noch einmal zu entwerfen und zu verifizieren. Der Lohn dafür ist eine eine höhere Automatisierung und bessere Wiederverwendung bei der Herstellung. Bei der Verifikation kommt es vor, daß konkrete Beweise mit entscheidbaren Methoden für bestimmte Bitbreiten nicht möglich sind, so daß mit der generischen Hardwarebeschreibung ein generischer Beweis gemacht werden kann.

In dieser Diplomarbeit wird eine generische Hardwarebeschreibungssprache GHDL (Generic Hardware Description Language) entwickelt, welche nicht als Erweiterung oder Aktualisierung von Hardwarebeschreibungssprachen wie z.B. VHDL(VHSIC Hardware Description Language, wobei VHSIC für Very High Speed IC steht) oder ähnliche zu verstehen ist. Einen Vergleich mit anderen Sprachen findet man in Kapitel 3.

[1] Generisch bedeutet hier, daß die Hardwarebeschreibung Parameter enthält.

1.2 Aufgabenstellung

Mit der in dieser Diplomarbeit entwickelten generischen Hardwarebeschreibungssprache soll es möglich sein, Hardwarestrukturen mit Parametern beschreiben zu können. Ferner soll durch diese Diplomarbeit der Grundstein für die Verifikation der Hardwarebeschreibung mittels Modell-Checker und Termersetzer [9] gelegt werden.

GHDL (Generic Hardware Description Language) baut bei der Implementierungsbeschreibung auf fünf verschiedenen Modultypen auf, die die Objekte darstellen. Eigenschaften, die das Verhalten der Objekte beschreiben, können zusätzlich angegeben werden. In der Praxis kommt es immer wieder vor, daß Prozeßmodelle und Datenmodelle durch eigene Notationen und Arbeitsmethoden während der Entwicklung scheinbar ein für allemal inkompatibel werden. Aufgrund dieser Kluft wurde eine Methode (Notation und Vorgehensweise) entwickelt, die dem Entwerfer zuerst das notwendige Verständnis für das Aufgabengebiet vermittelt. Erst danach werden - im Rahmen dieses soliden Verständnisses - die Anforderungen an das Verhalten (die Funktionen) erarbeitet. Diese Idee wurde praktisch an Beispielen getestet und zu einer systematischen Methode ausgebaut. Die saubere Trennung zwischen den Implementierungsbeschreibungen gehört hier dazu.

Es folgt eine Liste der wichtigsten Gründe und Vorteile für die Entwicklung von GHDL:

1. *Zielorientierte Darstellung von Hardwarestrukturen.* GHDL legt besonderen Wert auf den Hardwarebezug. Es ist relativ einfach eine (generische) Hardwarestruktur als Programmstruktur zu beschreiben.

2. *Zusammenarbeit zwischen Verifikation und Entwurf verbessern.* GHDL strukturiert die Beschreibung (Entwurf) und die Spezifikation (Verifikation), die in der Struktur von GHDL schon vorhanden sind.

3. *Zusammengehörigkeit explizit darstellen.* GHDL verwendet das Prinzip der Ganzheit, um die Zusammengehörigkeit von hardwaretechnischen Aspekten (Hardwarestrukturen) mit der temporalen Logik (Spezifikation) darzustellen und auszunutzen.

4. *Stabile Spezifikationen schaffen.* Durch die Orientierung an fachlichen Begriffen des Anwendungsgebiets (Entwurf und Spezifikation von Hardwarestrukturen) packt GHDL Informationen prägnant zusammen und reduziert die Auswirkungen einer ungenauen und vagen Formulierung.

1.3 Überblick über die einzelnen Kapitel

Die Ausarbeitung der Diplomarbeit besteht aus fünf Kapitel und zwei Anhängen. Die einzelnen Kapitel und Anhänge sind wie folgt gegliedert:

- **Kapitel 1:** Einführung in die Arbeit.
- **Kapitel 2:** Es wird der Stand der Technik bei Beschreibungssprachen vorgestellt. Dabei werden zahlreiche Beispiele vorgeführt. Standardbeispiel ist der Carry-Ripple-Addierer. Abschließend wird ein Vergleich gestellt auf Programmumfang, Verständlichkeit, Hardwarebezug, Zeitmodellierung und auf den generischen Aspekt hin.
- **Kapitel 3:** Hier werden die in GHDL vorkommenden Objekte und Modultypen im einzelnen vorgestellt. Im weiteren wird auf die Arbeitsweise von GHDL eingegangen. Abschließend wird ein Vergleich zwischen der erstellten generischen Hardwarebeschreibungsprache und den anderen ausgewählten Sprachen gestellt.
- **Kapitel 4:** In diesem Kapitel befinden sich zahlreiche Benchmark-Schaltungen mit denen GHDL getestet wurde.
- **Kapitel 5:** Ein Ausblick auf weitere Arbeiten.
- **Anhang A:** Enthält die GHDL-Programme zu Kapitel 4.
- **Anhang B:** Enthält die Grammatik zu GHDL.

Kapitel 2

Stand der Technik

Es gibt für die unterschiedlichsten Anwendungsgebiete zahlreiche Sprachen. Am verbreitetsten sind Programmiersprachen wie C [?] oder SML [23], die in verschiedenen Bereichen zur Softwareentwicklung eingesetzt werden. Im Logikbereich wird z.B. die Programmiersprache PROLOG [29] für die Prädikatenlogik 1.Stufe und HOL [12] für die Logik höherer Ordnung angewendet. Dabei stellt HOL einen Theorembeweiser dar, mit dessen Hilfe logische Aussagen interaktiv am Rechner bewiesen werden können. Andererseits gibt es auch synchrone Programmiersprachen wie ESTEREL [14] und LUSTRE [24], die über eine globale Uhr gesteuert bei jedem Takt eine Menge von Eingaben erhalten und eine Menge von Ausgaben berechnen. Weiterhin gibt es Sprachen, die in der Hardwareentwicklung angewendet werden. Zu diesen hardwarespezifischen Sprachen gehören VERILOG [30] und VHDL [17].

Um eine Aussage über die Anwendbarkeit bzw. Tauglichkeit von aktuellen Sprachen in der generischen Hardwareentwicklung und -verifikation machen zu können werden im folgenden stellvertretend LUSTRE, SML, HOL und VHDL vorgestellt. LUSTRE wurde dazu genommen, weil in LUSTRE generische Beschreibungen erlaubt sind. In HOL ist zudem die Verifikation generischer Hardwarestrukturen realisierbar. SML ist eine funktionale Programmiersprache, die im Prinzip keinen direkten Bezug zu Hardwarestrukturen hat. Schließlich wurde VHDL dazu genommen, weil VHDL sich in der Industrie durchgesetzt hat. Durch die ausgewählten Sprachen werden jeweils verschiedene Aspekte abgedeckt, die voneinander unabhängig in unterschiedlichen Bereichen angewendet werden.

Im folgenden werden diese Sprachen anhand einiger Kriterien miteinander verglichen. Die Kriterien sind:

- **Programmumfang**: Hier ist in erster Linie die Menge von Programmcode gemeint, die nötig ist um eine (generische) Hardwarestruktur zu beschreiben.

- **Verständlichkeit**: Ausschlaggebend ist hier die notwendige Zeit, die eine Person braucht, um die Programme zu verstehen. Es ist auch wichtig, wie umfangreich mit wenigen Befehlen die Hardwarestrukturen beschrieben werden können.

- **Hardwarebezug**: Es soll dem Benutzer der Programmiersprache leicht fallen, aus Hardwarestrukturen die entsprechenden Programmstrukturen zu entwerfen. Natürlich muß es auch leicht sein, aus Programmstrukturen die Hardwarestrukturen zu finden.

- **Zeitmodellierung**: Mit diesem Aspekt soll überprüft werden, ob die Beschreibungssprache eine Zeitmodellierung besitzt.

- **generische Eigenschaft**: Hier wird festgestellt, ob mit der Sprache Hardwarestrukturen generisch beschrieben werden können.

Nachdem wir nun die entsprechenden Kriterien vorliegen haben, mit denen die Beschreibungssprachen bewertet werden sollen, können wir an die einzelnen ausgewählten Beschreibungssprachen übergehen.

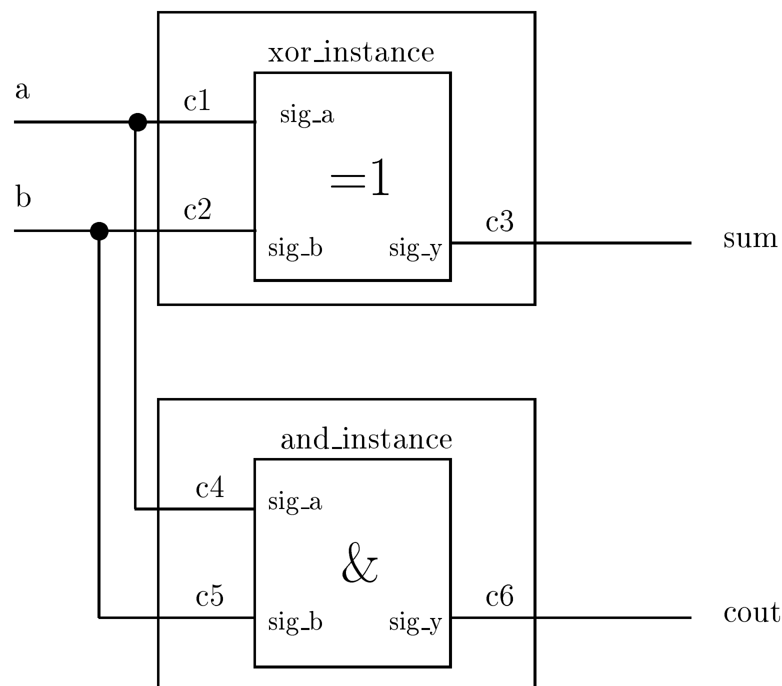

Abbildung 2.1: Struktur eines Halbaddierers

Dazu wird als Beispiel ein Carry-Ripple-Addierer in den folgenden Abschnitten untersucht. Damit soll gezeigt werden, wie gut die ausgewählten Sprachen hier angewendet werden können. Für die Programmierung in den einzelnen

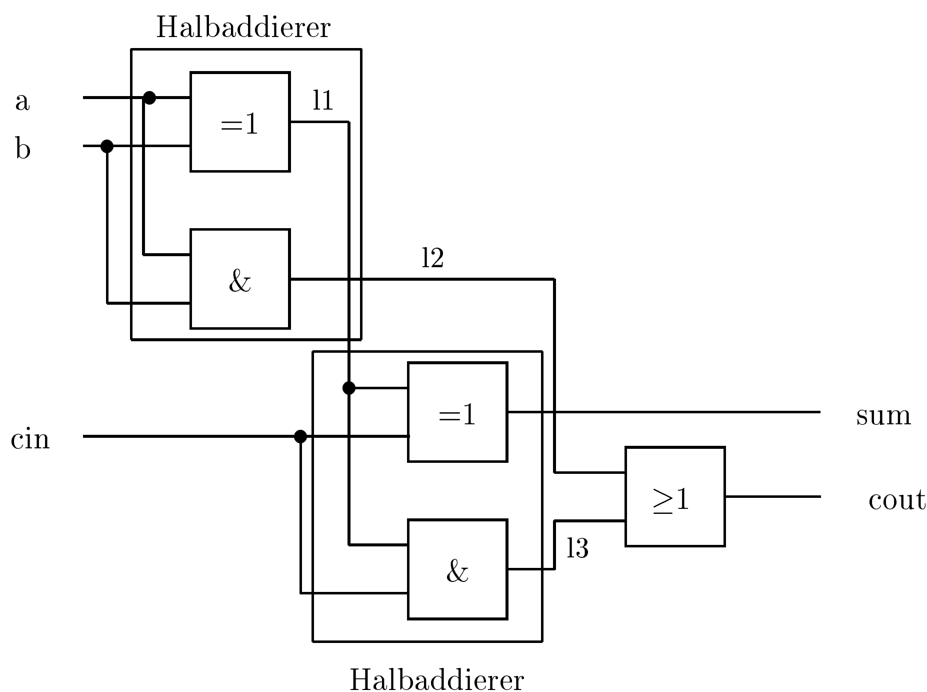

Abbildung 2.2: 1-Bit Volladdierer

Beschreibungssprachen wird der 1-Bit Halbaddierer (Abbildung 2.1), 1-Bit Volladdierer (Abbildung 2.2) und der Carry-Ripple-Adder (Abbildung 2.3) herangezogen.

2.1 LUSTRE

LUSTRE [13], [24] ist eine synchrone Programmiersprache, welche auf dem Datenflußprinzip besteht. In LUSTRE findet man Arrays, Rekursion und parametrisierte Unterprogramme vor. Ein LUSTRE-Programm beschreibt ein Netz, das durch eine globale Uhr gesteuert wird und dessen Knoten die einzelnen Funktionen sind. Bei der Ausführung erhält das Netz bei jedem Takt eine Menge von Eingaben und berechnet ein Menge von Ausgaben.

LUSTRE basiert auf Synchronität, die auch bei anderen Sprachen wie ESTEREL oder SIGNAL präsent ist. Das Netz reagiert permanent, so daß zur selben Zeit die Ausgaben produziert werden in der es die Eingaben erhält.

Das LUSTRE-Programm für einen Halbaddierer bekommt die Eingaben `a` und `b`, die in einen Und-Gatter und einen Xor-Gatter eingegeben werden. Die Ausgabe des LUSTRE-Programms ist die Summe `sum` und der Übertrag `cout`.

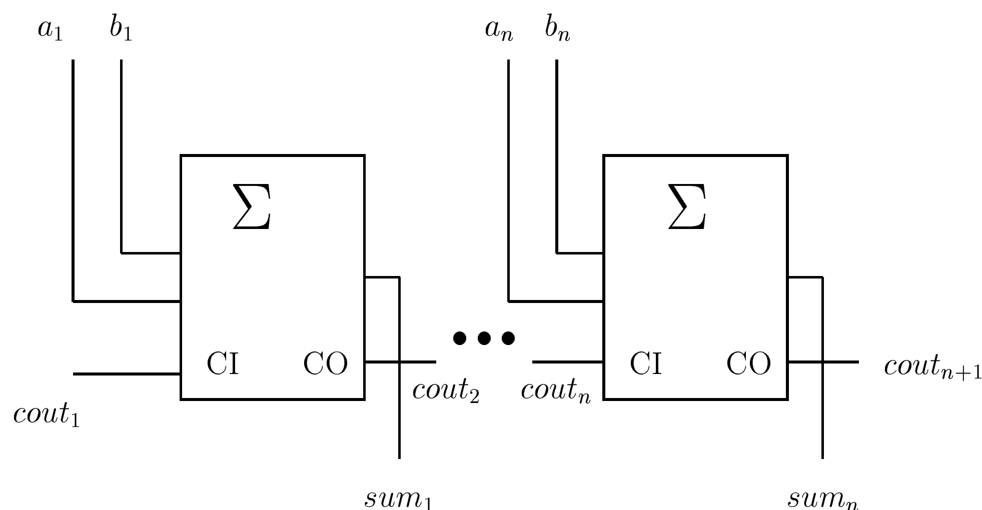

Abbildung 2.3: Schaltbild eines Carry-Ripple-Addierers

```
node HalfAdd(a, b bool) returns (sum, cout: bol);
let
    sum  = (a xor b);
    cout = (a and b);
tel;
```

Das folgende LUSTRE-Programm stellt einen 1-Bit Volladdierer vor. Die Schaltung wird in Abbildung 2.2 dargestellt.

```
node FullAdd(a, b, cin: bool) returns (sum, cout: bol);
let
     (l1,l2)  = HalfAdd(a,b);
     (l3,sum) = HalfAdd(cin,l2);
     cout     = (l1 or l3);
tel;
```

Eine generische Hardwarebeschreibung mittels LUSTRE soll anhand eines n-Bit breiten Carry-Ripple-Addierers vorgestellt werden.

```
node Add(const n: int; a, b: bool^n; cin: bool)
returns (sum: bool^n; cout: bool);
var C: bool^(n+1);
let
     C[0] = cin;
     (S, C[1..n]) = Add1(a, b, C[0..n-1]);
     cout = C[n];
```

```
tel;
```

Die Instantiierung für einen 32-Bit-Addierer sieht wie folgt aus.

```
node AddN32(a, b: bool^32; cin: bool) returns (sum: bool^32; cout: bool);
let
      (sum, cout) = Add(32, a, b, cin);
tel;
```

Durch die Instantiierung wird die generische Beschreibung der Hardwarestruktur durch konkrete Werte ersetzt. Hier wurde der generische Wert `n` durch 32 ersetzt.

Wie aus den Programmen zu entnehmen ist, benötigt LUSTRE relativ wenig Programmcode, um die Hardwarestrukturen zu beschreiben. Die gute Verständlichkeit der Programme macht es dem Programmierer bzw. Leser leicht, die Hardwarebeschreibungen zu verstehen und zu programmieren. Der Hardwarebezug ist ohne größere Mühe erkennbar. Da LUSTRE eine synchrone Programmiersprache ist und keine weitere Zeitmodellierung besitzt, ist die Zeitmodellierung nur synchron. Das heißt auch, daß mit LUSTRE keine asynchrone Schaltungen beschrieben werden können. Die synchrone Zeitmodellierung reicht aber aus, um einen Carry-Ripple Addierer zu beschreiben. Weiterhin ist die generische Hardwarebeschreibung mit LUSTRE möglich.

Mit LUSTRE ist es nicht möglich, eine Hardwareverifikation vorzunehmen, da keine Spezifikationen in LUSTRE vorgesehen sind. Es können lediglich die Hardwarestrukturen implementiert werden. Dennoch haben wir mit LUSTRE eine generische und synchrone Beschreibungssprache, die geeignet ist, generische Hardwarestrukturen hinreichend zu beschreiben.

2.2 SML

SML (Standard Meta Language) ist eine funktionale Programmiersprache [23]. Ein SML-Programm besteht aus einer Menge von Anweisungen. Die Anweisungen selbst beinhalten deklarative Variablen und Funktionen, die nach Bedarf aufgerufen werden können. Es gibt arithmetische Ausdrücke und komplexere Ausdrücke, die aus Anweisungen aufgebaut sind.

Als ein einfaches Beispiel soll wieder der Addierer betrachtet werden. Zuerst soll ein Halbaddierer (Abbildung 2.1) beschrieben werden.

Bevor der Halbaddierer programmmiert wird, sollen noch die Funktionen `or`, `xor` und `and` mit SML bereitgestellt werden.

```
fun or a b = a orelse b;

fun xor a b = not(a=b);

fun and a b = a andalso b;
```

Das Programm zum Halbaddierer ist ein Einzeiler.

```
fun half_add a b = (and a b, xor a b);
```

Einen 1-Bit Volladdierer (Abbildung 2.2) erhält man, durch das Hinzunehmen eines weiteren boolschen Eingangs `cin`. Die Berechnung von `sum` und `cout` sieht dann folgendermaßen aus:

```
fun full_add a b cin =
          let (l1,l2) = half_add a b
               (l3,sum) = half_add cin l2
               cout = or l1 l3
          in
               (sum, cout);
          end;
```

Nun soll ein n-Bit Addierer mit SML programmiert werden. Die Eingaben sind nun nicht mehr einfache boolesche Werte sondern boolesche Listen, wobei das niedrigstwertige Bit links steht. Die Länge der booleschen Listen `l1` und `l2` sind implizit angegeben.

```
fun Adder [] [] cin = (cin, [])
      | Adder(a::l1)(b::l2) cin =
               let
                    (c, sum) = Adder l1 l2 cin
                    (cout, s) = full_add a b c
               in
                    (cout, s::sum)
               end;
```

Wie zu sehen ist, sind alle Programme recht kurz. Der Hardwarebezug ist wenig erkennbar. Eine Zeitmodellierung gibt es in SML nicht. Die Hardwarebeschreibungen wurden in erster Linie nach der Art und Weise entworfen, wie die Ergebnisse ermittelt werden. Dabei spielt es keine Rolle, in welchen Hardwarekomponenten die Ergebnisse entstanden sind. Eine echte generische Hardwarebeschreibung liegt auch nicht vor, da hier die Listen rekursiv abgear-

beitet werden, bis der Basisfall - die leere Liste - erreicht ist. Mit funktionalen Sprachen wie SML ist es bedingt möglich, generische Hardwarebeschreibungen vorzunehmen. Weiterhin ist es nicht möglich, eine Hardwareverifikation vorzunehmen. SML ist ohne entsprechende Modifikationen für die generische Hardwarebeschreibung nicht empfehlenswert.

Im folgenden Abschnitt werden wir sehen, daß eine andere Programmiersprache, die auf SML aufbaut, wohl für die Hardwareverifikation geeignet ist.

2.3 HOL

HOL (Higher Order Logic) ist ein Theorembeweiser für Logik höherer Ordnung [12]. HOL ist in SML programmiert und somit eine Version von SML, die um eine Sammlung von SML-Funktionen ergänzt worden ist. Die Benutzung von HOL erfolgt durch den Aufruf der entsprechenden SML-Funktionen.

Der Benutzer beweist am Rechner interaktiv Theoreme auf Allgemeingültigkeit. Hierzu hat er zwei Möglichkeiten. Erstens versucht er, seinen Beweis vorwärts zu machen, d.h. er sammelt Lemmata und Theoreme, um an sein Ziel heranzukommen. Zweitens versucht er, von seinem Ziel aus alles so weit zu vereinfachen, bis die ganzen Teilprobleme für sich Tautologien liefern.

Das Verhalten von Modulen in ganzen Systemen wird durch Prädikate höherer Ordnung mit Funktionen als Argumente beschrieben. Prädikate können als charakteristische Funktionen angesehen werden, die Ein- und Ausgaben von Komponenten spezifizieren.

Die Realisierung eines Halbaddierers (Abbildung 2.1) kann in HOL wie folgt programmiert werden.

```
|- !a b sum cout. HALF_ADD a b sum cout =
     XOR a b sum ∧ AND a b cout
```

Ein HOL-Programm für einen 1-Bit Volladdierer kann folgendermaßen programmiert sein:

```
|- !a b cin sum cout.
     FULL_ADD a b cin sum cout =
    (?l1 l2 l3.
     HALF_ADD a b l1 l2 ∧ HALF_ADD l1 cin sum l3 ∧ OR l2 l3
```

Eine generische Beschreibung für den Carry-Ripple-Addierer mit HOL kann wie folgt aussehen:

```
|- (!a b cin sum cout.
     CARRY_RIPPLE 0 a b cin sum cout =
     FULL_ADD (HD a) (HD b) cin (HD sum) cout) ∧
    (!n a b cin sum cout.
     CARRY_RIPPLE (SUC n) a b cin sum cout =
      (?l1.
            FULL_ADD (HD a) (HD b) l1 (HD sum) cout ∧
            CARRY_RIPPLE n (TL a) (TL b) cin (TL sum) l1))
```

HOL zeigt ganz klar, daß nicht nur auf die Beschreibung der Hardwarestrukturen Wert gelegt wird, sondern auch auf die Verifikation. Somit ist HOL die erste von den ausgewählten Programmiersprachen, die die Hardwareverifikation unterstützt. Die Verifikation ist möglich, weil in der Hardwarebeschreibung nicht nur die Implementierung vorhanden ist, sondern auch die Spezifikation. Der Programmaufwand ist recht gering, obwohl die Spezifikation in der Sprache enthalten ist. Leider ist HOL schlecht verständlich, so daß ein zügiges Lesen und Schreiben nicht gewährleistet wird. Obwohl ein Hardwarebezug bei den Hardwarebeschreibungen erkennbar ist, ist kein klarer Zusammenhang zwischen Programmstruktur und Hardwarestruktur festzustellen. Da HOL die Vorzüge der Logik höherer Ordnung besitzt, ist eine Zeitmodellierung möglich. Auch der generische Aspekt wird durch HOL erfüllt.

Weil HOL wenig automatisiert ist und der Hardwarebezug nicht hinreichend ist, dürfte HOL hauptsächlich für die Verifikation von logischen Aussagen zu empfehlen sein.

2.4 VHDL

Die VHSIC Hardwarebeschreibungssprache ist in der Industrie ein Standard. VHDL steht für VHSIC Hardware Description Language, wobei VHSIC für Very High Speed IC steht [17].

Eine digitale Schaltung wird in VHDL durch hierarchisch zusammengesetzte Module beschrieben. Um Module in eine solche Beschreibung einfügen zu können, benötigt jedes Modul eine Entity-Deklaration, die die Schnittstelle zur Umwelt festlegt. Eine Entity-Relation sieht folgendermaßen aus:

```
entity name bf is
      PORT ( A: IN integer;
             B: In bit;
             Y: Out bit );
```

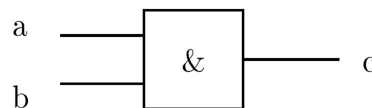

Abbildung 2.4: And-Gatter mit 2 Eingängen

```
end name;
```

Es wird damit ein Modul beschrieben mit dem Namen `name`, den beiden Eingangssignalen `A` und `B` und dem Ausgangssignal `Y`. Die Implementierung einer Schaltungfunktion eines Moduls, die sogenannte Architektur, wird folgendermaßen deklariert:

```
architecture identifier of entity_name is
      architecture_declarative_part
begin
      architecture_statement_part
end [architecture_simple_name];
```

Im Deklarationsteil werden Typen, Objekte, Funktionen, usw. deklariert. Im Anweisungsteil der Architekturdeklaration wird bei strukturellen Beschreibungen angegeben, aus welchen Untermodulen sich das Modul zusammensetzt und wie diese miteinander verbunden sind. Die einzelnen Module stellen Komponenten dar, die aufgrund ihrer Schnittstelle leicht mit anderen Komponenten verknüpft und hierarchisch gegliedert werden können.

Das erste Beispiel, das wir sehen werden, ist ein AND-Gatter wie in Abbildung 2.4. Die Modellierung des einfachen And-Gatters kann folgendermaßen aussehen:

```
ENTITY and2 IS
   PORT (a, b : IN BIT;
            c : OUT BIT);
END and2;

ARCHITECTURE and2_behav OF and2 IS
BEGIN
   c <= a AND b;
END and2_behav;
```

Es ist auch möglich, in VHDL zeitbezogene Aktionen zu modellieren. Wenn man etwa statt

```
c <= a AND b;
```

die Variante

```
c <= a AND b AFTER 5ns;
```

wählt, kann man erreichen, daß die Berechnung von a und b erst nach 5 Nanosekunden an c übergeben wird.

Es ist offensichtlich, daß die Programmierung mit VHDL mehr Programmcode benötigt als die bisherigen Sprachen. Dies wird am folgenden Beispiel noch deutlicher. Es soll ein 1-Bit-Halbaddierer mit VHDL entworfen werden. Als Modell soll hier Abbildung 2.1 dienen.

```
ENTITY halfadder IS
   PORT (a, b : IN BIT; sum, cout : OUT BIT);
END halfadder;

ARCHITECTURE structural OF halfadder IS
- - Komponentendeklarationen
   COMPONENT xor2
     PORT (c1, c2 : IN BIT; c3 : OUT BIT);
   END COMPONENT ;

   COMPONENT and2
     PORT (c4, c5 : IN BIT; c6 : OUT BIT);
   END COMPONENT ;
BEGIN
- - Komponenteninstantiierungen
   xor_instance : xor2 PORT MAP (a, b, sum) ;
   and_instance : and2 PORT MAP (a, b, cout) ;
END structural;

CONFIGURATION ha_config OF halfadder IS
- - Blockkonfiguration
   FOR structural     - - Komponentenkonfiguration
     FOR xor_instance: xor2
       USE ENTITY work.exor (behavioral)
       PORT MAP (c1, c2, c3) ;
     END FOR ;
   - - Komponentenkonfiguration
     FOR and2_instance: and2
       USE CONFIGURATION work.and2_config
       PORT MAP (a => c4, b =>c5, y = c6) ;
```

```
      END FOR ;
   END FOR ;
END ha_config ;
```

Wie zu sehen ist, ist nicht nur der Programmieraufwand größer, sondern auch die Anzahl der Befehle, die zu beherrschen sind.

Die Programmierung eines 1-Bit Volladdierers mit VHDL kann folgendermaßen geschrieben werden:

```
ENTITY add_1b IS
      PORT(a : IN std_logic; -- First summand
        b : IN std_logic; -- Second summand
        cin : IN std_logic; -- Carry IN
        sum : OUT std_logic; -- Sum OUT
        cout : OUT std_logic); -- Carry OUT
END add_1b;

ARCHITECTURE behavior OF add_1b IS
      BEGIN
        sum <= a xor b xor cin;
        cout <= (a and b) or (cin and (a or b));
      END behavior;

CONFIGURATION cfg_add_1b_behavior OF add_1b IS
      FOR behavior
      END FOR;
END cfg_add_1b_behavior;
```

Die generische Darstellung des n-Bit-Addierers folgt nun:

```
ENTITY add_nb IS
      GENERIC(N: POSITIVE := 4);
      PORT(a: IN std_logic_vector(N-1 DOWNTO 0);
           b: IN std_logic_vector(N-1 DOWNTO 0);
           cin: std_logic;
           sum: OUT std_logic_vector(N-1 DOWNTO 0);
           cout: OUT std_logic);
END add_nb;

ARCHITECTURE structure OF add_nb IS
COMPONENT add_1b
      PORT(a : IN std_logic; -- First summand
```

```
          b : IN std_logic; -- Second summand
          cin : IN std_logic; -- Carry IN
          sum : OUT std_logic; -- Sum OUT
          cout : OUT std_logic); -- Carry OUT
END COMPONENT;

     SIGNAL term: std_logic_vector(N DOWNTO 0);

BEGIN
     term(0) <= cin;
     loop_1: FOR i IN 0 TO N-1 GENERATE
          add_i: add_1b
          PORT MAP(a(i), b(i), term(i), sum(i), term(i+1));
     END GENERATE;
     cout <= term(N);
END structure;

CONFIGURATION cfg_add_nb_structure OF add_nb IS
     FOR structure
       FOR loop_1
          FOR add_i: add_1b
               USE CONFIGURATION WORK.cfg_add_1b_behavior;
          END FOR;
       END FOR;
     END FOR;
END cfg_add_nb_structure;
```

VHDL unterscheidet sich von den bisherigen Sprachen durch den größeren Programmumfang. Weil die Beschreibungsmöglichkeit von VHDL detaillierter ist, ist die Verständlichkeit der Hardwarebeschreibungen eher befriedigend. Der Hardwarebezug ist recht gut. Die Zeitmodellierung ist in VHDL auch enthalten. Die generische Beschreibung von Hardwarestrukturen ist auch möglich.

Die Verifikation wird teilweise unterstützt. In VHDL gibt es `Assertions`, mit denen Zusicherungen gemacht werden können. Spezifikationen, wie sie in HOL enthalten sind, sind in VHDL bedingt vorhanden.

2.5 Vergleich der Beschreibungssprachen

Die vorgestellten Sprachen waren alle in der Lage, einen Carry-Ripple Addierer zu beschreiben. SML hat dabei mehr eine Sicht auf das Hardwareverhalten, weil hier die Beziehung zum Hardwareaufbau nicht direkt ersichtlich ist. Die Hardwarebeschreibung liegt mehr auf der Ebene, wie das Ergebnis berechnet

	LUSTRE	SML	HOL	VHDL
Programmumfang	gering	gering	gering	groß
Verständlichkeit	gut	befriedigend	schlecht	befriedigend
Hardwarebezug	gut	wenig	wenig	sehr gut
Zeitmodellierung	synchron	nicht direkt	vorhanden	vorhanden
generisch	ja	ja	ja	ja

Abbildung 2.5: Vergleich der Beschreibungssprachen

wird, ohne explizit die Komponenten sichtbar zu machen.

HOL beschreibt die Hardwarestruktur und die Hardwarespezifikation direkt. Als Theorembeweiser ermöglicht HOL zudem auf der Hardwarestruktur Verifikation zu betreiben. Die Generizität in HOL basiert auf konsistenten Beschreibungen, d.h. die generische Beschreibung wird über Induktionsschemata gemacht.

LUSTRE hingegen zeigte Ansätze, wie die Komponenten, wo die Berechnungen gemacht werden, in Beziehung zueinander stehen. Ein echtes Herauslesen der Hardwarestruktur war auch hier nicht möglich.

Aus den VHDL-Programmen konnte abgeleitet werden, wie die Komponenten zueinander stehen und wie sie ihre Daten verarbeiten. Die prägnante Hardwarebeschreibung, die bei den anderen vorgestellten Sprachen positiv auffielen, ließ jedoch bei VHDL zumindest auf der RT-Ebene zu wünschen übrig.

Die nun im folgenden vorzustellende generische Hardwarebeschreibungssprache (GHDL) stellt den Bezug zur Hardware her, ist zusätzlich leicht verständlich und bedarf recht wenig Programmieraufwand. Der Befehlsvorrat ist überschaubar klein und dennoch enthält GHDL die Vorzüge, die die vorgestellten Programmiersprachen haben, ohne dabei ihre Nachteile zu übernehmen.

Kapitel 3

GHDL

GHDL (Generic Hardware Description Language) ermöglicht durch die synchrone Arbeitsweise, das zeitliche Verhalten in Programmen in Betracht zu ziehen. Durch die synchrone Interpretation kann ein effizientes und sequentielles Programm compiliert werden. Weil Spezifikationen bei GHDL in Anlehnung an die temporale Logik implementiert wurden, können diese somit auch modelliert werden. Das Nachbilden einer Hardwarestruktur ist relativ einfach. Weiterhin ist es erlaubt, rekursiv zu programmieren, um eine prägnante und klare Hardwarebeschreibung zu gewährleisten.

Es ist möglich, die Hardwarebeschreibungen (generische und nicht generische) vom Benutzer direkt oder alternativ dazu auch aus einer Datenbank, die die Hardwarebeschreibungen gespeichert hält, zu bekommen. Die generischen Beschreibungen werden dann „abgerollt" (instantiiert), d.h. die induktiven Parameter werden durch konkrete Werte ersetzt, die durch den Benutzer oder durch die in der Datenbank enthaltenen Beschreibungen gegeben sind. Parameter sind oft Bitbreiten oder eine Maßzahl dafür, die angibt wie die Rekursion gesteuert werden soll.

Wenn man in Abbildung 3.1 die linke Seite heranzieht, hat man den Vorteil, daß die Verifikation automatisch gemacht wird. Dafür nimmt man aber den Nachteil in Kauf, nur eine einzige Instanz beweisen zu können. Auf der rechten Seite in Abbildung 3.1 hat man dagegen den Vorteil, alle Instanzen verifizieren zu können, aber den Nachteil, daß dies nicht automatisch möglich ist. Oft will man nur eine spezielle Lösung haben, so daß über die linke Seite gegangen wird. Dann ist der Nachteil, daß man nur eine einzige Instanz verifizieren kann, kein Nachteil mehr, da ohnehin nur eine einzige Lösung gewollt war. Meistens wird man wohl eher so vorgehen, daß man einige Spezialfälle verifiziert und dann erst die Verifikation über die rechte Seite macht und toleriert, daß die Verifikation wesentlich länger dauert.

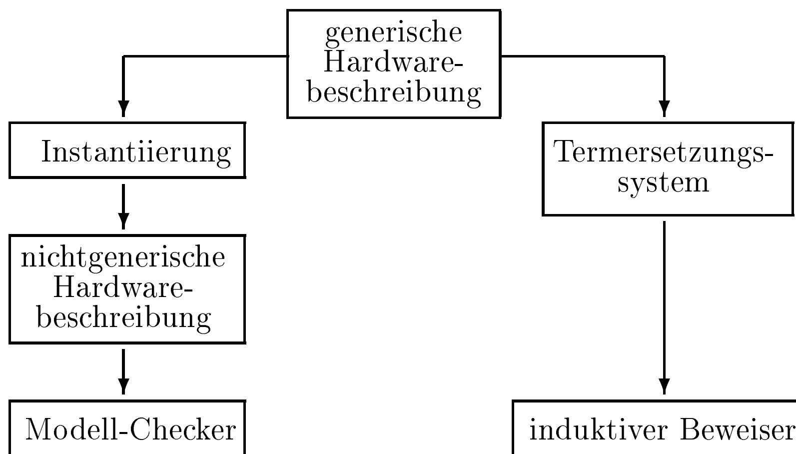

Abbildung 3.1: Arbeitsweise von GHDL

3.1 Datentypen

In diesem Abschnitt werden die Datentypen vorgestellt, die in GHDL benutzt werden. Die Typen, die in GHDL erlaubt sind, reichen von einfachen skalaren Zahlen bis hin zu abstrakten Datentypen.

Datentypen von GHDL sind:

1. *numerische Werte* : Sie repräsentieren zum einen die Bitwerte (0 oder 1), Listenlängen bei den generischen Hardwarebeschreibungen oder dienen als Abbruchkriterium für die Rekursion bzw. Rekursionsauflösung.

2. *Listen* : Mit Listen werden Eingänge mit variabler Bitbreite spezifiziert. Dabei können Listen aus booleschen Werten oder Listen bestehen.

3.2 Spezifikationssprache

Boolesche Funktionen eignen sich zur Modellierung und Verifikation von Schaltnetzen. Unter einem Schaltnetz wird hier die Verschaltung von Gattern verstanden. Um asynchrones Schaltverhalten auszuschließen, dürfen keine Rückkopplungen vorhanden sein: Auf jedem möglichen Pfad des Schaltnetzes wird jedes Gatter genau einmal durchlaufen [9].

Das Verhalten eines Schaltnetzes kann direkt durch eine boolesche Funktion repräsentiert werden. Die Schaltnetzeingänge entsprechen den Variablen der Funktion, die Ausgangswerte dem Funktionswert. Boolesche Funktionen lassen jedoch nur statische Betrachtungen von Schaltnetzen zu: Beschaltungen der Gattereingängen mit Eingangssignalwerten entsprechen Belegungen der

Variablen der booleschen Funktionen. Aus dieser Belegung resultiert der Ausgangswert des Schaltnetzes. Somit können hier Gatter nur verzögerungsfrei modelliert werden.

Ein typisches Beweisziel bei der Verifikation ist

$$\models I \rightarrow S,$$

bei dem unter der Annahme einer gegebenen Implementierungsbeschreibung I gezeigt werden soll, daß eine Spezifikation S gilt. Der Beweis erfolgt durch den Einsatz von Theorembeweisern oder Modell-Checkern.

3.2.1 Lineare temporale Logik

Lineare temporale Logiken finden eine große Anwendung bei der Verifikation digitaler Schaltungen. Sie erlauben zwar nicht die Verwendung von abstrakten Datentypen, lassen dafür aber eine komfortable zeitliche Abstraktion zu. Bei der gewöhnlichen Aussagenlogik genügt eine Variablenbelegung, um den Wahrheitswert einer Formel zu bestimmen. Bei der linearen temporalen Logik (LTL) geht man davon aus, daß die Variablenbelegung von der Zeit abhängt. Die Zeit wird dabei als Abfolge diskret aufeinander folgender Zeitpunkte aufgefaßt und läßt sich somit durch die natürlichen Zahlen modellieren.

Die Syntax von LTL wird neben Variablen und aussagenlogischen Verknüpfungen auch durch Verknüpfungen mit temporalen Operatoren gebildet:

- **X**: Next
- **G**: Always
- **F**: Eventually
- **U**: Until
- **W**: When

Die folgende Definition beschreibt die Menge der LTL-Formeln mit einer gewissen Operatormenge:

Definition (LTL-Formeln) *Die Menge der LTL-Formeln über eine Menge von Variablen V ist die kleinste Menge, welche die folgenden Punkte erfüllt:*

- *Jede Variable $x \in V$ ist eine LTL-Formel.*
- *Sind t_1 und t_2 Terme, dann ist auch $t_1 = t_2$ eine LTL-Formel.*
- *Sind φ und ψ LTL-Formeln, so sind $\neg\varphi$, $\varphi \wedge \psi$ und $\varphi \vee \psi$ LTL-Formeln.*

- *Sind φ und ψ LTL-Formeln, so sind* $\mathbf{X}\ \varphi$, $\mathbf{G}\ \varphi$, $\mathbf{F}\ \varphi$, $[\varphi\ \mathbf{W}\ \psi]$, $[\varphi\ \mathbf{U}\ \psi]$ *LTL-Formeln.*

Zusätzlich ist es möglich mit Termoperatoren auf Termen Auswertungen vorzunehmen. Terme sind Konstanten und Variablen. Falls t ein Term ist, dann ist X t auch ein Term. Wenn *op* ein Operator ist und t ein Term, dann ist auch *op* t ein Term. Dabei haben die einzelnen Termoperatoren die folgenden Bedeutungen:

$HD(l)$: Bezeichnet den Kopf einer nichtleeren Liste, d.h.
$HD([x_0, ..., x_n]) = x_0$

$TL(l)$: Bezeichnet den Rest einer nichtleeren Liste, d.h.
$TL([x_0, ..., x_n]) = [x_1, ..., x_n]$

$CONS(e, l)$: Fügt am vorderen Ende der Liste l das Element e an, d.h.
$CONS(e, [x_0, ..., x_n]) = [e, x_0, ..., x_n]$

$APPEND(l_1, l_2)$: Bezeichnet die Konkatenation zweier Listen l_1 und l_2, d.h.
$APPEND([x_0, ..., x_n], [y_0, ..., y_m]) = [x_0, ..., x_n, y_0, ..., y_m]$

$FIRSTN(n, l)$: Bezeichnet das n-elementige Präfix einer Liste l, d.h.
$FIRSTN(n, [x_0, ..., x_{n+m}]) = [x_0, ..., x_{n-1}]$

$LASTN(n, l)$: Bezeichnet die letzten n Elemente einer Liste l, d.h.
$LASTN(n, [x_0, ..., x_{n+m}]) = [x_{m+1}, ..., x_{n+m}]$

$BUTFIRSTN(n, l)$: Bezeichnet die Restliste nach Entfernung der ersten n Elementen, d.h.
$BUTFIRSTN(n, [x_0, ..., x_{n+m}]) = [x_n, ..., x_{n+m}]$

$BUTLASTN(n, l)$: Bezeichnet die Restliste nach Entfernung der letzten n Elementen, d.h.
$BUTLASTN(n, [x_0, ..., x_{n+m}]) = [x_0, ..., x_m]$

Die Verifikation kann nun mit temporaler Logik in zwei Schritten durchgeführt:

1. Erstellen von Spezifikation S und Implementierung I in temporaler Logik.

2. Beweis, daß $\mathbf{G}\ (I \rightarrow S)$ (partielle Spezifikation) bzw. $\mathbf{G}\ (I \leftrightarrow S)$ eine Tautologie ist.

Abbildung 3.2 soll nun mit der linearen temporalen Logik spezifiziert werden. Die Spezifikation kann folgendermaßen gelesen werden: Es soll immer gelten,

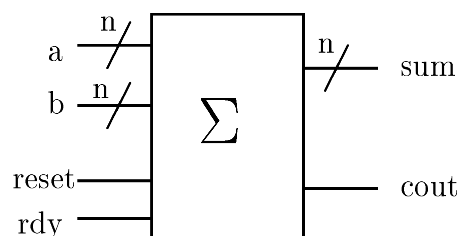

Abbildung 3.2: Addierer mit Reset und Ready

wenn *reset* nicht gilt und *rdy* gilt soll daraus folgen, daß die Liste bestehend aus *cout* und *sum* der Summe aus $a + b$ entspricht, wenn *rdy* erneut gilt.

$$\mathbf{G}(\ \neg reset\ \wedge\ rdy \rightarrow ([cout \triangleright sum] = a + b)\ \mathbf{W}\ rdy)$$

Nachdem wir über die Datentypen in GHDL zu der Spezifikationsprache gekommen sind, werden wird jetzt die einzelnen Implementierungsbeschreibungen der Hardwaretypen näher kennenlernen.

3.3 Implementierungsbeschreibung

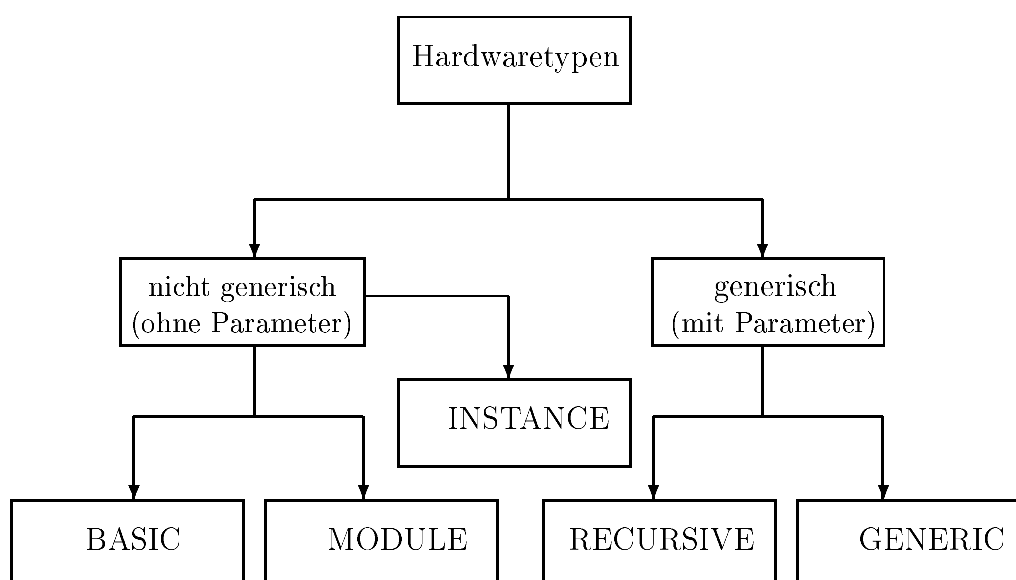

Abbildung 3.3: Hardwaretypen im Überblick

Hier wird auf alle Hardwaretypen, die in GHDL vorkommen, ausführlicher eingegangen. Im einzelnen handelt es sich um die in Abbildung 3.3 vorgestelleten Komponenten. Den Anfang macht der nicht generische Hardwaretyp *BASIC*.

3.3.1 BASIC

In der ersten Zeile steht neben der Art des Hardwaretyps (hier `BASIC`) der Name der zu realisierenden Funktion. Zwischen den Klammern folgen dann die Eingabevariablen, die durch Kommas getrennt sind. `INIT` und `NEXT` leiten den Teil für die die mögliche Initialisierung der Anfangs- und Folgezustände. Die einzelnen `TRANSITION`-Anweisungen sind mit Semikola zu trennen. Mit dem Hardwaretyp *BASIC* kann man somit einen endlichen Automaten modellieren. Durch Anwenden des `DEFINE`-Teils können benutzerspezifische Definitionen vorgenommen werden. Die Ausgabe der errechneten Werte erfolgt über den `OUTPUT`-Teil. Die allgemeine Struktur eines GHDL-Programms vom Hardwaretyp `BASIC` ist in GHDL-Grammatik folgendermaßen definiert:

```
BASIC Funktionsname (Liste von Variablen)
INIT oder NEXT
        Transition_0;
              ⋮
        Transition_k;
DEFINE
        Definition_0;
              ⋮
        Definition_l;
OUTPUT
        Ausgabe_0;
              ⋮
        Ausgabe_m;
END;
```

XOR soll hier für den Fall mit zwei Eingängen und einem Ausgang programmiert werden.

```
BASIC Xor(inp1,inp2)
    OUTPUT
        out := ~(inp1 = inp2);
END;
```

Weitere Beispiele für Gatter in GHDL sind:

```
BASIC And(inp1,inp2)
    OUTPUT
        out := inp1 & inp2;
END;
```

```
BASIC Or(inp1,inp2)
   OUTPUT
      out := inp1 | inp2;
END;

BASIC Not(inp)
   OUTPUT
      out := ~inp;
END;
```

Darüber hinaus können mit dem Hardwaretyp *BASIC* auch Anfangs- und Folgezustände modelliert werden. Dadurch ist es möglich, die Funktionsweise eines D-Flipflops zu programmieren.

Mit `INIT` wird der Anfangszustand festgelegt und mit `NEXT` der Folgezustand. Das Zeichen ∼ bedeutet, daß die folgende Angabe negiert ist.

```
BASIC Dflipflop(inp)
      INIT(q) := 0;
      NEXT(q) := inp;
   OUTPUT
      out1 := q;
      out2 := ~q;
END;
```

3.3.2 MODULE

Im Hardwaretyp *MODULE* können nun die Gatter, die mit dem Hardwaretyp *BASIC* erstellt wurden, zu Kompositionen zusammengefaßt werden. Als ein Beispiel soll ein 1-Bit-Halbaddierer dienen, der als Gatter XOR und AND verwendet. Abbildung 3.4 dient als Vorlage für die Programmierung.

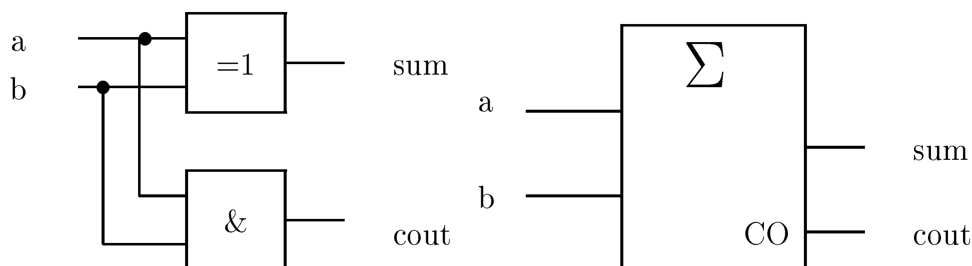

Abbildung 3.4: Schaltbild und -symbol eines 1-Bit-Halbaddierers

Bevor mit GHDL programmiert wird, soll auch hier zuerst die allgemeine Programmbeschreibung des Hardwaretyps gezeigt werden.

```
MODULE Funktionsname (Liste von Variablen)
        Komponente_0;
              ⋮
        Komponente_k;
DEFINE
        Definition_0;
              ⋮
        Definition_l;
OUTPUT
        Ausgabe_0;
              ⋮
        Ausgabe_m;
SPEC
        Spezifikation_0;
              ⋮
        Spezifikation_n;
END;
```

Wie beim Hardwaretyp *BASIC* besteht auch die Möglichkeit über eine Schnittstelle die Variablenliste zu übergeben. Der Unterschied zwischen dem Hardwaretyp *BASIC* und *MODULE* besteht darin, das im Hardwaretyp *MODULE* zwei neue Aspekte hinzugekommen sind. Das ist zum einen der Komponentenblock und zum anderen der Spezifikationsblock. Der Komponentenblock beschreibt, wie die einzelnen Hardwarestrukturen in der Schaltung miteinander verknüpft sind, d.h. aus dem Komponentenblock kann man die Hardwarestruktur herauslesen und umgekehrt ist es einfach, aus der Hardwarestruktur auf die Komponetenbeschreibung zu kommen. Im Spezifikationsteil dagegen wird dem Anwender die Möglichkeit gegeben, die entworfene Schaltung zu spezifizieren. Anhand der Spezifikation ist es dann möglich zu überprüfen bzw. zu beweisen, ob die Implementierung der Spezifikation genügt.

Es können aber auch Hardwaretypen der Art *MODULE* selbst wieder in Komposition zueinander stehen. Dazu betrachten wir einen 1-Bit-Volladdierer, der zu einer Eingabe von zwei Bitwerten `a`, `b` und einem Übertrag `cin` die Summe `sum` mit Übertrag `cout` berechnet (Abbildung 3.5).

```
MODULE HalfAdder(a,b)
      C1 : Xor(a,b);
      C2 : And(a,b);
```

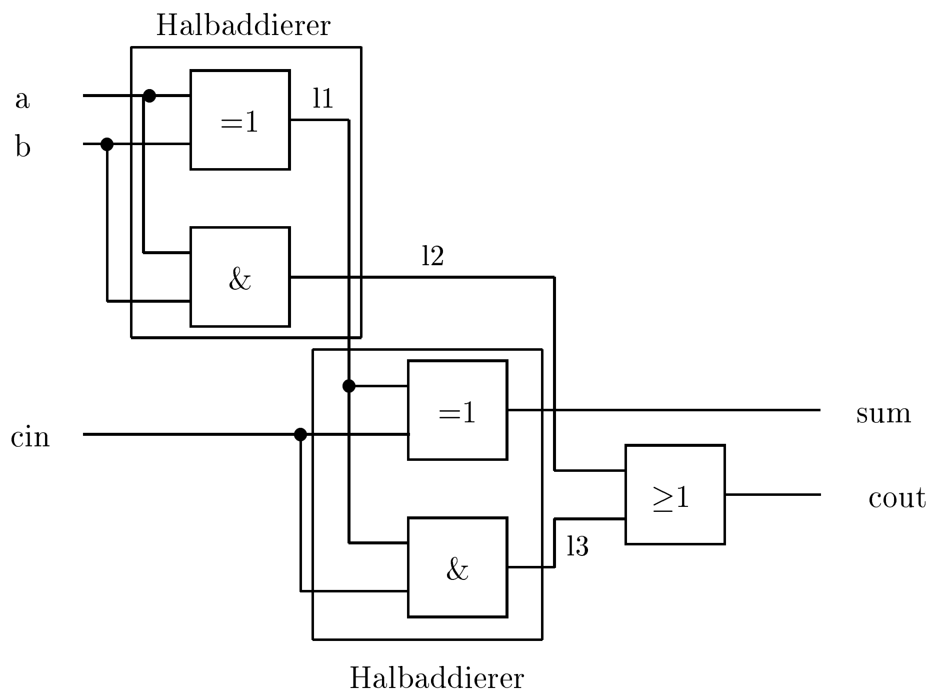

Abbildung 3.5: Schaltbild eines 1-Bit-Volladdierers

```
   OUTPUT
      sum := C1.o;
      cout := C2.o;
   SPEC
      G([cout,sum] = a + b);
END;

MODULE FullAdder(a,b,cin)
      C1 : HalfAdder(a,b);
      C2 : HalfAdder(cin,C1.sum);
      C3 : Or(C1.cout,C2.cout);
   OUTPUT
      sum := C2.sum;
      cout := C3.o;
   SPEC
      G([cout,sum] = a + b + cin);
END;
```

Das Beispiel mit dem Volladdierer kann wiederum selbst zu höherwertigen Volladdierern vergrößert werden. Wenn man das Schaltsymbol eines 1-Bit-Volladdieres (Abbildung 3.6) zur Darstellung verwendet, sieht ein Carry-Ripple-

Addierer wie in Abbildung 3.7 aus.

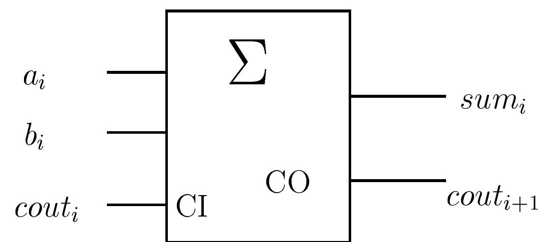

Abbildung 3.6: Schaltsymbol eines 1-Bit-Volladdierers

Dabei werden zwei Dualzahlen mit mehreren Stellen so addiert, daß für jede Stelle ein Volladdierer plaziert wird. Der Übertrag der Stelle i wird im Volladdierer der Stelle $i+1$ berücksichtigt.

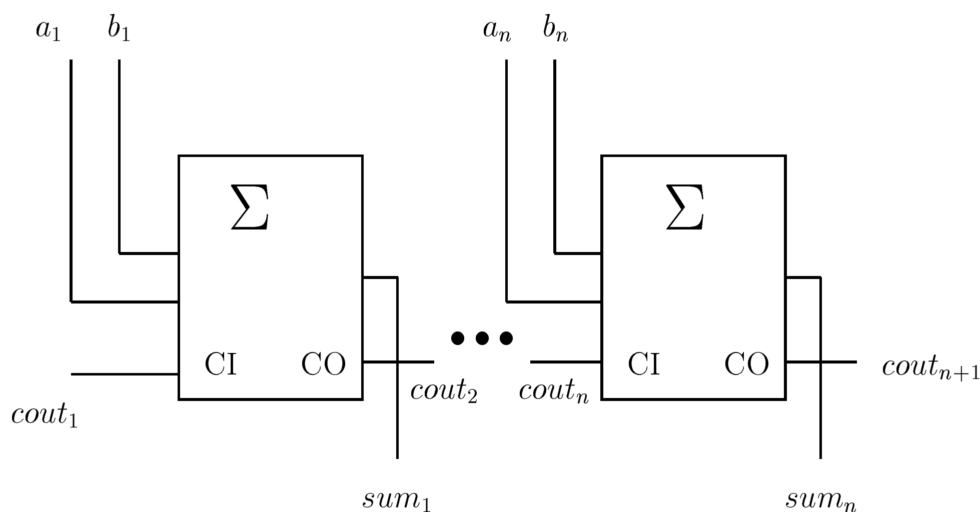

Abbildung 3.7: Schaltbild eines Carry-Ripple-Addierers

Die Programmierung des Carry-Ripple-Addierers mit beliebig vielen Stellen wird dann gleich im folgenden Abschnitt erstellt und kommentiert.

3.3.3 RECURSIVE

Hier werden zusätzlich Datengleichungen (`DATAEQUATIONS`) bereitgestellt, welche in der Spezifikation eine Trennung von Zeit und Daten erlauben. Mit der Rekursionszahl wird die Rekursion gesteuert. Der Unterschied zu bisherigen Schnittstellenelementen, hier Listen von Termen, besteht darin, daß die Terme zusätzlich Längenangaben besitzen. Die Längenangaben legen die Anzahl

der Elemente in der jeweiligen Liste fest. Mit `Ausdruck` wird die Rekursion gesteuert.

Die Struktur eines GHDL-Programms mit dem Hardwaretyp *RECURSIVE* ist in GHDL definiert als:

```
RECURSIVE Funktionsname (Rekursionszahl; Liste von Termen)
   IF (Ausdruck) THEN
               Komponent_0;
                   ⋮
               Komponent_k;
       DEFINE
               Definition_0;
                   ⋮
               Definition_l;
       OUTPUT
               Ausgabe_0;
                   ⋮
               Ausgabe_m;
   ELSE
               Komponent_0;
                   ⋮
               Komponent_n;
       DEFINE
               Definition_0;
                   ⋮
               Definition_o;
       OUTPUT
               Ausgabe_0;
                   ⋮
               Ausgabe_p;
   END
   DATAEQUATIONS
           Datengleichung_0;
               ⋮
           Datengleichung_q;
   SPEC
           Spezifikation_0;
               ⋮
           Spezifikation_r;
END;
```

3.3.3.1 Beispiel: Carry-Ripple-Addierer

Hier folgt nun das GHDL-Programm für den Carry-Ripple-Addierer mit `n` Stellen.

```
RECURSIVE CarryRippleAdder_n(n;l1:n,l2:n,carry:1)
   IF (n>1) THEN
         C1: FullAdder(1;HD(l1),HD(l2),carry);
         C2: CarryRippleAdder_n(n-1;TL(l1),TL(l2),C1.o);
       OUTPUT
         sum := CONS(C1.sum,C2.sum);
   ELSE
         C1: FullAdder(1;HD(l1),HD(l2),carry);
       OUTPUT
         sum := [C1.sum];
         cout:= C1.cout;
   END
   DATAEQUATIONS
       f1 := ( [cout,sum] = addition(l1,l2,carry));
   SPEC
     G f1;
END;
```

Mit `l1:n` bzw. `l2:n` wird die Länge der jeweiligen Listen angegeben. Mit dem ersten `n` wird die Rekursion gesteuert. Die Rekursion selbst ist sehr einfach. Solange die Rekursionstiefe, hier die 1, nicht erreicht wird, wird mit dem Induktionsfall weitergerechnet. Bei `C1` werden die Köpfe der beiden Listen `l1` und `l2` mit dem Übertrag aus dem vorangegangenen Schritt in den 1-Bit-Volladdierer `FullAdder` eingespeist. Im darauf folgenden Schritt werden bei `C2` Reste der beiden Listen mit dem Übertrag aus der aktuellen Rekursionsstufe in den nächsten Rekursionschritt übergeben. Wenn am Schluß die Rekursionstiefe erreicht wird, wird der Basisfall `ELSE`-Teil angewandt, ein einfaches Einlesen des letzten 1-Bit-Volladdierers mit den letzten beiden Werten der Listen und dem Übertrag aus der vorhergehenden Rekursionsstufe.

`C1` bzw. `C2` stellen die Komponenten dar. Hier ist es ein einfacher 1-Bit-Volladdierer und ein rekursiv definierter Carry-Ripple-Addierer.

Im Ausgabeteil `OUTPUT` werden die Ergebnisse aus den beiden Komponenten `C1` und `C2` durch den Operator `CONS` miteinander verbunden, indem das Ergebnis aus `C1` dem Ergebnis von `C2` vorangestellt wird.

Die Gleichung `DATAEQUATIONS` vergleicht die Ausgabe des Moduls mit dem Ergebnis der Listenfunktion `addition`. Der Vergleich wird in `f1` gespeichert. Schließlich wird mit `G f1` im `SPEC`-Teil verlangt, daß `f1` ständig gelten soll.

Auf die Instantiierung des Carry-Ripple-Addierers möchte ich an dieser Stelle auf den Abschnitt **INSTANCE** in diesem Kapitel verweisen.

3.3.3.2 Omega-Netzwerk

Als eine weitere rekursive Hardwarebeschreibung soll aus der Literatur ein dynamisches Netz entnommen werden, das zwei Knoten miteinander verbindet [?]. Der Unterschied zwischen einer dynamischen und einer statischen Verbindung liegt darin, daß bei statischen Netzen eine feste Verbindung zwischen den Paaren existiert. Genauer soll hier ein Multiprozessorsystem mit einem Schaltwerk vorgestellt werden.

Die Schaltelemente eines Schalternetzwerks bestehen im Normalfall aus Zweierschaltern mit zwei Eingängen und zwei Ausgängen, die entweder durchschalten oder die Ein- und Ausgänge überkreuzen können, wie in Abbildung 3.8 dargestellt.

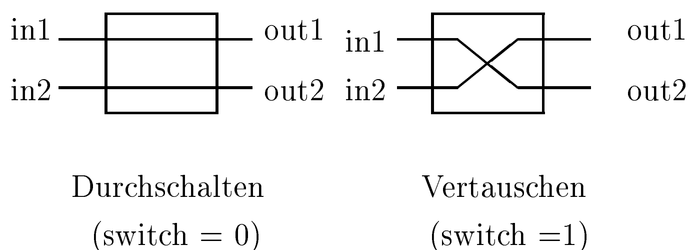

Abbildung 3.8: Schalterstellungen von Zweierschaltern

Für Verbindungsnetze auf der Basis von Zweierschaltern gibt es auch die Bezeichnung „Permutationsnetze“. Dieser Name weist auf die Eigenschaft hin, daß p Eingänge des Netzes gleichzeitig auf p Ausgänge geschaltet werden können und somit eine Permutation der Eingänge erzeugt wird.

Auf diese Permutationseigenschaft wird in der Programmierung mit GHDL in `PerfectShuffle_n` zurückgegriffen werden.

Die meisten Permutationsnetzwerke sind regulär aufgebaut: sie besitzen p Eingänge, p Ausgänge und k Stufen mit jeweils $p/2$ Zweierschaltern, wobei die Zahl p normalerweise eine Zweierpotenz ist.

Zwischen den Eingängen des Permutationsnetzwerks und den Eingängen der ersten Stufe, zwischen den Ausgängen der i-ten Stufe und den Eingängen der $i+1$-ten Stufe sowie den Ausgängen der letzten Stufe und den Ausgängen des Verbindungsnetzwerks existieren bijektive Zuordnungen.

Obwohl beliebige Permutationen denkbar sind, werden nur wenige unterschiedliche Grundmuster verwendet.

Eine mögliche Darstellung dieser Grundmuster erhält man folgendermaßen. Man stellt die Eingänge als binären Zahlenwert dar, d.h. man numeriert die Eingänge beginnend mit 0 bis zum (2^n-1)-ten Eingang durch. Auf diese Weise ordnet man jedem Eingang sozusagen eine Art Adresse zu. Die Permutation läßt sich durch eine Bitmanipulation dieser Adresse darstellen, so daß am Ausgang neue Bitmuster entstehen. Sortiert man die neu entstandenen Bitmuster nach ihrem binären Zahlenwert und verbindet man jeden Eingang mit dem aus der Bitmanipulation zugeordneten Ausgang, so ergibt sich ein charakteristisches Grundmuster.

Eine der häufig benutzten Grundmuster soll im folgenden dargestellt werden. Dabei wird davon ausgegangen, daß die Zuordnung der Adreßbits zu den Eingängen nach dem Schema (a_n, a_{n-1}, ..., a_2, a_1) erfolgt.

Die Mischungspermutation M ("Perfect Shuffle") läßt sich durch eine einfache Kreisverschiebung der Areßbits darstellen:

$$M(a_n, a_{n-1}, ..., a_2, a_1) = M(a_{n-1}, ..., a_2, a_1, a_n)$$

In Abbildung 3.9 ist das zugehörige Grundmuster für 8 Eingänge und 8 Ausgänge (n = 3) dargestellt.

Abbildung 3.9: Mischungspermutation für 8 Ein- und Ausgänge

Bei der Mischungspermutation wird der obere und untere Adreßbereich vermischt. Ein Beispiel soll dies verdeutlichen:

$$M([a, b, c, d],[1, 2, 3, 4]) = [a, 1, b, 2, c, 3, d, 4]$$

Eine der bekannten Netzformen ist das Omega-Netzwerk, das auf der Mischungspermutation beruht. Das Netzwerk für $p = s^n$ Ein- und Ausgänge umfaßt $n = ld(p)$ Stufen von Zweierschaltern, die untereinander jeweils nach

dem Grundmuster der Mischungspermutation verknüpft sind. Ein speichergekoppeltes Omega-Netzwerk verbindet Prozessoren mit Speichermodulen, wie in Abbildung 3.10 dargestellt.

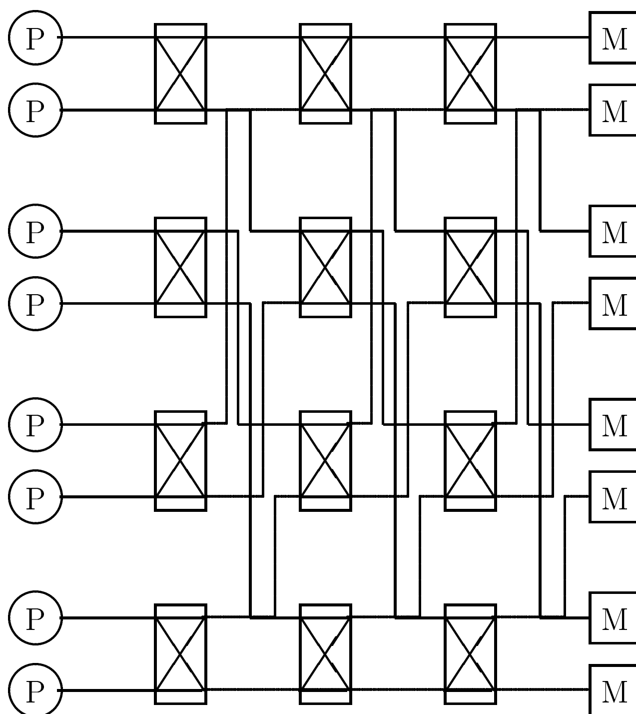

Abbildung 3.10: Speichergekoppeltes Omega-Netzwerk

Die Gesamtzahl der Zweierschalter in einem Omega-Netzwerk mit $p = 2^n$ Ein- und Ausgängen beträgt $(p/2) * ld(p)$.

Beispielsweise erlaubt ein Netz mit $2^{12} = 4096$ unterschiedliche Zuordnungen (Permutationen) von 8 Eingängen zu 8 Ausgängen gegenüber der Gesamtzahl von $8! = 40320$ insgesamt möglichen Zuordnungen.

Nachdem nun der theoretische Rückhalt in Sachen „Netzwerktheorie" hergestellt ist, wenden wir uns der Programmierung mit der generischen Hardwarebeschreibungssprache GHDL zu.

Das Vorgehen wird so aussehen, daß zuerst der Zweierschalter mit dem Hardwaretyp *MODULE* programmiert wird. Die hierzu nötigen Gatter, wie AND, OR oder NEG, kann man aus dem Abschnitt für den Hardwaretyp *BASIC* entnehmen. Im Anschluß daran werden wir die Realisierung der Mischungspermutation mit dem Programm `RECURSIVE PerfectShuffle_n` darstellen.

Abschließend wird dann mit dem Programm `RECURSIVE Combine_n1` die einzelnen Stufe des Omega-Netzwerkes verbunden.

Mit dem Programm `GENERIC Omega_Netzwerk_n` wird dann die Beschreibung für das Netzwerk komplettiert.

```
MODULE Zweierschalter(switch,in1,in2)
      C1: NEG(switch);
      C2: AND(switch,in1);
      C3: AND(switch,in2);
      C4: AND(C1.o,in1);
      C5: AND(C1.o,in2);
      C6: OR(C3.o,C4.o);
      C7: OR(C2.o,C5.o);
   OUTPUT
      out1 := C6.o;
      out2 := C7.o;
END;
```

Die Eingabe „switch“ entscheidet, ob der Zweierschalter über Kreuz schaltet oder einfach die Eingänge mit den Ausgängen direkt verbunden sind.

```
RECURSIVE PerfectShuffle_n(n;s:n,a:n,b:n)
   IF (n>1) THEN
         C1: Zweierschalter(1;HD(s),HD(a),HD(b));
         C2: PerfectShuffle_n(n-1;TL(s),TL(a),TL(b));
       OUTPUT
           o := CONS(C1.out1,CONS(C1.out2,C2.o));
   ELSE
         C1: Zweierschalter(1;HD(s),HD(a),HD(b));
       OUTPUT
           o := CONS(C1.out1,C1.out2);
   END
END;
```

Mit `s:n` werden die Zweierschalter bezeichnet, die sich in der jeweiligen Stufe befinden. Die weiteren Eingaben `a:n` und `b:n` stellen die obere und untere Hälfte des Adreßbereichs dar. Die Vertauschung wird entsprechend Abbildung 3.9 durchgeführt.

Um nun die einzelnen Stufen miteinander verbinden zu können, brauchen wir noch das folgende Programm, das $n1$ Stufen von Mischungspermutationen verbindet.

```
RECURSIVE Combine_n1(n1;s:(n1*n),a:n,b:n)
   IF (n1>1) THEN
         C1: PerfectShuffle_n(n;FIRSTN(n,s),a:n,b:n);
         C2: NetHelp_n1(n1-1;LASTN(n*n1-n,s),FIRSTN(n,C1.o),
                              LASTN(n,C2.o));
       OUTPUT
           o := APPEND(C1.o,C2.o);
   ELSE
         C1: PerfectShuffle_n(n;FIRSTN(n,s),a:n,b:n);
       OUTPUT
           o := [C1.o];
   END
END;
```

Das eigentliche Omega-Netzwerk wird durch das nachstehende Programm erst richtig realisiert. Die Beschreibung des Programms obliegt aber dem folgenden Unterabschnitt.

3.3.4 GENERIC

Um eine verständliche Erklärung geben zu können, soll auch hier zuerst die allgemeine Struktur für die Hardwarebeschreibung gegeben werden.

```
GENERIC Funktionsname (Parameterliste; Liste von Termen)
     Komponente_0;
         ⋮
     Komponente_k;
  DEFINE
     Definition_0;
         ⋮
     Definition_l;
  OUTPUT
     Ausgabe_0;
         ⋮
     Ausgabe_m;
  DATAEQUATIONS
     Datengleichung_0;
         ⋮
     Datengleichung_n;
  SPEC
     Spezifikation_0;
```

⋮
Spezifikation$_o$;
```
END;
```

Damit sieht die Realisierung des Omega-Netzwerks so aus:

```
GENERIC OmegaNetwork_n1(n1;s:n*n1,a:n,b:n)
      C1: Combine_n1(n1;s:n*n1,a:n,b:n);
   OUTPUT
        o := LASTN(2*n,C1.o);
END;
```

Als Ergebnis haben wir nun für das Netzwerk mit `n1*n` Zweierschaltern, `n1` Stufen und den Eingaben `a:n`, `b:n` die letzten `2*n` Elemente aus dem Komponenten `C1`, die die letzte Stufe im Omega-Netzwerk ist.

Der Hardwaretyp *GENERIC* unterscheidet sich von *RECURSIVE* dahingehend, daß *GENERIC* hauptsächlich eine „auslösende Rolle“ hat. Das heißt, so wie der Hardwaretyp *MODULE* den Hardwaretyp *BASIC* verwendet und somit in Richtung des Hardwaretyps *BASIC* eine „auslösende“ Wirkung hat, ist es auch mit den Hardwaretypen *RECURSIVE* und *GENERIC*. Im Hardwaretyp *RECURSIVE* werden Hardwarebeschreibungen, die mit dem Hardwaretyp *MODULE* beschrieben worden sind, verwendet.

Wir haben nun gesehen wie die Hardwaretypen *BASIC*, *MODULE*, *RECURSIVE* und *GENERIC* miteinander zusammenhängen. Ein Schaubild (Abbildung 3.11) soll die Beziehung als „*wird verwendet von*“ zeigen.

3.3.5 INSTANCE

Wie schon in Abschnitt **3.3.3 RECURSIV** angedeutet, wird hier der generisch beschriebene Carry-Ripple-Addierer instantiiert. Die allgemeine Struktur dieses Hardwaretyps sieht wie folgt aus:

```
INSTANCE Funktionsname (Liste von Termen)
```

Komponente$_0$;
⋮
Komponente$_k$;

```
END;
```

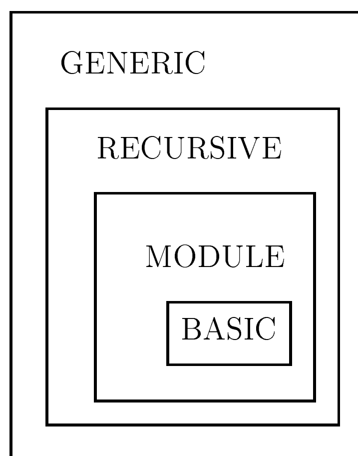

Abbildung 3.11: Beziehung zwischen den Hardwaretypen

Dabei werden die Eigenschaften wie zum Beispiel `SPEC` oder `OUTPUT` aus den Hardwarebeschreibungen, die im Komponentenblock vorkommen, vererbt.

Dazu wird das Programm, das mit dem Hardwaretyp $RECURSIVE$ erstellt wurde mit der vorangestellten Bezeichnung $INSTANCE$ und einer Instantiierung der Induktionsvariablen n neu geschrieben. Die Induktionsvariable wird mit $n = 9$ instantiiert. Die Listenlängen stimmen hier mit der Induktionsvariable bzw. Rekursionszahl überein (allgemein muß dies nicht der Fall sein). Die restlichen Vorkommen der Induktionsvariable und der Listenlängen wird automatisch durch GHDL ersetzt und in jedem Rekursionschritt aktualisiert.

```
INSTANCE CarryRippleAdder_9(l1:9,l2:9,carry:1)
      C1: CarryRippleAdder_9(9;l1:9, l2:9, carry);
END;
```

Nach dem Abrollen der generischen Hardwarebeschreibung, bekommen wir als Ergebnis einen Hardwaretyp $MODULE$. Dies ist auch die Richtung, die bei jeder Instantiierung eingeschlagen wird (Abbildung 3.12).

```
MODULE CarryRippleAdder_9(a0,a1,a2,a3,a4,a5,a6,a7,a8,b0,b1,b2,
                          b3,b4,b5,b6,b7,b8,cin)
      C1: FullAdder(a0,b0,cin);
      C2: FullAdder(a1,b1,Cout.1);
      C3: FullAdder(a2,b2,Cout.2);
      C4: FullAdder(a3,b3,Cout.3);
      C5: FullAdder(a4,b4,Cout.4);
      C6: FullAdder(a5,b5,Cout.5);
```

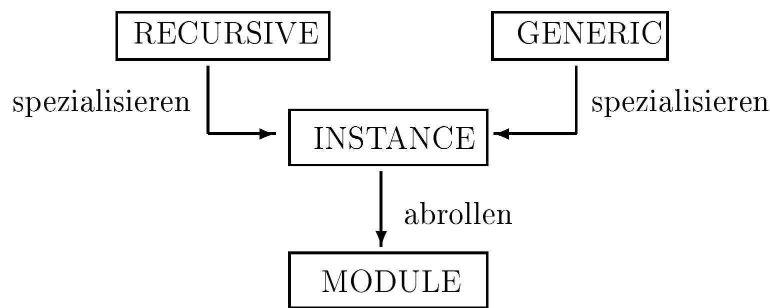

Abbildung 3.12: Instantiierung

```
      C7: FullAdder(a6,b6,Cout.6);
      C8: FullAdder(a7,b7,Cout.7);
      C9: FullAdder(a8,b8,Cout.8);
OUTPUT
      sum1 := Csum.1;
      sum2 := Csum.2;
      sum3 := Csum.3;
      sum4 := Csum.4;
      sum5 := Csum.5;
      sum6 := Csum.6;
      sum7 := Csum.7;
      sum8 := Csum.8;
      sum9 := Csum.9;
      cout := Cout.9;
DATAEQUATION
      f1 := ( [cout,sum9,sum8,sum7,sum6,sum5,sum4,sum3,sum2
              ,sum1] = addition(a,b,carry));
SPEC
      G f1;
END;
```

Eine komplette Beschreibung eines Carry-Ripple-Addierers liegt nun vor uns. Über 1-Bit-Halbaddierer sind wir auf 1-Bit-Volladdierer gekommen, mit deren Hilfe dann schließlich der allgemeine, generische Carry-Ripple-Addierer aufgebaut wurde. Zum Schluß instantiierten wir die rekursive Hardwarebeschreibung, um sie abzurollen zu einer konkreten Hardwaredarstellung.

Ein Vergleich soll dieses Kapitel abschließen. Die vorliegende Hardwarebeschreibungssprache GHDL hält tatsächlich das in Kapitel 2 versprochene. GHDL ist einfach zu programmieren und hat überdies einen recht kleinen Programmumfang. Die Verständlichkeit dürfte auch nichts zu wünschen übrig

	LUSTRE	SML	HOL	VHDL	GHDL
Programm-umfang	gering	gering	gering	groß	gering
Verständlich-keit	gut	befriedigend	schlecht	befriedigend	sehr gut
Hardware-bezug	gut	wenig	wenig	sehr gut	sehr gut
Zeit-modellierung	synchron	nicht direkt	vorhanden	vorhanden	synchron
generisch ?	ja	ja	ja	ja	ja
Verifikation	nein	nein	ja	nein	ja

Abbildung 3.13: Beschreibungssprachen im Vergleich

lassen. Der Hardwarebezug der Programme entspricht fast einer bijektiven Abbildung. Die Zeitmodellierung wird auch eingehalten, so daß gesagt werden kann, daß eine gut strukturierte, übersichtliche und generische Hardwarebeschreibungssprache implementiert worden ist. Abbildung 3.13 veranschaulicht diesen Sachverhalt.

Die zusätzliche Verifikationsmöglichkeit unterscheidet GHDL von den anderen ausgewählten Sprachen. Die Verifikation wird durch das Vorhandensein der Implementierung und Spezifikation in GHDL gewährleistet. In den anderen Sprachen hat man nur die Möglichkeit die Hardwarestruktur zu beschreiben, ohne auf die Verifikation einzugehen. Dabei haben wir sehen können, daß der Hardwarebezug bei den anderen Sprachen manchmal schwer zu erkennen war.

Kapitel 4

Benchmark-Schaltungen

Angefangen vom Hardwaretyp $BASIC$ bis zur obersten Hierachie der Hardwaretypen, $GENERIC$, haben wir an Beispielen gesehen, wie einfach es ist, mit GHDL Hardwarestrukturen programmtechnisch zu modellieren und somit der Verifikation zugänglich zu machen. Die Lesbarkeit der Programme ist relativ einfach. Dies dürfte auch auf das Verständnis der Programme zutreffen.

In diesem Kapitel werden Benchmark-Schaltungen exemplarisch vorgestellt, die mit GHDL programmiert wurden.

4.1 Island Tunnel Controller

Dieses Steuerproblem entspricht einer Ampelsteuerung [**?**]. In einem Einbahnstraßentunnel zwischen dem Festland und der Insel wird konkurrierend ein- und ausgefahren. Zu jedem Zeitpunkt können mehrere Fahrzeuge auf der Insel und auf dem Festland sein. Die Schaltung hat die Kontrolle über Ampeln am Ende und am Anfang des Tunnels. Von der Ampel wird erwartet, daß sie nicht zur selben Zeit an beiden Enden des Tunnels grün ist. Falls ein Fahrzeug in den Tunnel fahren möchte, sollte dies auch nach einer gewissen Zeit gewährleistet sein. Das System soll also lebendig sein. Ferner sollte der Verkehr auf der Insel zu keinem Zeitpunkt mehr als n Fahrzeuge haben.

Es gibt vier Sensoren, um Fahrzeuge bestimmen zu können. Ein Sensor befindet sich am Tunneleingang auf der Insel *il_enter*, ein weiterer am Tunnelausgang auch auf der Insel *il_exit*. Die zwei anderen Sensoren befinden sich auf der Seite des Festlandes *ml_enter* und *ml_exit*. Die Steuerung hat zwei Zähler *IC* und *TC*. Für das Zählen der Fahrzeuge im Tunnel wird *TC* verwendet. Der Inselzähler *IC* zählt dabei die Fahrzeuge auf der Insel oder Fahrzeuge, die im Tunnel mit Fahrrichtung Insel unterwegs sind. Für den Entwurf braucht man die folgenden Annahmen:

A1: Fahrzeuge werden im Tunnel nicht hergestellt, d.h. wenn im Tunnel kein Fahrzeug ist, kann auch keines den Tunnel verlassen.
A2: Fahrzeuge verschwinden nicht im Tunnel.
A3: Fahrzeuge werden auch nicht auf der Insel produziert.
A4: Der Inselzähler zählt die Fahrzeuge, die auf der Insel sind. Der Inselzähler zählt auch solche Fahrzeuge, die mit Fahrtrichtung Insel im Tunnel sind.
A5: Jedes Fahrzeug verläßt einmal die Insel. Zu diesem Zeitpunkt darf kein Fahrzeug auf der Festlandseite den Tunnel betreten.
A6: Wenn die Ampel auf der Insel grün ist, verlassen nur die Fahrzeuge den Tunnel, die auf der Festlandseite sind, bis die Ampel auf dem Festland grün wird. (dasselbe gilt für die andere Richtung)
A7: Falls ein Fahrzeug auf einer Seite mit einer grünen Ampel den Tunnel befahren möchte, so soll dies nach einer Weile möglich sein.

Die Annahmen können als temporallogische Ausdrücke formuliert werden:

A1: $G\ [TC = 0 \rightarrow \neg il_exit \wedge \neg ml_exit]$
A2: $G\ [TC \neq 0 \rightarrow F\ [il_exit \vee ml_exit]]$
A3: $G\ [IC = 0 \rightarrow \neg il_enter]$
A4: $G\ [IC = TC \rightarrow \neg il_enter]$
A5: $G\ [IC \neq 0 \rightarrow F\ il_enter]$
A6: $G\ [(il_green \rightarrow [(\neg il_exit)\ U\ ml_green]) \wedge$
$(ml_green \rightarrow [(\neg ml_exit)\ U\ il_green])]$
A7: $G\ [(il_enter] \wedge il_freen \wedge X\ il_green \rightarrow\ F\ \neg il_enter) \wedge$
$(ml_enter \wedge ml_green \wedge X\ ml_green \rightarrow F\ \neg ml_enter)]$

Einige der Spezifikationen hängen von der maximalen Anzahl von Fahrzeugen ab, die auf der Insel sein dürfen oder von der Anzahl der Fahrzeuge, die im Tunnel sein dürfen. Das Modell für die Realisierung ist in Abbildung 4.1 veranschaulicht.

S1: Zu keinem Zeitpunkt sind beide Ampeln grün.
S2: Zu keinem Zeitpunkt sind mehr als n Fahrzeuge auf der Insel.
S3: Wenn ein Fahrzeug in den Tunnel fahren möchte, dann hat es die Chance dies nach einer endlichen Zeit zu tun.
S4: Die Ampeln wechseln ihre Zustände nur dann, wenn der Tunnel leer ist.

Die formale Spezifikation dazu sieht dann wie folgt aus:

S1: $G \neg [ml_green \wedge il_green]$
S2: $G[IC \leq n]$
S3: $G[\neg G[il_enter \wedge il_red] \wedge \neg [ml_enter \wedge ml_red]]$
S4: $G[(ml_red \wedge X\ ml_green \rightarrow TC = 0) \wedge$

$(il_red \wedge X\ il_green \rightarrow TC = 0)]$

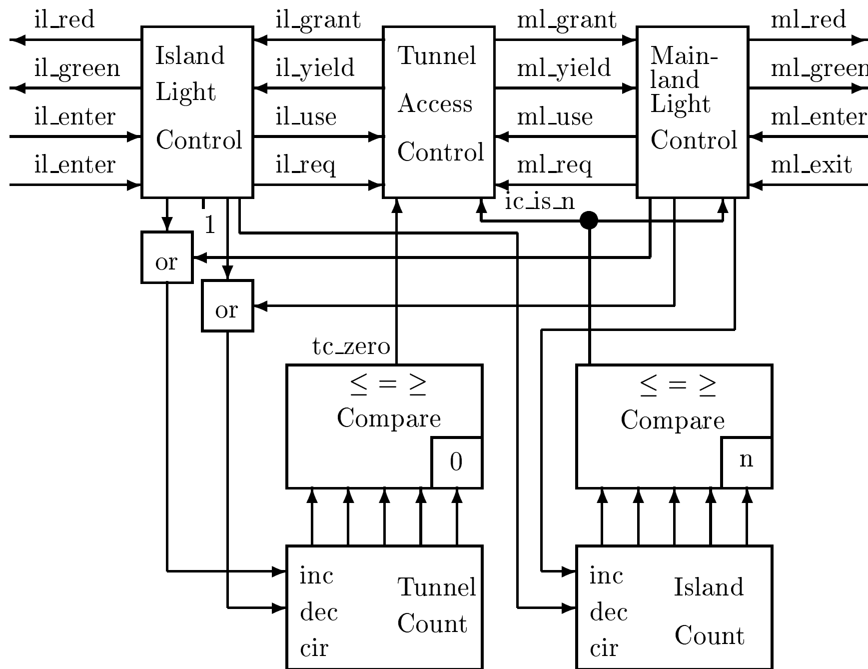

Abbildung 4.1: Island Tunnel Controller

Eine ausführlichere Darstellung von diesem Sachverhalt ist in [?] zu finden.

Das GHDL-Programm zu Island Tunnel Controller befindet sich im Anhang.

4.2 Arbiter

Ein Arbiter steuert Komponenten, die um eine gemeinsame Ressource, z.B. einen Bus, Anfragen stellen. Die Steuerung berücksichtigt alle Anfragen und gewährt nur einem Komponenten die Benutzung an der gemeinsamen Ressource. Die Schaltung hat n Anfragen/Eingaben $rep_0, req_1, req_2, ..., rep_{n-1}$ und n Ausgaben $ack_0, ack_1, ack_2, ..., ack_{n-1}$. Es wird angenommen, daß jeder Zugriff nicht länger als eine Zeiteinheit dauert. Damit wird sichergestellt, daß zu jedem Zeitpunkt eine neue Komponente auf die Ressource Zugriff bekommt. Weiterhin wird verlangt, daß so lange angefragt wird bis die Ressource zugeteilt wird oder nicht mehr benötigt wird. Der Sachverhalt wird in Abbildung 4.2 dargestellt.
Die Arbiter-Schaltung mit mehreren Stufen zeigt Abbildung 4.3. Die Implementierung für n Komponenten besteht aus n Zellen, die auf die gleiche Art

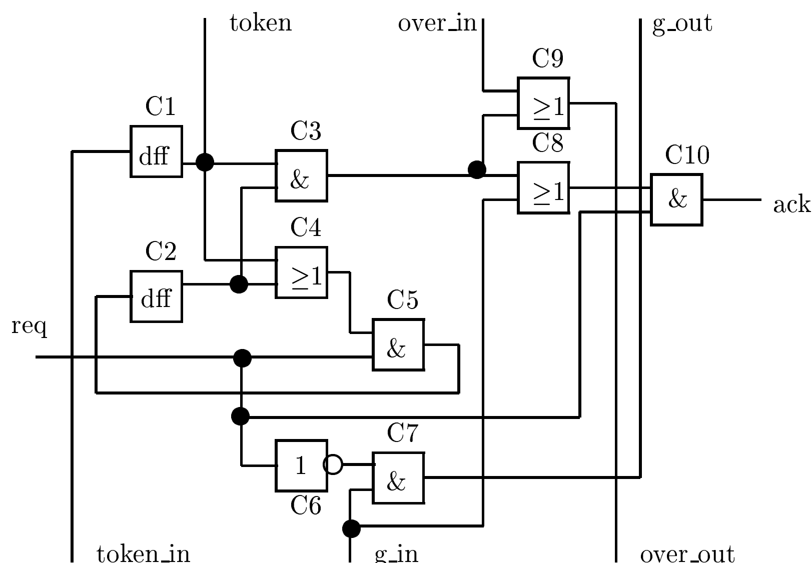

Abbildung 4.2: Implementierung eines Arbiters

und Weise aufgebaut sind, bis auf den ersten Komponenten. In der Schaltung befindet sich ein Token, der in der Schaltung zirkuliert. Bei der Initialisierung der Zellen wird einer Zelle das Token zugeteilt, da ansonsten jede Komponente auf das Token seines Nachbarn warten würde. Das würde dazu führen, das sich das System verklemmen würde. Neben dem Token gibt es im System noch eine Marke, die dann gesetzt ist, wenn die entsprechende Komponente Zugriff zur Ressource haben möchte. Wenn die Zelle mit dem Token die Marke nicht gesetzt hat, wird der Zugriff auf die Ressource statisch berechnet. Wenn dieser Fall eintritt, bekommt die Komponente mit dem niedrigsten Index den Zugriff.

Der Arbiter muß im einzelnen die vier Forderungen erfüllen:

F1: Zu jedem Zeitpunkt hat nur eine Komponente Zugriff auf die Ressource.
F2: Nur anfordernde Komponenten haben Zugriff auf die Ressource.
F3: Die Zuteilung der Ressource ist fair.
F4: Wenn zu einem Zeitpunkt keine Komponente eine Anforderung stellt, wird die Zuteilung zum nächsten Zeitpunkt statisch gelöst.

Das GHDL-Programm zum Arbiter ist im Anhang.

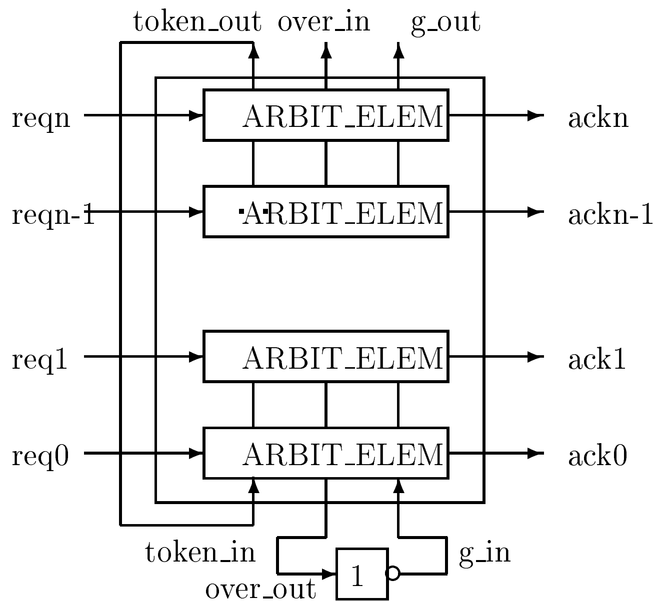

Abbildung 4.3: Verkettete Arbiter-Stufen

4.3 Safe-Box

Dieses Beispiel basiert auf dem chinesischen Ring-Puzzle. Eine Safe-Box hat n Knöpfe $k_{n-1}, ..., k_0$. Jeder Knopf besitzt zwei mögliche Positionen. Der Zustand 0 für „auf“ und 1 für „zu“. Die Knöpfe können nicht unabhängig voneinander gedreht werden.

1. Am Anfang sind alle Knöpfe in geschlossenem Zustand.

2. Der am weteitesten links liegende Knopf k_{n-1} kann immer gedreht werden.

3. Wenn k_{n-1} nicht gedreht wird, kann nur der rechte Nachbar zu einem geschlossenen Knopf von links gesehen gedreht werden.

4. Wenn der letzte Knopf k_0 der einzigste geschlossene Knopf ist, kann die obige Regel nicht angewandt werden. Die einzigste verbleibende Möglichkeit besteht darin, Knopf k_{n-1} umzudrehen.

5. Zu einem Zeitpunkt kann nur ein Knopf gewendet werden.

Die Aufgabe besteht darin, die Safe-Box zu öffnen. Dazu müssen alle Knöpfe den Zustand „auf“ einnehmen. Das chinesische Ring-Puzzle kann als Benchmark verwendet werden, indem man das Gegenteil annimmt und behauptet, daß die Safe-Box nicht geöffnet werden kann. Das Verifikationssystem muß dann zeigen, daß die Behauptung falsch ist. Dazu muß das Verifikationssystem

eine Sequenz von Drehvorgängen zeigen, mit der es möglich ist, die Safe-Box zu öffnen. Die minimale Länge der Lösungssequenz $l(n)$ für das Safe-Box-Problem mit n Knöpfen, hat die Länge $O(2^n)$.

Ein Knopf k_i wird zu einem Zeitpunkt gewendet, genau dann wenn

$$Xk_i = \neg k_i$$

zu dem Zeitpunkt erfüllt ist.

Lemma: *Das n-Knopf Safe-Box-Problem wird durch die folgende Transitionsgleichungen der Zustände beschrieben:*

$$(\bigwedge_{i=0}^{n-1}(k_i = T)) \wedge (\bigwedge_{i=0}^{n-2} G(Xk_i = \neg k_i = \bigwedge_{j=i+2}^{n-1} \neg k_j))$$

Dabei gelten folgende Fakten für die Zustände;

1. *Jeder Zustand hat höhstens zwei Vorgänger. Eine davon ist das Wenden des am weitesten links liegenden Knopfes k_{n-1}. Der zweite Zustand entsteht durch das Wenden von Knopf k_i, wenn $k_{i+1} \wedge \bigwedge_{j=i+2}^{n-1} \neg k_j$ zutrifft.*
2. *Die einzigen zwei Zustände, die nur einen Vorgänger haben, sind (0, ..., 0) und (0, ..., 0,1).*
3. *Wenn ein Zustand s_1 Vorgänger von Zustand s_0 ist, so gilt auch daß s_0 Vorgänger von Zustand s_1 ist.*
4. *Die minimale Länge einer Knopfsequenz, um ein n-Knopf Safe-Box zu öffnen ist:*

$$l(n) = \begin{cases} \frac{2}{3} * (2^n - \frac{1}{2}) & : \ n \ \ ungerade \\ \frac{2}{3} * (2^n - 1) & : \ n \ \ gerade \end{cases}$$

Die Beweise zu diesem Lemma befinden sich in [**?**].

In Abbildung 4.4 wird das Zustandsdiagramm für das Öffnen des Safe-Box mit 4 Knöpfen gezeigt. Die Transitionsgleichungen aus dem Lemma können als eine synchrone Schaltung bestehend aus den n Zellen für die n Knöpfe, angesehen werden. Jede Zelle hat die Eingaben *l_node, l_conj_in* und die Ausgaben k_i und *l_conj_out*, die folgendermaßen definiert sind:

$$k_i = T \wedge G[Xk_i = \neg k_i = l_node \wedge l_conj_in]$$
$$l_conj_out = \neg l_node \wedge l_conj_in$$

Die interne Implementierung (Abbildung 4.5) einer Zelle kann somit aus dem Lemma und der entsprechenden Beschreibung abgeleitet werden.

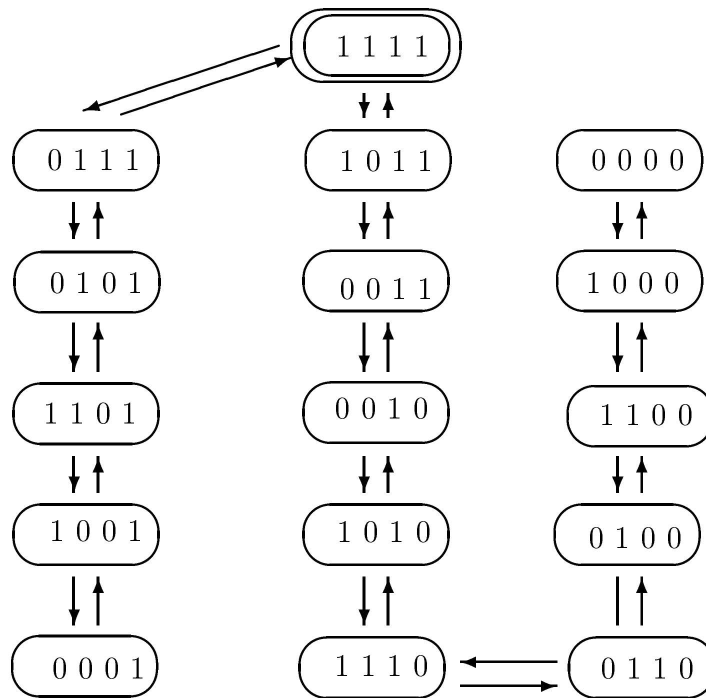

Abbildung 4.4: Safe-Box-Zustandsdiagramm für 4 Knöpfe

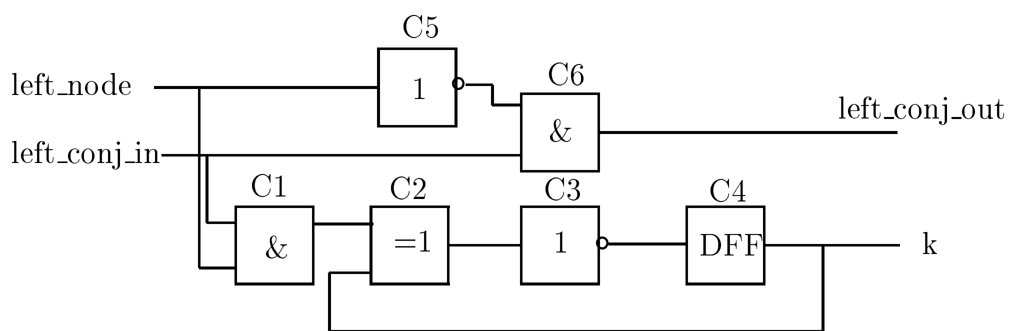

Abbildung 4.5: Interne Zellendarstellung

Kapitel 5

Ausblick

Generische Hardwarebeschreibungsprachen werden sich wahrscheinlich in der Zukunft industriell etablieren. Die Entwicklung im Hardwaresektor drängt immer mehr nach Geschwindigkeitssteigerung sowohl in der Verarbeitung von Daten als auch in der Herstellung der Hardwarekomponenten. Die Preise fallen bei stetiger Leistungssteigerung. Damit diese Tendenz auch in der Zukunft anhalten kann, müssen in der Entwicklung die notwendigen Voraussetzungen geschaffen werden. Eine der notwendigen Voraussetzungen scheint es zu sein, wiederverwendbare und leicht wartbare Module bereitzustellen.

Gerade diese Forderung wird durch generische Hardwarebeschreibungen angestrebt, eine allgemeine und somit wiederverwendbare Beschreibung von Hardwarestrukturen zu erhalten, die einmal entworfen und einmal verifiziert werden und für spezielle Anwendungen instantiiert werden können.

Zwischen den vorgestellten Sprachen dürfte wohl GHDL eine gute Wahl sein. Eigenschaften wie

- leichte Lesbarkeit,
- leichte Erweiterbarkeit,
- wenige Befehle,
- Strukturiertheit und
- Möglichkeit zur Verifikation

spielen dabei ein wichtige Rolle.

Literaturverzeichnis

[1] F. Balarin and G. York. Verilog HDL modeling styles for formal verification. In D. Agnew, L. Claesen, and R. Camposano, editors, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 439–452, Ottawa, Canada, April 1993. IFIP WG10.2, CHDL'93, IEEE COMPSOC, Elsevier Science Publishers B.V., Amsterdam, Netherland.

[2] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *International Conference on Theorem Provers in Circuit Design (TP-CD)*, pages 129–156, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[3] B.C. Brock, W.A. Hunt, and W.D. Young. Introduction to a formally defined hardware description language. In *International Conference on Theorem Provers in Circuit Design (TPCD)*, pages 3–35, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[4] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[5] E.M. Clarke and E.A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag.

[6] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[7] R. de Simone and A. Ressouche. Compositional semantics of Esterel and verification by compositional reductions. In David L. Dill, editor,

Conference on Computer Aided Verification (CAV), volume 818 of *Lecture Notes in Computer Science*, pages 441–454, Standford, California, USA, June 1994. Springer-Verlag.

[8] E.A. Emerson. Alternative semantics for temporal logics. *Theoretical Computer Science*, 26:121–130, 1983.

[9] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[10] E.A. Emerson and J.Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[11] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative Temporal Reasoning. *Journal of Real-Time Systems*, 4:331–352, 1992.

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

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, sep 1991.

[14] L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of Esterel programs and applications to telecommunications software. In P. Wolper, editor, *Conference on Computer Aided Verification (CAV)*, volume 939 of *Lecture Notes in Computer Science*, pages 127–140, Liege, Belgium, July 1995. Springer Verlag.

[15] J.J. Joyce. Generic structures in the formal specification and verification of digital circuits. In G.J. Milne, editor, *Fusion of Hardware Design and Verification*, pages 51–76, Glasgow, Scotland, July 1988. IFIP WG 10.2, North-Holland.

[16] M. Langewin and E. Cerny. Comparing generic state machines. In K. G. Larsen and A. Skou, editors, *Workshop on Computer Aided Verification (CAV).* Springer Verlag, July 1991.

[17] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 12 edition, 1993.

[18] K.L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[19] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[20] P. Moorby. History of Verilog. *IEEE Design & Test of Computers*, pages 62–63, September 1992.

[21] J. O'Leary, M. Linderman, M. Leeser, and M. Aagaard. HML: A hardware description language based on standard ML. In D. Agnew, L. Claesen, and R. Camposano, editors, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 313–320, Ottawa, Canada, April 1993. IFIP WG10.2, CHDL'93, IEEE COMPSOC, Elsevier Science Publishers B.V., Amsterdam, Netherland.

[22] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer Verlag, 1994.

[23] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[24] F. Rocheteau and N. Halbwachs. Pollux, a Lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, 1991.

[25] K. Schneider, T. Kropf, and R. Kumar. Control-Path Oriented Verification of Sequential Generic Circuits with Control and Data Path. In *European Design and Test Conference (EDTC)*, pages 648–652, Paris, France, March 1994. IEEE Computer Society Press.

[26] K. Schneider, T. Kropf, and R. Kumar. Control-path oriented verification of sequential generic circuits with control and data path. Technical Report SFB358-C2-9/94, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 1994. ftp://goethe.ira.uka.de/pub/hvg/techreports/SFB358-C2-9-94.ps.gz.

[27] K. Schneider, R. Kumar, and T. Kropf. Modelling generic Hardware Structures by Abstract Datatypes. In L. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, pages 419–429, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, Elsevier Science Publishers.

[28] M. Shahdad, R. Lipsett, E. Marschner, K. Sheehan, and H. Cohen. VHSIC hardware description language. *IEEE Computer*, pages 94–103, February 1985.

[29] L. Sterling and E. Shapiro. *The Art of PROLOG.* MIT Press, 1986.

[30] E. Sternheim, R. Singh, and Y. Trivedi. *Digital Design with Verilog HDL*. Design Automation Series. Automata Publishing Company, 1990.

[31] P.J. Windley. *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Division of Computer Science, Davis, June 1990.

[32] P.J. Windley. A theory of generic interpreters. In *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 122–134, Arles,France, May 1993. Springer-Verlag.

Anhang A

GHDL-Programme zu Kapitel 4

A.1 Island Tunnel Controller

Nun soll nach einer Vorstellung des Problems das GHDL-Programm folgen. Die Implementierungen für die Steuerung an beiden Tunnelenden kann wie folgt programmiert werden:

```
BASIC SideCntrl(reset,enter,exit,grant,yield,res_level)
   INIT(red) := 1;
   NEXT(red) := (red & ~exit & ~grant)
              | (exiting & ~exit)
              | (green & res_level & yield)
              | (green & ~res_level)
              | reset;
   INIT(green) := 0;
   NEXT(green) := ~reset &
              [ (red & ~exit & grant)
              | (green & res_level & ~yield & ~enter)
              | (entering & ~enter)
              ];
   INIT(entering) := 0;
   NEXT(entering) := ~reset &
              [ (green & res_level & ~yield & enter)
              | (entering & enter)
              ];
   INIT(exiting) := 0;
   NEXT(exiting) := ~reset & [(red | exiting) & exit];
OUTPUT    green_light := green | entering;
            # turn green light on
   red_light := red | exiting;
            # turn red light on
```

```
   tl_use := green | entering;
           # tunnel is currently controlled
   tl_req := red & enter;
           # tunnel control   is requested
   tc_dec := red & exit;
           # tunnel counter decrement
   tc_inc := green & res_level & ~yield & enter;
           # tunnel counter increment
   ic_change := green & res_level & ~yield & enter;
           # island counter change
END;

MODULE SideCntrlCheck(reset,enter,exit,grant,yield,res_level)
     C0 : SideCntrl(reset,enter,exit,grant,yield,res_level);
   OUTPUT
     red := C0.red;
     green := C0.green;
     entering := C0.entering;
     exiting := C0.exiting;
     green_light := C0.green_light;
     red_light := C0.red_light;
     tl_use := C0.tl_use;
     tl_req := C0.tl_req;
     tc_dec := C0.tc_dec;
     tc_inc := C0.tc_inc;
     ic_change := C0.ic_change;
   SPEC red;
   SPEC G[red | green | entering | exiting];
   SPEC G[red -> ~green & ~entering & ~exiting];
   SPEC G[green -> ~red & ~entering & ~exiting];
   SPEC G[entering -> ~green & ~red & ~exiting];
   SPEC G[exiting -> ~green & ~red & ~entering];
END;
```

Nachdem wir nun die Steuerung an den beiden Enden der Tunnels haben, können wir uns dem Zugang in den Tunnel zuwenden.

```
BASIC TunnelCntrl(reset,il_use,il_req,ml_use,ml_req,ic_ls_16,tc_zero)
   INIT(dispatch) := 1;
   NEXT(dispatch) := reset
            | ilclear & tc_zero
            | mlclear & tc_zero
            | dispatch &
```

```
            (~il_req & ~ml_req
            | il_req & ~ml_use & tc_zero
            | ~il_req & ml_req & tc_zero & ic_ls_16 & ~il_use
            | ~il_req & ml_req & ~ic_ls_n
            );    INIT(iluse) := 0;
   NEXT(iluse) := ~reset &
            [ iluse & il_use
            | dispatch & ~il_req & ml_req & ic_ls_n & il_use
            ];
   INIT(mluse) := 0;
   NEXT(mluse) := ~reset &
            [ mluse & ml_use
            | dispatch & il_req & ml_use
            ];
   INIT(ilclear) := 0;
   NEXT(ilclear) := ~reset &
            [ iluse & ~il_use
            | ilclear & ~tc_zero
            | dispatch & ~il_req & ml_req & ic_ls_16 & ~il_use &
~tc_zero
            ];
   INIT(mlclear) := 0;
   NEXT(mlclear) := ~reset &
            [ mluse & ~ml_use
            | mlclear & ~tc_zero
            | dispatch & il_req & ~ml_use & ~tc_zero
            ];
OUTPUT
   il_grant := (dispatch & il_req & ~ml_use & tc_zero)
            | (mlclear & tc_zero);
   il_yield := iluse;
   ml_grant := (dispatch & ~il_req & ml_req & ic_ls_16 & ~il_use &
tc_zero)
            | (ilclear & tc_zero);
   ml_yield := mluse;
END;
```

Jetzt nur noch die Verkehrskontrolle auf der Insel sicherstellen. Dann wird die generische Realisierung dieses Problems für n Fahrzeuge gezeigt.

```
MODULE IslandTrafficCntrl(reset,ml_enter,ml_exit,il_enter,il_exit,
                      tc_zero,ic_ls_max)
   IslandCntrl : SideCntrl(reset,il_enter,il_exit,C1.il_grant,
```

```
C1.il_yield,1);
   MainlandCntrl : SideCntrl(reset,ml_enter,ml_exit,C1.ml_grant,
                    C1.ml_yield,ic_ls_max);
   C1 : TunnelCntrl(reset,IslandCntrl.tl_use,IslandCntrl.tl_req,
                     MainlandCntrl.tl_use,MainlandCntrl.tl_req,
                     ic_ls_max,tc_zero);
   C2 : Or2(IslandCntrl.tc_inc,MainlandCntrl.tc_inc);
   C3 : Or2(IslandCntrl.tc_dec,MainlandCntrl.tc_dec);
OUTPUT
   il_red := IslandCntrl.red_light;
   il_green := IslandCntrl.green_light;
   ml_red := MainlandCntrl.red_light;
   ml_green := MainlandCntrl.green_light;
   tc_inc := C2.o;
   tc_dec := C3.o;
   ic_inc := MainlandCntrl.ic_change;
   ic_dec := IslandCntrl.ic_change;
END;
```

Nun zur generischen Darstellung. Der n-Bit Zähler für die Fahrtrichtungen `UD_Counter_n` und die eigentliche Steuervorrichtung `IslandTraffic_n` können nun als GHDL-Programm gezeigt werden.

```
GENERIC UD_Counter_n(n;incr:n,decr:n,reset)
   C0: XOR2(1;incr,decr);
   C1: Inc_n(n;C4.o);
   C2: Dec_n(n;C4.o);
   C3: Mux_n(n;C1.o,C2.o);
   C4: Reg_n(n;reset,C0.o,C3.o);
OUTPUT
     o := [C4.o];
END;
```

```
GENERIC IslandTraffic_n(n; reset,ml_enter,ml_exit,il_enter,il_exit)
   TrafficCntrl : IslandTrafficCntrl(reset,ml_enter,ml_exit,
                    il_enter,il_exit, TLCompare.eq,ILCompare.ls);
   TLCount : UD_Counter_n(n;TrafficCntrl.tc_inc,TrafficCntrl.tc_dec,
                    reset);
   ILCount : UD_Counter_m(n;TrafficCntrl.ic_inc,TrafficCntrl.ic_dec,
                    reset);
   ZeroSeq : ProduceZero_n(n-1;0);
   OneSeq : ProduceOne_n(n-1;1);
   TLCompare : Compare_n(n;TLCount.o,0,ZeroSeq.o);
```

```
   ILCompare : Compare_n(n;ILCount.o,1,ZeroSeq.o);
OUTPUT
   il_red := TrafficCntrl.il_red;
   il_green := TrafficCntrl.il_green;
   ml_red := TrafficCntrl.ml_red;
   ml_green := TrafficCntrl.ml_green;
   tc_zero := TLCompare.eq;
   ic_ls_max := ILCompare.ls;
   ic_eq_max := ILCompare.eq;
   ic := [ILCount.o];
   tc := [TLCount.o];
END;
```

A.2 Arbiter

Die Beschreibung der Zelle_0 kann mit GHDL folgendermaßen gemacht werden.

```
MODULE ArbiterCell0(req,token_in,g_in,over_in)
     C1 : Dflipflop0(token_in);
     C2 : Dflipflop0(C5.o);
     C3 : And(C1.o1,C2.o1);
     C4 : Or(C1.o1,C2.o1);
     C5 : And(C4.o,req);
     C6 : Not(req);
     C7 : And(C6.o,g_in);
     C8 : Or(C3.o,g_in);
     C9 : Or(over_in,C3.o);
     C10 : And(C8.o,req);
  OUTPUT
     token_out := C1.o1;
     over_out := C9.o;
     g_out := C7.o;
     ack := C10.o;
     pers := C2.o1;
   SPEC G[ ack = req & [(token_out & pers) | g_in ]];
   SPEC G[ over_out = over_in | (token_out & pers)];
   SPEC G[ g_out = g_in & ~req];
   SPEC G[ X token_out = token_in];
   SPEC G[ X pers = req & (pers | token_out)];
   SPEC ~token_out & ~pers;
END;
```

Zelle$_1$, Zelle$_2$, Zelle$_{n-1}$ sind vom Aufbau her alle identische, so daß gleich der Ansatz mit der generischen Programmierung gemacht werden kann. Die verketteten Zellen werden durch das Programm `RECURSIVE ArbitChain_n` dargestellt.

```
RECURSIVE ArbitChain_n(n;r:n,token_out,g_out,over_out)
      IF (n>1) THEN
        C1: ArbiterCell0(1;HD(r),token_out:1,g_out:1,over_out:1);
        C2: ArbitChain_n(n-1;TL(r),C1.token_out,C1.g_out,over_out);
        OUTPUT
          ack := CONS(C1.ack,C2.ack);
      ELSE
        C1: ArbiterCell1(1;HD(r),token_out,over_out);
        OUTPUT
          ack := [C1.ack];
      END
END;
```

Die gesamte Beschreibung des Szenarios sieht dann so aus:

```
GENERIC Arbiter_n(n;req:n)
      C1: ArbitChain_n(n;req:n,C3.o:1,C2.o:1,0);
      C2: Not(1;C1.over_out);
      C3: Id(1;C1.token_out);
      OUTPUT
        ack := [C1.ack];
END;
```

A.3 Safe-Box

```
MODULE CrackerCell(left_node,left_conj_in)
      C1 : And(left_node,left_conj_in);
      C2 : Xor(C1.o,C4.o2);
      C3 : Not(C2.o);
      C4 : Dflipflop(C3.o);
      C5 : Not(left_node);
      C6 : And(left_conj_in,C5.o);
   OUTPUT
      k := C4.o2;
      left_conj_out := C6.o;
      SPEC k;
      SPEC G[X k = ~k = left_node & left_conj_in];
      SPEC G[left_conj_out = left_conj_in & ~left_node];
END;
```

Hier noch ein paar ausgesuchte Spezialfälle:

```
MODULE SafeBox2Equiv(n1)
      C0 : CrackerCell(n1,1);
   OUTPUT
      n0 := C0.k;
   SPEC G[n1 | n0];
END;

MODULE SafeBox3Equiv(n2)
      C1 : CrackerCell(n2,1);
      C0 : CrackerCell(C1.k,C1.left_conj_out);
   OUTPUT
      n1 := C1.k;
      n0 := C0.k;
   SPEC G[n2 | n1 | n0];
END;
```

Generische Beschreibung der Hardwarestruktur:

```
RECURSIVE CrackerCellHelp_n(n)
      IF (n>1) THEN
           C1: CrackerCell(1;C2.k, C2.left_conj_out);
           C2: CrackerCell(1;CrackerCellHelp_n(n-1);
           OUTPUT
             o := CONS(C1.o, C2.o);
```

```
     ELSE
          C1: CrackerCell(1;nx,1);
          OUTPUT
            o := [C1.o, C2.o];
     END
END;

GENERIC SafeBoxNequiv(n; nx)
     C1: CrackerCellHelp_n(n);
     OUTPUT
          o := [C1.o];
END;
```

Anhang B

Grammatik

```
%token TRUE
%token FALSE
%token VAR

%token THIS
%token DOT
%token ADD
%token SUB

%token NEG
%token NEXT
%token ALWAYS
%token EVENTUAL
%token ALLPATH
%token ONEPATH

%left XOR EQUIV
%left IMPLIES
%left OR
%left AND
%left WHEN
%left SWHEN
%left UNTIL
%left SUNTIL
%left BEFORE
%left SBEFORE
%left ATNEXT
%left SATNEXT

%token LET IN
```

```
%token IF THEN ELSE

%token DEL11 DEL12
%token DEL21 DEL22
%token SEMICOLON

%token BASIC
%token MODULE
%token LEMMA
%token RECURSIVE
%token GENERIC
%token INSTANCE
%token BIGVAR
%token INIT
%token NXXT
%token ASSIGN
%token DEFINE
%token OUTPUT
%token END
%token DATAEQUATIONS
%token SPEC

%left COMMA
%left COLON

%token BIG
%token LESS

%token CONS
%token APPEND
%token HD
%token TL
%token FIRSTN
%token LASTN
%token BUTLASTN
%token BUTFIRSTN
%token INC

%token NUMBER
%left MINUS
%left PLUS
%left DIV
%left MULT
```

```
%token FIRST_COMPLEMENT
%token SECOND_COMPLEMENT
%token IS_ZERO_LIST
%token IS_ONE_LIST
%token NOT_LIST
%token ONE_LIST
%token ZERO_LIST
%token SNOC
%token LESS_LIST
%token EQUAL_LIST
%token XOR_LIST
%token AND_LIST
%token OR_LIST
%token RLESS
%token REQUAL
%token XOR_QCTL
%token AND_QCTL
%token OR_QCTL
%token MULT_LIST
%token LENGTH
%token REVERSE

%start modules

%% modules : module
| module modules
;

module : BASIC bigvar DEL11 varlist DEL12 transitions defines outputs
END SEMICOLON
| MODULE bigvar DEL11 varlist DEL12 components defines outputs specs
END SEMICOLON
| RECURSIVE bigvar DEL11 numlist SEMICOLON termlist DEL12
IF expr THEN components defines outputs
ELSE components defines outputs
END
dataequations specs
END SEMICOLON
| GENERIC bigvar DEL11 numlist SEMICOLON termlist DEL12
components defines outputs dataequations specs
END SEMICOLON
| INSTANCE bigvar DEL11 numlist SEMICOLON termlist DEL12
```

```
IF expr THEN components defines outputs
ELSE components defines outputs
END
dataequations specs
END SEMICOLON
| LEMMA bigvar ltlterm SEMICOLON
;

components :bigvar COLON bigvar DEL11 termlist DEL12 SEMICOLON components
|bigvar COLON bigvar DEL11 termlist DEL12 SEMICOLON
|bigvar COLON bigvar DEL11 numlist SEMICOLON termlist DEL12 SEMICOLON
components
|bigvar COLON bigvar DEL11 numlist SEMICOLON termlist DEL12 SEMICOLON
;

transitions : INIT DEL11 var DEL12 ASSIGN num SEMICOLON transitions
| INIT DEL11 var DEL12 ASSIGN initval SEMICOLON transitions
| NXXT DEL11 var DEL12 ASSIGN propterm SEMICOLON transitions
|
;

defines : DEFINE assignlist
|
;

outputs : OUTPUT assignlist
;

dataequations : DATAEQUATIONS eqterm
|
;

eqterm : var ASSIGN DEL11 var EQUIV var DEL11 varlist DEL12 DEL12
SEMICOLON eqterm
| var ASSIGN var DEL11 DEL11 var DEL11 varlist DEL12 COMMA var
DEL12 DEL12 SEMICOLON eqterm
| var ASSIGN DEL11 DEL21 varlist DEL22 EQUIV var DEL11 varlist
DEL12 DEL12 SEMICOLON eqterm
|
;

specs : SPEC ltlterm SEMICOLON specs
|
```

```
;

assignlist : var ASSIGN termlist SEMICOLON
| var ASSIGN termlist SEMICOLON assignlist
| var ASSIGN propterm SEMICOLON
| var ASSIGN propterm SEMICOLON assignlist
| var ASSIGN DEL21 varlist DEL22 SEMICOLON
| var ASSIGN DEL21 varlist DEL22 SEMICOLON assignlist
;

termlist : term
| term COMMA termlist
;

varlist : var
| var COMMA varlist
|
;

var : VAR
| NUMBER
;

bigvar : BIGVAR
;

initval : TRUE
| FALSE
;

ltlterm : IF ltlterm THEN ltlterm ELSE ltlterm
| LET var EQUIV ltlterm IN ltlterm
| ltlterm1
;

ltlterm1 : ltlterm1 ATNEXT ltlterm2
| ltlterm1 SATNEXT ltlterm2
| ltlterm2
;

ltlterm2 : ltlterm2 BEFORE ltlterm3
| ltlterm2 SBEFORE ltlterm3
| ltlterm3
```

```
;

ltlterm3 : ltlterm3 UNTIL ltlterm4
| ltlterm3 SUNTIL ltlterm4
| ltlterm4
;

ltlterm4 : ltlterm4 WHEN ltlterm5
| ltlterm4 SWHEN ltlterm5
| ltlterm5
;

ltlterm5 : ltlterm5 EQUIV ltlterm6
| ltlterm5 XOR ltlterm6
| ltlterm6
;

ltlterm6 : ltlterm6 IMPLIES ltlterm7
| ltlterm7
;

ltlterm7 : ltlterm7 OR ltlterm8
| ltlterm8
;

ltlterm8 : ltlterm8 AND ltlterm9
| ltlterm9
;

ltlterm9 : VAR
| NUMBER
| FALSE
| TRUE
| NEXT ltlterm9
| ALWAYS ltlterm9
| EVENTUAL ltlterm9
| ONEPATH ltlterm9
| ALLPATH ltlterm9
| NEG ltlterm9
| DEL11 ltlterm DEL12
| DEL21 ltlterm DEL22
;
```

```
propterm : IF propterm THEN propterm ELSE propterm
| LET var EQUIV propterm IN propterm
| propterm1
;

propterm1 : propterm1 EQUIV propterm6
| propterm1 XOR propterm6
| propterm6
;

propterm6 : propterm6 IMPLIES propterm8
| propterm7 ;

propterm7 : propterm7 OR propterm8
| propterm8
;

propterm8 : propterm8 AND propterm9
| propterm9
;

propterm9 : var
| FALSE
| TRUE
| NEG propterm9
| DEL11 propterm DEL12
| DEL21 propterm DEL22
;

term : HD DEL11 term DEL12
| TL DEL11 term DEL12
| term COLON num
| term2
;

term2 : CONS DEL11 term2 COMMA term2 DEL12
| SNOC DEL11 term2 COMMA term2 DEL12
| APPEND DEL11 term2 COMMA term2 DEL12
| FIRSTN DEL11 num COMMA term2 DEL12
| LASTN DEL11 num COMMA term2 DEL12
| BUTFIRSTN DEL11 num COMMA term2 DEL12
| BUTLASTN DEL11 num COMMA term2 DEL12
| term3
```

```
;

term3 : VAR
| BIGVAR
| NUMBER
| FALSE
| TRUE
| DEL11 term DEL12
| DEL21 term DEL22
;

expr : DEL11 num BIG num DEL12
| DEL11 num LESS num DEL12
;

numlist : num
| num COMMA numlist
|
;

num : num PLUS num
| num MINUS num
| num MULT num
| num DIV num
| DEL11 num DEL12
| NUMBER
;
%%
```