8 Simulationsablauf

Da die nebenläufigen Anweisungen i.d.R. auf einem Prozessor, d.h. nur "quasi" parallel ablaufen, ist ein spezieller Simulationsablauf nötig, der die Nebenläufigkeit von VHDL "nachbildet".

Dazu sind die Zusammenhänge zwischen nebenläufigen und sequentiellen Anweisungen und der Unterschied zwischen Signal- und Variablenzuweisungen näher zu beleuchten.

Sequentielle Anweisungen stehen in Prozessen, Funktionen oder Prozeduren. Funktionen und Prozeduren werden durch spezielle Aufrufe aktiviert, die selbst wieder sequentiell oder nebenläufig sind. Prozesse gelten als nebenläufige Anweisung. Sie werden durch eine **Liste sensitiver Signale** im Prozeß-Kopf oder durch `WAIT`-**Anweisungen** innerhalb des Prozesses gesteuert. Die Prozesse werden immer dann aktiviert und ausgeführt, wenn auf mindestens einem der Signale aus der "sensitivity list" ein Ereignis auftritt. Ist keine "sensitivity list" vorhanden, so wird ein Prozeß dadurch reaktiviert, daß die Bedingung der `WAIT`-Anweisung erfüllt wird.

8.1 Delta-Zyklus

Während der Simulation schreitet die (Simulations-)Zeit fort, wobei jeder Zeitpunkt, an dem Transaktionen eingetragen sind, bearbeitet wird. Ein Simulationszeitpunkt besteht dabei im allgemeinen aus mehreren Zyklen, die um eine infinitesimal kleine Zeit, mit "Delta" (Δ) bezeichnet, versetzt sind. Jeder dieser Delta-Zyklen besteht wiederum aus zwei aufeinanderfolgenden Phasen:

1. **Prozeß-Ausführungsphase** ("process evaluation"):
   Hierbei werden alle aktiven Prozesse bis zur END-Anweisung bzw. bis zur nächsten WAIT-Anweisung abgearbeitet. Dies beinhaltet das Ausführen aller enthaltenen Anweisungen bis auf die

Signalzuweisungen, also insbesondere auch die Zuweisung von Variablen.

2. **Signalzuweisungsphase** ("signal update"):
   Nach Ausführung des oder der für dieses "Delta" aktiven Prozesse werden die in diesen Prozessen zugewiesenen Signaländerungen durchgeführt. Dadurch können weitere Prozesse oder nebenläufige Signalzuweisungen aktiviert werden. In diesem Fall wird der Zyklus ein "Delta" später wieder von vorne durchlaufen.

Zu Beginn der Bearbeitung eines Simulationszeitpunktes t_n werden zunächst die dort vorgesehenen Signalzuweisungen durchgeführt, danach alle sensitiven Prozesse ausgeführt, um anschließend die von diesen Prozessen ausgelösten Signalzuweisungen durchzuführen, usw. Dieser Ablauf wird an einem Zeitpunkt t_n solange wiederholt, bis sich ein stabiler Zustand einstellt, d.h. bis sich keine neue Signalzuweisung mehr ergibt und kein weiterer Prozeß aktiviert wird. Man erhält ein Ablaufschema, das in folgender Art und Weise dargestellt werden kann:

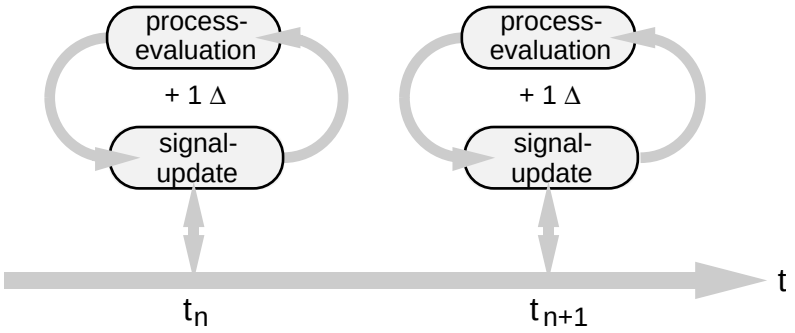

Abb. B-19: Simulationsablauf

Es ist an dieser Stelle zu beachten, daß auch die nebenläufigen Anweisungen als Prozesse interpretiert werden. Letztendlich läßt sich nämlich jede nebenläufige Anweisung durch einen äquivalenten Prozeß ersetzen. Die Signale auf der rechten Seite einer nebenläufigen Signalzuweisung werden bei dieser Umsetzung in die "sensitivity-list" aufgenommen. Für folgende nebenläufige Signalzuweisung:

```
csa: sig_b <= '1', '0' AFTER 2 ns WHEN sel = 1 ELSE
              '0', '1' AFTER 3 ns WHEN sel = 2 ELSE
              'Z';
```

lautet die äquivalente Darstellung als Prozeß:

```
csa: PROCESS (sel)
BEGIN
  IF    sel = 1 THEN sig_b <= '1', '0' AFTER 2 ns;
  ELSIF sel = 2 THEN sig_b <= '0', '1' AFTER 3 ns;
  ELSE               sig_b <= 'Z';
  END IF;
END PROCESS csa;
```

Damit wird ersichtlich, daß nebenläufige Signalzuweisungen immer durch Ereignisse auf den Signalen der rechten Seite aktiviert werden.

8.2 Zeitverhalten von Signal- und Variablenzuweisungen

Es sei noch einmal explizit erwähnt, daß Signale zwar beim Abarbeiten der entsprechenden sequentiellen Signalzuweisungen projektiert werden - was soviel heißt wie ein Vormerken des neuen Signalwertes ("scheduling") - aber erst am Ende des Prozesses oder beim Erreichen der nächsten `WAIT`-Anweisung den neuen Wert annehmen.

Besondere Beachtung erfordern deshalb Prozesse, die gemischt Variablen- und Signalzuweisungen verwenden und gegenseitige Abhängigkeiten zwischen Variablen und Signalen aufweisen. Auch bei der Prozeßkommunikation ist darauf zu achten, daß sich die triggernden Signale zum richtigen Zeitpunkt ändern.

Ein kleines Beispiel soll verdeutlichen, daß Signalzuweisungen innerhalb von Prozessen sich nicht wie herkömmliche sequentielle Anweisungen verhalten:

```
ARCHITECTURE number_one OF example IS
  SIGNAL a, b : integer := 0;
BEGIN
  a <= 1 AFTER 1 ns, 2 AFTER 2 ns, 3 AFTER 3 ns;
  PROCESS (a)           -- Aufruf bei Ereignis auf Signal a
    VARIABLE c : integer := 0;
  BEGIN
    b <= a + 2;         -- Signalzuweisung
    c := 2 * b;         -- Variablenzuweisung
  END PROCESS;
END number_one;
```

Der Prozeß wird (nach dem initialen Durchlauf) zum Zeitpunkt 1 ns + 1Δ durch ein Ereignis auf dem Signal a getriggert. Das Signal b wird erst nach Ende des Prozeßdurchlaufes in der Signalzuweisungsphase den Wert 3 erhalten. Zum Zeitpunkt der Variablenzuweisung liegt noch der alte Wert von b (2), herrührend von der Initialisierungsphase, vor. Die Variable c wird also zum Zeitpunkt 1 ns + 1Δ nicht wie erwartet den Wert 6, sondern den Wert 4 annehmen. Der gewünschte Wert 6 wird erst beim nächsten Prozeßdurchlauf erreicht; in diesem Fall zum Zeitpunkt 2 ns + 1Δ.

Dieses Problem kann gelöst werden, indem die arithmetischen Operationen nur mit Variablen durchgeführt werden, da diese den neuen Wert ohne jegliche Verzögerung annehmen:

```
ARCHITECTURE number_four OF example IS
  SIGNAL a : integer := 0;
BEGIN
  a <= 1 AFTER 1 ns, 2 AFTER 2 ns, 3 AFTER 3 ns;
  PROCESS (a)           -- Aufruf bei Ereignis auf Signal a
    VARIABLE b, c : integer := 0;
  BEGIN
    b := a + 2;         -- Variablenzuweisung
    c := 2 * b;         -- Variablenzuweisung
  END PROCESS;
END number_four;
```

Die Variable b nimmt hier sofort den Wert 3 an, so daß c wie gewünscht den Wert 6 erhält.

Alternativ zu dieser Lösung könnte neben a auch das Signal b in die "sensitivity-list" des Prozesses aufgenommen werden:

```
ARCHITECTURE number_three OF example IS
  SIGNAL a, b : integer := 0;
BEGIN
  a <= 1 AFTER 1 ns, 2 AFTER 2 ns, 3 AFTER 3 ns;
  PROCESS (a, b) -- Aufruf bei Ereignis auf Signal a oder b
    VARIABLE c : integer := 0;
  BEGIN
    b <= a + 2;         -- Signalzuweisung
    c := 2 * b;         -- Variablenzuweisung
  END PROCESS;
END number_three;
```

Der Prozeß wird bei dieser Variante zum Zeitpunkt 1 ns + 1Δ zuerst durch das Ereignis auf dem Signal a aktiviert. Anschließend aktiviert die Änderung des Signals b von 2 auf 3 erneut den Prozeß (bei 1 ns + 2Δ). Beim zweiten Prozeßdurchlauf wird der Wert der Variablen c mit dem inzwischen aktualisierten Wert von b richtig berechnet.

Eine dritte Variante zur Lösung des Problems unter Verwendung von WAIT-Anweisungen findet sich auf der beiliegenden Diskette.

Da Signale nicht im gleichen Delta-Zyklus neu zugewiesen werden müssen, auch wenn sie am Ende des Simulationszeitpunktes gleiche Werte besitzen, können Assertions, die auf Gleichheit zweier Signale prüfen, "falsche" Fehlermeldungen produzieren.

8.3 Aktivierung zum letzten Delta-Zyklus ✓93

In ✓93 wurde der Begriff "postponed" im Zusammenhang mit Prozessen, Prozeduraufrufen, Signalzuweisungen und Assertions eingeführt. Er besagt, daß die entsprechend markierten VHDL-Anweisungen erst im definitiv letzten Delta-Zyklus eines Simulationszeitpunktes aktiviert werden.

8.3.1 Prozesse

Die erweiterte Syntax der Prozeßanweisung einschließlich des optionalen Schlüsselwortes `POSTPONED` lautet nun (✓93) :

```
[process_label :] [POSTPONED] PROCESS
      [(sig_name_1 {, sig_name_n})] [IS]
  ...
BEGIN
  ...
END [POSTPONED] PROCESS [process_label] ;
```

Die Kennzeichnung eines Prozesses als `POSTPONED` hat folgende Konsequenzen:

- Da der Prozeß erst zum letzten Delta-Zyklus aktiviert wird, befinden sich alle Signale in einem für diesen Simulationszeitpunkt stabilen Zustand.
- Derartige Prozesse dürfen keine neuen Delta-Zyklen mehr verursachen. Dies bedeutet insbesondere:
  - es dürfen keine unverzögerten Signalzuweisungen enthalten sein.
  - es dürfen keine "`WAIT FOR 0 fs;`"-Anweisungen enthalten sein,
- Falls der Prozeß auf mehrere Signale sensitiv ist, kann im letzten Delta-Zyklus durch Attribute wie z.B. `EVENT` nicht mehr festgestellt werden, welches Signal den Prozeß aktiviert hat.

8.3.2 Assertions

Gerade bei Assertions ist es wichtig, daß evtl. zu überprüfende Signale einen stabilen Zustand erreicht haben. Unnötige Fehlermeldungen werden so vermieden. Bei nebenläufigen Assertions lautet die Syntax mit dem Schlüsselwort `POSTPONED` (✓93):

```
[assert_label :] [POSTPONED] ASSERT condition
                 [REPORT "message_string"]
                 [SEVERITY severity_level] ;
```

8.3.3 Signalzuweisungen

Auch bei nebenläufigen Signalzuweisungen ist eine Verlagerung in den letzten Delta-Zyklus durch das Schlüsselwort POSTPONED möglich. Signalzuweisungen ohne jegliche Verzögerung sind dabei allerdings nicht erlaubt. Die entsprechende Syntax am Beispiel der nicht bedingten Signalzuweisung lautet (✓**93**):

```
[assignment_label :] [POSTPONED] sig_name
     <= [TRANSPORT] value_1 [AFTER time_1]
                 {, value_n  AFTER time_n } ;
```

8.3.4 Prozeduraufrufe

Ein weiterer Befehl, der in den letzten Delta-Zyklus verlegt werden kann, ist der nebenläufige Aufruf einer Prozedur nach der folgenden Syntax mit dem Schlüsselwort POSTPONED (✓**93**):

```
[proc_call_label :]
                  [POSTPONED] procedure_name
                  [(argument_values)] ;
```

9 Besonderheiten bei Signalen

Neben den obigen Erläuterungen über nebenläufige und sequentielle Signalzuweisungen gibt es noch einige Besonderheiten bei der Handhabung von Signalen in VHDL, die hier erwähnt werden sollen.

9.1 Signaltreiber und Auflösungsfunktionen

Ein Signal ist ein Informationsträger, der verschiedene Signalwerte annehmen kann. Im Falle herkömmlicher Signale, wie sie in vorhergehenden Kapiteln erläutert wurden, wird dieser Wert durch Signalzuweisungen festgelegt. Dahinter verbirgt sich aber ein Mechanismus, der erst bei mehrfachen, gleichzeitigen Zuweisungen an ein und dasselbe Signal wichtig wird: Das Konzept von Signaltreibern und Auflösungsfunktionen ("resolution functions").

Soll z.B. ein Bus oder eine bidirektionale Verbindung aufgebaut werden, sind jeweils mehrere Signalquellen für ein Signal vorzusehen, die in gewisser Weise einen Teil zum resultierenden Signalwert auf dem Bus oder der bidirektionalen Leitung beitragen. Diese einzelnen Signalquellen werden in VHDL als Signaltreiber ("driver") bezeichnet.

Signaltreiber werden durch VHDL-Signalzuweisungen erzeugt. Dies geschieht oft auch unbeabsichtigt durch mehrfache Signalzuweisungen. Das Signal `w` im nachfolgenden Beispiel besitzt durch die beiden Signalzuweisungen zwei Signaltreiber, die u.U. auch gleichzeitig aktiv sein können, da die Anweisungen nebenläufig (parallel) ablaufen.

```
ARCHITECTURE arch_one OF anything IS
BEGIN
   w <= y AFTER 25 ns WHEN sel = 1 ELSE '0' ;
   w <= z AFTER 30 ns WHEN sel = 2 ELSE '0' ;
END arch_one ;
```

Bei mehreren gleichzeitig aktiven Signaltreibern ist es möglich, daß von den einzelnen Treibern unterschiedliche Signalwerte zugewiesen werden. Solche Fälle werden durch die sog. Auflösungsfunktionen geregelt. Abb. B-20 zeigt schematisch die Aufgabe dieser Funktionen:

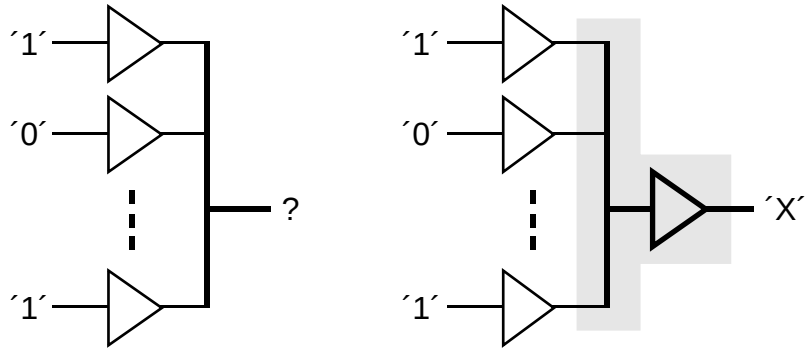

Abb. B-20: Wirkung von Auflösungsfunktionen

Bei Deklaration eines eigenen Logiktyps und Verwendung mehrfacher Treiber muß auch die entsprechende Auflösungsfunktionen beschrieben werden. Bei der Deklaration werden alle Signale und Ports gekennzeichnet, auf die eine solche Funktion angewandt werden soll; man spricht dann von aufgelösten Signalen (sog. "resolved signals"). Dies kann auf zwei Arten erfolgen:

- durch Deklaration eines Untertyps (`res_type_name`), der vom Basistyp (`unres_type_name`) durch Angabe der Auflösungsfunktion (`res_fct_name`) abgeleitet wird:

```
SUBTYPE  res_type_name
          IS res_fct_name unres_type_name ;
SIGNAL   res_sig_name_1 {, res_sig_name_n} :
          res_type_name [:= def_value] ;
```

- durch Deklaration des Signals direkt unter Angabe der Auflösungsfunktion:

```
SIGNAL   res_sig_name_1 {, res_sig_name_n} :
          res_fct_name unres_type_name
         [:= def_value] ;
```


Die Auflösungsfunktion wird bei jeder Signalzuweisung implizit aufgerufen und berechnet den neuen Signalwert aus den Beiträgen der einzelnen Treiber. Theoretisch können alle verwendeten Ports und Signale mit aufgelöstem Typ deklariert werden. Der implizite Aufruf der entsprechenden Funktion bei jeder Signalzuweisung erhöht die Simulationszeit jedoch ganz erheblich.

Auflösungsfunktionen müssen beliebig viele Treiber berücksichtigen können und einen Defaultwert zurückliefern, falls keiner der Treiber aktiv einen Signalpegel erzeugt (siehe nachfolgenden Abschnitt über kontrollierte Signale und `DISCONNECT`-Anweisung).

Am Beispiel einer 4-wertigen Logik `mvl4` mit den möglichen Signalwerten `'X'`, `'0'`, `'1'`, `'Z'` soll das typische Vorgehen bei der Deklaration von aufgelösten Signalen und Auflösungsfunktionen erläutert werden:

```
PACKAGE fourval IS
  TYPE mvl4 IS ('X', '0', '1', 'Z');
  TYPE mvl4_vector IS ARRAY (natural RANGE <>) OF mvl4;
  FUNCTION resolved (a: mvl4_vector) RETURN mvl4;
  SUBTYPE mvl4_r IS resolved mvl4;
  TYPE mvl4_r_vector IS ARRAY (natural RANGE <>) OF mvl4_r;
END fourval;
```

Nach Deklaration des Basistyps `mvl4` und des entsprechenden Vektors `mvl4_vector` wird die Funktion `resolved` bekanntgegeben. Ihre Funktionalität wird im nachfolgend aufgeführten Package Body beschrieben. Daraufhin kann ein aufgelöster Untertyp `mvl4_r` abgeleitet werden. Der aufgelöste Vektortyp `mvl4_r_vector` besteht wiederum aus aufgelösten Einzelelementen.

```
PACKAGE BODY fourval IS
  FUNCTION resolved (a: mvl4_vector) RETURN mvl4 IS
    VARIABLE result : mvl4 := 'Z';
    -- Defaultwert: 'Z': schwaechster Logikwert
  BEGIN
      FOR m in a'RANGE LOOP
        IF  a(m) = 'X' OR
           (a(m) = '1' AND result = '0') OR
           (a(m) = '0' AND result = '1') THEN RETURN 'X';
        ELSIF
           (a(m) = '0' AND result = 'Z') OR
           (a(m) = '1' AND result = 'Z') THEN
             result := a(m);
        END IF;
      END LOOP;
    RETURN result;
  END resolved;
END fourval;
```

Folgendes Beispiel zeigt die Anwendung des Typs `mvl4_r`:

```
ENTITY resolve IS
  PORT (sel : IN positive; x: OUT mvl4_r);
END resolve;

ARCHITECTURE behavioral OF resolve IS
BEGIN
  x <= '0' WHEN sel = 1 ELSE 'Z';
  x <= '1' WHEN sel = 2 ELSE 'X';
END behavioral;
```

Falls `sel` auf `2` wechselt, würde die Auflösungsfunktion `resolved` für das Signal `x` den Wert `'1'` aus den beiden konkurrierenden Werten `'1'` und `'Z'` bestimmen. Wechselt `sel` dagegen auf `1`, so ergibt sich der Wert `'X'` für das Signal `x`.

Das explizite Abschalten eines einzelnen Treibers für ein aufgelöstes Signal kann unter Kenntnis der Auflösungsfunktion durch Angabe des Defaultwertes (in diesem Fall `'Z'`) erfolgen:

```
ARCHITECTURE behavioral OF demo IS
BEGIN
   y <= '1' AFTER 5 ns,       -- aktive '1' treiben
        'Z' AFTER 40 ns ;     -- 'Z' entspricht Abschalten
END behavioral ;
```

9.2 Kontrollierte Signalzuweisungen

VHDL bietet neben den bedingten Zuweisungen eine weitere Möglichkeit, Signalzuweisungen zu steuern, die sog. kontrollierten Signalzuweisungen ("guarded signal assignment"). Dies sind nebenläufige Signalzuweisungen, die durch das Schlüsselwort `GUARDED` gekennzeichnet werden:

```
[label :] sig_name <= GUARDED sig_waveform ;
```

Die kontrollierende Bedingung, die erfüllt sein muß, damit solche Signalzuweisungen auch durchgeführt werden, nennt man "*guard_expression*". Die dadurch kontrollierten Signalzuweisungen müssen innerhalb eines Blockes stehen, der die Kontrollbedingung im Blockrahmen enthält. Die entsprechende Blocksyntax lautet:

```
block_label : BLOCK (guard_expression)
  ...
BEGIN
  ...
END BLOCK ;
```

Dieses Konstrukt wird durch den VHDL-Compiler in eine äquivalente `IF`-Anweisung umgewandelt:

```
IF guard_expression THEN
  sig_name <= sig_waveform ;
END IF ;
```

Diese Anweisung steht in einem äquivalenten Prozeß mit gleichem Label (`block_label`), der eine entsprechende `WAIT ON`-Anweisung am Ende mit allen Signalen enthält, die im vorgesehenen Signalverlauf (*sig_waveform*) oder in der Bedingung (*guard_expression*) auftreten.

Mit kontrollierten Signalzuweisungen kann z.B. auch die Funktionalität des bereits erwähnten D-Latch realisiert werden:

```
ARCHITECTURE concurrent_5 OF latch IS
BEGIN
  q_assignment : BLOCK (clk = '1')
  BEGIN
    q <= GUARDED d;
  END BLOCK;
END concurrent_5;
```

9.3 Kontrollierte Signale

Die im vorhergehenden Abschnitt besprochenen kontrollierten Signalzuweisungen arbeiten mit herkömmlichen Signalen. Werden die in solchen Anweisungen neu zugewiesenen Signale aber explizit als kontrollierte Signale ("guarded signals") deklariert, so erfolgt bei einer nicht erfüllten *guard_expression* das Abschalten des entsprechenden Signaltreibers.

Die Kennzeichnung eines Signals als kontrolliertes Signal erfolgt durch zusätzliche Angabe eines der Schlüsselwörter `REGISTER` oder `BUS` bei der Deklaration. Kontrollierte Signale müssen einen aufgelösten Typ besitzen:

```
SIGNAL sig_name_1 {, sig_name_n} :
       res_type_name REGISTER [:= def_value] ;

SIGNAL sig_name_1 {, sig_name_n} :
       res_type_name BUS [:= def_value] ;
```

Auf solche Signale angewandte kontrollierte Signalzuweisungen haben folgende Bedeutung (gezeigt anhand der äquivalenten `IF`-Anweisung):

```
IF guard_expression THEN
   sig_name <= sig_waveform ;
ELSE
   sig_name <= NULL ;
END IF ;
```


Die Zuweisung von NULL auf das Signal bedeutet nichts anderes, als daß der Treiber dieser Signalzuweisung abgeschaltet wird. Für das resultierende (aufgelöste) Signal hat dies folgende Konsequenzen:

- Wurden nicht alle Treiber abgeschaltet, so wird das resultierende Signal nur anhand der nicht abgeschalteten Treiber ermittelt,
- wurden alle Treiber abgeschaltet, so wird im Falle der Signaldeklaration als REGISTER der zuletzt vorhandene Signalwert beibehalten,
- wurden alle Treiber abgeschaltet, so wird im Falle der Signaldeklaration als BUS der in der "resolution function" angegebene Defaultwert angenommen.

Das Abschalten des Signaltreibers erfolgt unmittelbar, d.h. ohne Verzögerung, nachdem die *guard_expression* den Wert false angenommen hat, es sei denn, es wurde für das Signal im Anschluß an dessen Deklaration als kontrolliertes Signal eine explizite Verzögerungszeit (*time_expr*) durch die sog. DISCONNECT-Anweisung vereinbart:

```
DISCONNECT sig_name_1 {, sig_name_n} :
     res_type_name AFTER time_expr ;

DISCONNECT OTHERS :
     res_type_name AFTER time_expr ;

DISCONNECT ALL :
     res_type_name AFTER time_expr ;
```

Jedes kontrollierte Signal darf nur eine DISCONNECT-Anweisung erhalten. Die Schlüsselwörter OTHERS und ALL beschreiben alle noch nicht explizit mit einer Abschaltverzögerung versehenen bzw. alle Signale des aufgelösten Typs res_type_name.

Die Handhabung von kontrollierten Signalen soll anhand eines Beispiels verdeutlicht werden:

```
USE work.fourval.ALL;
ENTITY mux_2_1 IS
  PORT (in_signals : IN  mvl4_vector (1 TO 2);
        choice     : IN  integer;
        out_signal : OUT mvl4_r BUS );
END mux_2_1;
```

```
ARCHITECTURE with_guards OF mux_2_1 IS
  DISCONNECT out_signal : mvl4_r AFTER 25 ns;
BEGIN

  choice_1 : BLOCK (choice = 1)
  BEGIN
    out_signal <= GUARDED in_signals(1) AFTER 20 ns;
  END BLOCK;

  choice_2 : BLOCK (choice = 2)
  BEGIN
    out_signal <= GUARDED in_signals(2) AFTER 18 ns;
  END BLOCK;

END with_guards;
```

Wechselt das Eingangssignal `choice` auf `1`, so wird der Signaltreiber des ersten Blockes aktiv und gibt das erste Eingangssignal nach 20 ns an den Ausgang weiter. Wechselt `choice` auf `2`, so wird durch den zweiten Block das zweite Eingangssignal nach 18 ns ebenfalls auf den Ausgang gelegt. Da der erste Signaltreiber aber erst nach 25 ns abschaltet, sind für 7 ns zwei Treiber aktiv. Das resultierende Signal wird durch die Auflösungsfunktion ermittelt. Wechselt `choice` auf einen Wert ungleich `1` oder `2`, so schaltet der zuletzt aktive Treiber nach 25 ns ab und das Ausgangssignal wechselt in den Defaultwert (in diesem Falle `'Z'`), da es als `BUS` deklariert ist. Bei einer Deklaration als `REGISTER` würde der letzte Signalwert erhalten bleiben.

10 Gültigkeit und Sichtbarkeit

Um bei großen Entwürfen die Übersicht über die Vielzahl von auftretenden Objekten, Typen, Unterprogrammen usw. nicht zu verlieren, ist es sinnvoll, eine eindeutige Struktur der Namensgebung einzuführen und konsequent beizubehalten. Dafür soll ein Blick auf die Regeln geworfen werden, nach denen entschieden wird, ob ein Objekt in einer Anweisung verwendet werden kann und welches VHDL-Element effektiv zum Einsatz kommt.

10.1 Gültigkeit

Der Gültigkeitsbereich eines Objektes oder eines VHDL-Elementes hängt im wesentlichen vom Ort der Deklaration ab und umschließt alle hierarchisch tieferliegenden Design- und Syntax-Einheiten nach folgender Schichtung:

- Deklarationen im Package gelten für alle Design-Einheiten, die das Package verwenden.
- Deklaration im Deklarationsteil einer Entity gelten für alle dieser Entity zugehörenden Architekturen und die darin enthaltenen Blöcke und Anweisungen.
- Deklaration im Deklarationsteil einer Architektur haben für alle enthaltenen Blöcke und Anweisungen Gültigkeit.
- Erfolgt die Deklaration in einem Block, so umfaßt der Gültigkeitsbereich alle im Block enthaltenen Anweisungen.
- Deklaration innerhalb eines Prozesses gelten nur für die im Prozeß enthaltenen Anweisungen.
- Deklarationen in einer Schleife (Laufvariable) oder im Deklarationsteil einer Funktion bzw. einer Prozedur haben nur die eingeschränkte Gültigkeit für diese spezielle Anweisung.

10.2 Sichtbarkeit

Entscheidend dafür, ob ein VHDL-Element in einer Anweisung verwendet werden kann ist, ob es am Ort des Auftretens sichtbar ist. Sichtbarkeit bedeutet dabei entweder direkte Sichtbarkeit oder Sichtbarkeit durch Auswahl.

10.2.1 Direkte Sichtbarkeit

Der Bereich, in dem VHDL-Elemente direkt sichtbar sind, umfaßt in der Regel ihren Gültigkeitsbereich nach der Deklaration, es sei denn, das VHDL-Element ist verborgen.

Verborgenheit liegt vor, falls an einer Stelle mehrere gleichartige, gültige VHDL-Elemente existieren, die an hierarchisch verschiedenen Orten deklariert wurden. In diesem Fall maskiert oder verbirgt das lokal (hierarchisch niedriger) deklarierte VHDL-Element die anderen.

Sind gleichzeitig mehrere VHDL-Elemente mit gleichem Namen direkt sichtbar, werden die Regeln der Überladung (siehe Kapitel 11) angewandt. Treffen die Regeln der Überladung nicht zu, kann die Anweisung nicht ausgeführt werden.

10.2.2 Sichtbarkeit durch Auswahl

Nicht direkt sichtbare VHDL-Elemente können durch die evtl. hierarchische, vorangestellte Angabe des Deklarationsortes (nach den Regeln der sog. "selected names") angesprochen werden. Das Zeichen zur Trennung zwischen Prefix und eigentlichem Namen ist der Punkt.

"Selected names" werden nach folgender Syntax, z.B. für Objekte eines Packages, für Design-Einheiten aus einer Bibliothek oder für Record-Elemente verwendet:

```
lib_name.pack_name.obj_name_in_pack
lib_name.design_unit_name
record_name.record_element_name
```

Um bestimmte Elemente / Funktionen aus Packages oder Design-Einheiten aus Bibliotheken lokal ohne vollständigen Deklarationspfad an-

sprechen zu können, kann die USE-Anweisung auch innerhalb der Deklarationsanweisungen von Architekturen, Blöcken, Prozessen, usw. stehen.

Weitere Hinweise zur Sichtbarkeit von Bibliotheken und Packages finden sich im Abschnitt zu den LIBRARY- und USE-Anweisungen.

```
PACKAGE p IS
  CONSTANT c : integer := 1;
END p;
```

```
ENTITY e IS
  CONSTANT c : integer := 2;      -- in ARCH. zu e bekannt
END e;
```

```
ARCHITECTURE a OF e IS
  SIGNAL s1,s2,s3,s4,s5,s6,s7 : integer := 0;
BEGIN
  -- PACKAGE p hier nicht bekannt: sel.name work.p.c noetig
  s6 <= work.p.c ;       -- benutzt c aus PACKAGE p :   1
  s7 <= c;               -- benutzt c aus ENTITY  e :   2
  b : BLOCK
    CONSTANT c : integer := 3;
  BEGIN
    s3 <= c;             -- benutzt c aus BLOCK   b :  3
    x : PROCESS
      CONSTANT c : integer := 4;
      USE work.p;        -- mache PACKAGE p lokal sichtbar
    BEGIN
      s4 <= c;           -- benutzt c aus PROCESS x :   4
      l : FOR c IN 5 TO 5 LOOP
        s1 <= p.c;       -- benutzt c aus PACKAGE p :   1
        s2 <= e.c;       -- benutzt c aus ENTITY  e :   2
        s5 <= c;         -- benutzt c aus LOOP    l :   5
      END LOOP;
      WAIT;
    END PROCESS;
  END BLOCK;
END a;
```

11 Spezielle Modellierungstechniken

In diesem Kapitel sollen VHDL-Konstrukte und Modellierungstechniken erläutert werden, die bei einfachen Aufgabenstellungen in der Regel nicht benötigt werden. Es richtet sich deshalb vorwiegend an den fortgeschrittenen VHDL-Anwender.

11.1 Benutzerdefinierte Attribute

Neben den vordefinierten Attributen können auch vom Benutzer Attribute vergeben werden. Diese können allerdings im Gegensatz zu den vordefinierten Attributen nur konstante Werte besitzen.

Benutzerdefinierte Attribute können nicht nur für Signale, Variablen und Konstanten vergeben werden, sondern auch für die Design-Einheiten (Entity, Architecture, Configuration, Package) und weitere VHDL-Elemente wie Prozeduren, Funktionen, Typen, Untertypen, Komponenten und sogar für Labels.

Mit Hilfe von Attributen können diesen verschiedenen Elementen zusätzliche Informationen mitgegeben werden, die sich mit den bisher eingeführten VHDL-Konstrukten nicht abbilden lassen. Beispielsweise können einer Architektur Angaben über die maximalen Gehäuseabmessungen, den Bauteillieferanten oder die erwarteten Herstellungskosten zugeordnet werden.

Ein bedeutendes Anwendungsgebiet für benutzerdefinierte Attribute ist die VHDL-Synthese. Hier lassen sich mit programmspezifischen Attributen Vorgaben zur Zustandscodierung oder Pfadlaufzeit festlegen. Während der VHDL-Simulator diese Attribute nicht weiter bearbeitet, werden sie vom Synthesewerkzeug erkannt und entsprechend umgesetzt.

Gegenüber Kommentaren bietet die Verwendung von Attributen den Vorteil der strengen Typüberwachung und die Möglichkeit, den Attributwert mit Hilfe des Attributnamens (wie bei den vordefinierten Attributen) in den VHDL-Modellen abzufragen.

Bevor das Attribut einem VHDL-Element zugewiesen und mit einem Wert versehen werden kann, muß zunächst eine Attributdeklaration erfolgen.

Deklaration von Attributen

Attributdeklarationen haben folgendes Aussehen:

```
ATTRIBUTE attribute_name : type_name ;
```

Der Typ des Attributs (`type_name`) kann ein vordefinierter oder ein eigendefinierter Typ sein.

Definition von Attributen

Die Verknüpfung des Attributs mit einem oder mehreren Elementen unter gleichzeitiger Wertzuweisung geschieht durch Angabe des entsprechenden VHDL-Elements (`element_type`: Konstante, Variable, Signal, Entity, ... Label) mit Hilfe folgender Anweisung:

```
ATTRIBUTE attribute_name OF  element_name_1
                         {,  element_name_n}
     : element_type IS attribute_value ;
```

Anstelle der Elementnamen (oder Labels von bestimmten Anweisungen) sind auch die Schlüsselwörter `OTHERS` und `ALL` möglich.

Vordefinierte Attribute können auf diese Weise nicht mit neuen Werten belegt werden; sie werden ausschließlich vom Simulator belegt und können nur abgefragt werden.

Anwendung von Attributen

Die benutzerdefinierten Attribute können in der gleichen Art und Weise wie die vordefinierten Attribute abgefragt werden:

```
element_name'attribute_name
```

```
ENTITY attr_demo IS
  -- Typdeklarationen -----------------------------------
  TYPE level IS (empty, medium, full);
  TYPE state IS (instabil, stabil);
  TYPE res_type IS RANGE 0.0 TO 1.0E6;
  TYPE loc_type IS RECORD x_pos : integer;
                  y_pos : integer;
                  END RECORD;
  -- Attributdeklarationen ------------------------------
  ATTRIBUTE technology    : string;
  ATTRIBUTE priority      : level;
  ATTRIBUTE sig_state     : state;
  ATTRIBUTE resistance    : res_type;
  ATTRIBUTE location      : loc_type;
  -- Attributdefinition ---------------------------------
  ATTRIBUTE technology OF attr_demo : ENTITY IS "ttl";
END attr_demo;
```

```
ARCHITECTURE demo OF attr_demo IS
  SIGNAL in_a, in_b, in_c : bit := '1';
  -- Attributdefinition ---------------------------------
  ATTRIBUTE sig_state  OF in_a       : SIGNAL IS instabil;
  ATTRIBUTE sig_state  OF OTHERS     : SIGNAL IS stabil;
BEGIN
  -- Attributanwendung ----------------------------------
  ASSERT attr_demo'technology = "cmos"
         REPORT "Kein CMOS-Modul" SEVERITY note;
END demo;
```

Die neue Norm (✓**93**) erlaubt, daß jede Anweisung - auch sequentielle - ein Label erhalten kann. Somit kann nahezu jede Zeile aus dem VHDL-Quelltext mit einem Attribut versehen werden.

Ebenso können in ✓**93** Gruppen mit Attributen belegt werden.

11.2 Gruppen ✓93

Mit der Überarbeitung der Norm wurde auch der Begriff der Gruppe in die Sprache eingeführt. Mehrere Objekte, Design-Einheiten und Unterprogramme können in einer Gruppe zusammengefaßt und gemeinsam mit Attributen dekoriert werden. Gruppenelemente können alle mit Namen versehene VHDL-Elemente sein (siehe auch Aufzählung bei den benutzerdefinierten Attributen; in ✓93 kommen dazu noch `LITERAL`, `UNITS`, `GROUP` und `FILE`).

Da auch Labels in eine Gruppe aufgenommen werden können, lassen sich z.B. Prozesse oder nebenläufige Signalzuweisungen, die mit einem Label versehen sind, in Gruppen zusammenfassen.

Typdeklaration von Gruppen

Bevor man jedoch konkrete Gruppen bildet, muß ähnlich wie bei herkömmlichen Objekten, in einer Deklaration der Gruppentyp festgelegt werden:

```
GROUP group_type_name IS
            (    element_1_type  [<>]
             { , element_n_type  [<>] } );
```

Die optionale Angabe der Zeichen `<>` bedeutet, daß beliebig viele Elemente des genannten Typs auftreten können. Beispiele:

```
GROUP path IS (SIGNAL, SIGNAL);  -- Pfad: 2 Signale
GROUP pins IS (SIGNAL <>);       -- Pins: beliebig
                                 -- viele Signale (<>)
```

Deklaration von Gruppen

Nachdem der Gruppentyp bekanntgegeben wurde, können in der Gruppendeklaration konkrete VHDL-Einheiten zu einer Gruppe zusammenfügt werden.

Die Syntax für die Deklaration einer Gruppe lautet:

```
GROUP   group_name : group_type_name
    ( group_element_1 { , group_element_n } );
```

Anzahl und Typen der Gruppenelemente müssen dabei der Typdeklaration entsprechen.

Anwendung von Gruppen

Ein Anwendungsgebiet für Gruppen kann z.B. die Angabe von Verzögerungszeiten einer Gruppe `path`, die aus zwei Signalen besteht, mit Hilfe von benutzerdefinierten Attributen sein:

```
ENTITY dlatch_93 IS                -- !!! VHDL'93-Syntax !!!
  PORT (d, clk  : IN  bit;
        q, qbar : OUT bit);

  GROUP path IS (SIGNAL, SIGNAL);
  GROUP d_to_q      : path (d, q);
  GROUP d_to_qbar   : path (d, qbar);
  GROUP clk_to_q    : path (clk, q);
  GROUP clk_to_qbar : path (clk, qbar);
  ATTRIBUTE propagation : time;
  ATTRIBUTE propagation OF d_to_q      : GROUP IS 3.8 ns;
  ATTRIBUTE propagation OF d_to_qbar   : GROUP IS 4.2 ns;
  ATTRIBUTE propagation OF clk_to_q    : GROUP IS 2.8 ns;
  ATTRIBUTE propagation OF clk_to_qbar : GROUP IS 2.9 ns;
END dlatch_93;
```

```
ARCHITECTURE with_path_attributes OF dlatch_93 IS
BEGIN                              -- !!! VHDL'93-Syntax !!!
  PROCESS (d, clk)
  BEGIN
    IF clk'EVENT AND clk = '1' THEN
      q    <= d AFTER clk_to_q'propagation;
      qbar <= d AFTER clk_to_qbar'propagation;
    ELSIF d'EVENT THEN
      q    <= d AFTER d_to_q'propagation;
      qbar <= d AFTER d_to_qbar'propagation;
    END IF;
  END PROCESS;
END with_path_attributes;
```

11.3 Überladung

Ein Punkt, der erneut die Verwandtschaft von VHDL mit höheren Programmiersprachen zeigt, ist die Möglichkeit zur Überladung ("overloading") von:

- Unterprogrammen (Funktionen und Prozeduren),
- Operatoren und
- Werten von Aufzähltypen.

Unter Überladung versteht man die gleichzeitige Sichtbarkeit von mehreren gleichnamigen Unterprogrammen, Operatoren bzw. von Objektwerten die zu unterschiedlichen Aufzähltypen gehören können. Mit Hilfe der Überladung gelingt es beispielsweise, den Anwendungsbereich einer Funktion zu vergrößern.

Die verschiedenen Varianten eines Unterprogramms oder eines Operators unterscheiden sich nur in Anzahl und Typ ihrer Argumente und Ergebnisse. VHDL-Programme erkennen aus dem Kontext heraus (d.h. aus der Argumentzahl und deren Typen), welche der sichtbaren Varianten anzuwenden ist. Ist diese Entscheidung nicht eindeutig, d.h. sollten sich mehrere sichtbare Alternativen als "passend" für die zu erfüllende Aufgabe erweisen, erfolgt die Ausgabe einer Fehlermeldung.

Durch das Konzept der Überladung werden VHDL-Modelle übersichtlicher, da nicht für jede Variante einer bestimmten Funktionalität ein neuer Bezeichner vergeben werden muß.

11.3.1 Überladen von Unterprogrammen

Verschiedene Unterprogramme mit gleichem Namen, aber unterschiedlichem Verhalten, werden wie gewöhnlich deklariert. Dies gilt gleichermaßen für Prozeduren und Funktionen. Ein Beispiel:

```
ENTITY overload IS
----- Bestimmung des Maximums von zwei integer-Werten ------
  FUNCTION max (a_i, b_i : integer) RETURN integer IS
  BEGIN
    IF a_i >= b_i THEN RETURN a_i;
    ELSE              RETURN b_i;
    END IF;
  END max;
----- Bestimmung des Maximums von drei integer-Werten ------
  FUNCTION max (a_i, b_i, c_i : integer) RETURN integer IS
  BEGIN
    IF    a_i >= b_i AND a_i >= c_i THEN RETURN a_i;
    ELSIF b_i >= a_i AND b_i >= c_i THEN RETURN b_i;
    ELSE                                 RETURN c_i;
    END IF;
  END max;
----- Bestimmung des Maximums von zwei real-Werten ---------
  FUNCTION max (a_r, b_r : real) RETURN real IS
  BEGIN
    IF a_r >= b_r THEN RETURN a_r;
    ELSE               RETURN b_r;
    END IF;
  END max;
----- Bestimmung des Maximums von drei real-Werten ---------
  FUNCTION max (a_r, b_r, c_r : real) RETURN real IS
  BEGIN
    IF    a_r >= b_r AND a_r >= c_r THEN RETURN a_r;
    ELSIF b_r >= a_r AND b_r >= c_r THEN RETURN b_r;
    ELSE                                 RETURN c_r;
    END IF;
  END max;
END overload;
```

Die vier Funktionen `max` seien in einer Architektur gleichzeitig sichtbar. Für jeden Funktionsaufruf von `max` wählt das VHDL-Programm die Variante aus, die in folgenden Punkten dem Aufruf entspricht:

① Zahl der Argumente,

② Typen der Argumente,

③ Namen der Argumente (bei "named association"),

④ Typ des Rückgabewertes.

Folgendes Beispiel zeigt die Anwendung der oben beschriebenen, mehrfach überladenen Funktion `max`:

```
ARCHITECTURE behavioral OF overload IS
BEGIN
  PROCESS
    VARIABLE a, d : real := 0.0;
    VARIABLE b, c : integer := 0;
  BEGIN
    a := max(3.2, 2.1);   -- 3. Variante von max, a = 3.2
    b := max(1, 2, 3);    -- 2. Variante von max, b = 3
    c := max(0, 9);       -- 1. Variante von max, c = 9
    d := max(real(6), 4.3, 2.1); -- 4. Var. von max, d = 6.0
    WAIT;
  END PROCESS;
END behavioral;
```

Besitzen bei einem Funktionsaufruf mehrere Varianten Gültigkeit, so verbirgt die lokal deklarierte Variante (z.B. im Deklarationsteil der Architektur) die hierarchisch höher deklarierte Variante (z.B. aus einem Package). Mit vollständiger hierarchischer Angabe des Unterprogrammnamens (durch "selected names") kann trotzdem auf die gewünschte Variante zugegriffen werden.

11.3.2 Überladen von Operatoren

Operatoren unterscheiden sich von einfachen Funktionen durch zwei Punkte:

- Der Name bzw. das Symbol von Operatoren gilt nicht als "normaler" Bezeichner, sondern ist eine Zeichenkette (String) und steht deshalb bei der Deklaration in Anführungsstrichen.
- Die Operanden stehen beim üblichen Aufruf eines Operators vor bzw. nach dem Operator selbst (bei unären Operatoren nur danach) und nicht in nachgestellten runden Klammern. Operatoren können aber auch in der Syntax normaler Funktionsaufrufe angesprochen werden:

  `"c <= a AND b;"` entspricht `"c <= "AND" (a,b);"`

Die Handhabung von Operatoren als String ist bei der Deklaration von überladenen Operatoren zu beachten. Ansonsten entspricht diese dem Überladen von Funktionen und Prozeduren. Folgendes Beispiel zeigt die Überladung des "="-Operators. Bei Verwendung des Packages `fourval` sind damit drei Varianten des "="-Operators verfügbar: Die Variante `[0]` stellt den vordefinierten Operator "=" dar, welcher ein Ergebnis vom Typ `boolean` zurückliefert. Die beiden benutzerdefinierten Varianten `[1]`, `[2]` dagegen vergleichen zwei Werte vom Typ `mvl4` und errechnen ein Ergebnis vom Typ `bit` bzw. `mvl4`.

```
PACKAGE fourval IS
   TYPE mvl4 IS ('X', '0', '1', 'Z');
   -- Variante [0]: vordefinierter Operator "="
   -- FUNCTION "=" (a,b : mvl4) RETURN boolean;
   -- Variante [1]: eigene Definition des Operators "="
   FUNCTION "=" (a,b : mvl4) RETURN bit;
   -- Variante [2]: eigene Definition des Operators "="
   FUNCTION "=" (a,b : mvl4) RETURN mvl4;
END fourval;
```

```
PACKAGE BODY fourval IS
  FUNCTION "=" (a, b : mvl4) RETURN bit IS          -- [1]
    VARIABLE result : bit := '0';
  BEGIN
    IF (a = '1' AND b = '1') OR (a = '0' AND b = '0') OR
       (a = 'X' AND b = 'X') OR (a = 'Z' AND b = 'Z')
       THEN result := '1';
    END IF;
    RETURN result;
  END "=";

  FUNCTION "=" (a, b : mvl4) RETURN mvl4 IS         -- [2]
    VARIABLE result : mvl4 := '0';
  BEGIN
    IF (a = '1' AND b = '1') OR (a = '0' AND b = '0') OR
       (a = 'X' AND b = 'X') OR (a = 'Z' AND b = 'Z')
       THEN result := '1';
    END IF;
    RETURN result;
  END "=";
END fourval;
```

11.3.3 Überladen von Aufzähltypwerten

Neben Unterprogrammen und Operatoren können auch die einzelnen Werte eines Aufzähltyps überladen werden. Man denke z.B. an den Signalwert `'0'` des Logiktyps `bit` und die `'0'` des benutzerdefinierten Logiktyps `mvl4`. Tritt ein Ausdruck mit solchen Werten im VHDL-Quelltext auf, so muß aus dem Kontext heraus klar sein, um welchen Typ es sich dabei handelt. Kann dies nicht eindeutig festgestellt werden, so ist eine explizite Typkennzeichnung erforderlich.

11.3.4 Explizite Typkennzeichnung

Bei mehrdeutigen Ausdrücken (s.o.) und Typen von Operanden ist die explizite Kennzeichnung des gewünschten Typs durch sog. qualifizierte Ausdrücke ("qualified expressions") erforderlich. Unter Angabe des Typ- bzw. Untertypnamens haben sie folgende Syntax:

`type_name'(`*`expression`*`)`

Eine mögliche Konstellation für die Notwendigkeit eines solchen Konstruktes zeigt folgendes Beispiel. Hier wird durch die Typkennzeichnung explizit angegeben, welche Variante des mehrfach überladenen Vergleichsoperators "=" zu verwenden ist.

```
USE work.fourval.ALL;  -- siehe oben
ENTITY equal IS
  PORT (a,b,c,d : IN  mvl4;
        w :       OUT mvl4; x,y,z :   OUT boolean );
END equal;

ARCHITECTURE behavioral OF equal IS
BEGIN                                -- verw. Funktion in [ ]
  w <= (a = b) = (c = d);            -- (a [2] b) [2] (c [2] d)
  x <= (a = b) = mvl4'(c = d);       -- (a [2] b) [0] (c [2] d)
  y <= (a = b) = bit'(c = d);        -- (a [1] b) [0] (c [1] d)
  z <= (a = b) = boolean'(c = d); -- (a [0] b) [0] (c [0] d)
END behavioral;
```

11.4 PORT MAP bei strukturalen Modellen

Mit Hilfe von PORT MAP-Anweisungen können Signale auf unterschiedlichen Ebenen auf verschiedene Arten miteinander verbunden werden.

PORT MAP-Anweisungen können an verschiedenen Stellen auftreten:

- In der Design-Einheit "Configuration" verbinden sie "formals" (Ports der instantiierten Entity) mit "locals" (Ports der Komponente).
- In der Anweisung zur Komponenteninstantiierung verbinden sie "locals" (Ports der Komponente) mit "actuals" (Signale in der Architektur).

Für den erstgenannten Fall sollen einige Aspekte der PORT MAP aufgezeigt werden, die auch für Komponenteninstantiierungen gelten:

- Im einfachsten Fall werden durch eine mit Kommata getrennte Liste von "local"-Portnamen beide Signalebenen (in der gleichen Reihenfolge wie in der Komponentendeklaration angegeben) miteinander verbunden (sog. "positional association").
- Explizites Zuweisen von Signalen mit dem Zuweisungszeichen "=>" eröffnet weitaus mehr Möglichkeiten (sog. "named association"):
  - Signale können in unterschiedlicher Reihenfolge miteinander verknüpft werden,
  - "formal"-Ports können unbeschaltet bleiben (Schlüsselwort OPEN). In diesem Fall muß ein "formal"-Port mit dem Modus IN einen Defaultwert besitzen,
  - ein "local"-Port kann mit mehr als einem "formal"-Port verknüpft werden.

In Abb. B-21 sind die erlaubten Konstellationen dargestellt.

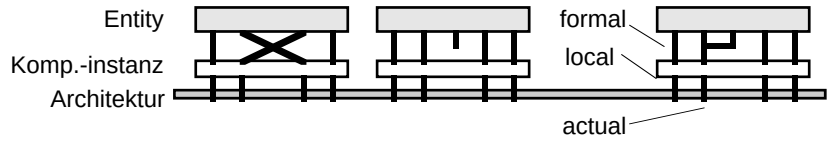

Abb. B-21: Erlaubte PORT MAP-Konstrukte in der Konfiguration

Nicht erlaubt hingegen sind unbeschaltete "locals" und "formals", die mit mehr als einem "local" verknüpft werden.

Folgende Alternative des Komplexgatters aus Kapitel 5 zeigt die Anwendung des Schlüsselwortes OPEN:

```
CONFIGURATION aoi_config_4a OF aoi IS
  FOR structural_4
    FOR nor_stage
      FOR or_c  : or2  USE ENTITY work.or2 (behavioral);
      END FOR;
    END FOR;
    FOR and_stage
-- hier:Verwendung eines AND3-Gatters in einem AND2-Sockel
-- durch Angabe von OPEN als ACTUAL des dritten Eingangs-
-- ports, work.and3 muss dort einen Defaultwert besitzen!
      FOR and_b : and2 USE ENTITY work.and3 (behavioral)
        PORT MAP (a => OPEN, b => a, c => b, y => y);
      END FOR;
      FOR and_a : and3 USE ENTITY work.and3 (behavioral);
      END FOR;
    END FOR;
  END FOR;
END aoi_config_4a;
```

11.5 File - I/O

Files (Dateien) sind serielle Anordnungen von Daten eines bestimmten Typs, an deren Ende ein "end of file" (EOF)-Zeichen steht. Files werden nicht wie herkömmliche Objekte gelesen und zugewiesen, sondern können nur mit Hilfe spezieller Funktionen gehandhabt werden. Man unterscheidet bei VHDL zwischen Textfiles und typspezifischen Files.

Bei ersteren handelt es sich um beliebig strukturierte ASCII-Dateien, die mit Hilfe der Prozeduren aus dem Package `textio` gelesen und geschrieben werden können. Die typspezifischen Files dagegen können jeweils nur Daten eines Typs enthalten. Sie werden vom jeweiligen VHDL-Simulator mit den implizit vorhandenen Prozeduren `read` und `write` gelesen bzw. geschrieben. Dieser Dateityp ist nicht menschenlesbar und kann nicht mit ASCII-Editoren erzeugt werden.

Typdeklaration von Files

Wie bei allen VHDL-Objekten ist zunächt eine Typdeklaration erforderlich, die den Namen des Filetyps mit dem Typ der Elemente verbindet:

```
TYPE file_type_name IS FILE OF element_type ;
```

Neben den Basistypen (`bit`, `integer`, ...) können auch benutzerdefinierte Typen, Arrays und Records in einem File enthalten sein.

Deklaration von Files

Daraufhin kann nun ein konkretes Fileobjekt von diesem Typ deklariert werden. Dazu muß der Modus (`IN`: Leseoperation möglich, `OUT`: Schreiboperation möglich) und der Filename angegeben werden:

```
FILE file_name : file_type_name IS IN
          "physical_file_name" ;

FILE file_name : file_type_name IS OUT
          "physical_file_name" ;
```

Durch den String `"physical_file_name"` wird der Dateiname im Filesystem angegeben, während `file_name` den Bezeichner des Objektes darstellt. Die Anwendung von Files in VHDL ist allerdings nicht sehr flexibel, da Files nur gelesen oder geschrieben werden können (es ist kein `INOUT`-Modus möglich) und auch die Reihenfolge streng sequentiell ist.

11.5.1 Typspezifische Files

In diesem Fall werden Daten ein und desselben Typs in einem File abgespeichert oder gelesen, so z.B. `integer`, `bit`, `bit_vector`. Es stehen dafür folgende Leseprozeduren zur Verfügung:

```
read (file_name, object_name) ;

read (file_name, object_name, length) ;
```

Beide Prozeduren lesen Daten aus dem File `file_name` in die Variable `object_name`. Handelt es sich beim einzulesenden Objekt um einen unbeschränkten Vektortyp (z.B.: `bit_vector`), so stellt die zweite Variante die Länge (`length`) des tatsächlich gelesenen Vektors zur Verfügung.

Das Gegenstück zu `read` ist die Prozedur `write`, die den Inhalt von `object_name` auf das File `file_name` schreibt:

```
write (file_name, object_name) ;
```

Die folgende Funktion liefert eine Boolesche Variable, die entweder den Wert `true` für erreichtes Dateiende annimmt oder `false` ist, falls noch weitere Daten vorhanden sind.

```
endfile (file_name)
```

Folgendes Beispiel zeigt ein VHDL-Modell, das den Verlauf des Eingangssignals `data_8` bis zum Zeitpunkt 100 ns auf die Datei `data_8.out` schreibt. Diese Datei könnte anschließend in einem anderen Modell mit der Prozedur `read` wieder eingelesen werden.

```
PACKAGE memory_types IS
  SUBTYPE byte IS bit_vector(0 TO 7);
END memory_types;
```

```
USE work.memory_types.ALL;
ENTITY write_data IS
  PORT (data_8 : IN byte);
END write_data;
```

```
ARCHITECTURE behavioral OF write_data IS
BEGIN
  write_process: PROCESS
    TYPE byte_file IS FILE OF byte;
    FILE output : byte_file IS OUT "data_8.out";
  BEGIN
    WAIT on data_8;
    IF now <= 100 ns THEN
      write (output, data_8);
    ELSE
      WAIT;
    END IF;
  END PROCESS;
END behavioral;
```

11.5.2 Textfiles

Eine Möglichkeit, innerhalb von Files auch unterschiedliche Typen abzuspeichern, bietet das Package `textio` aus der Bibliothek `std`. Es muß vor der Verwendung seiner Routinen erst mit folgender Anweisung bekanntgemacht werden:

```
USE std.textio.ALL ;
```

Das Package deklariert den Typ `line` und einen entsprechenden Filetyp `text`. Files dieses Typs können mit folgenden Anweisungen zeilenweise gelesen und geschrieben werden:

```
readline  (file_name, line_object_name) ;
writeline (file_name, line_object_name) ;
```

Mit Hilfe der überladenen Prozeduren aus dem Package `textio`:

```
read (line_object_name, object_name) ;
write (line_object_name, object_name) ;
```

können verschiedene Objekttypen innerhalb der Zeilen gelesen und geschrieben werden.

Die möglichen Typen sind:

- `bit`
- `bit_vector`
- `boolean`
- `time`
- `integer`
- `real`
- `character`
- `string`

Für die Leseprozeduren existieren auch Varianten mit Booleschem Prüfwert zur Kontrolle auf erfolgreiche Datei-Operation. Für die Schreibprozeduren existieren Varianten zum Ausrichten der Objekte innerhalb eines gegebenen Bereiches. Weiterhin gibt es die Funktion:

```
endline (line_object_name)
```

zur Überprüfung auf erreichtes Zeilenende (vgl. `endfile`).

Das prinzipielle Vorgehen beim Arbeiten mit Text-Files soll an einem Beispiel verdeutlicht werden. Es stellt eine flexible Testbench für ein beliebiges Modell mit drei Ein- und einem Ausgangsport dar. Die Zeilen im File `"stimres"` sind folgendermaßen aufgebaut:

Signalname (Typ `character`) - Leerzeichen - Signalwert (Typ `bit`) - Leerzeichen - Zeitangabe der Zuweisung (Typ `time`):

```
a 0 1 ns
b 0 2 ns
r 1 4 ns
  ...
  ...
```

Neben den Stimuli (Signale `a` , `b`, `c`) wird gleichzeitig die erwartete Antwort (Signal `r`) definiert und zu den entsprechenden Zeitpunkten mit dem Ausgangssignal `y` verglichen. Bei nicht übereinstimmenden Werten wird eine Fehlermeldung in das File `"errors"` geschrieben:

```
USE std.textio.ALL;

ENTITY tb_3pin IS
END tb_3pin;
```

```
ARCHITECTURE arch_1 OF tb_3pin IS
  SIGNAL a,b,c,r,y: bit;
  COMPONENT mut_socket
    PORT (a,b,c: IN  bit; y: OUT bit);
  END COMPONENT;
  FILE stimres : text IS IN  "stimres";
  FILE errors  : text IS OUT "errors";
BEGIN
  model_under_test: mut_socket PORT MAP (a,b,c,y);
  read_stimuli: PROCESS --------- Lese Stimuli ----------
    VARIABLE lin          : line; VARIABLE boo: boolean;
    VARIABLE t            : time;
    VARIABLE str_var,space: character; VARIABLE data: bit;
  BEGIN
    WHILE (endfile(stimres) = false) LOOP
      readline (stimres, lin);
      read (lin,str_var,boo);
      ASSERT boo REPORT "Error reading file!";
      IF ((str_var = 'a') OR (str_var = 'b')
       OR (str_var = 'c') OR (str_var = 'r')) THEN
        read (lin,space); read (lin,data);
        read (lin,space); read (lin,t);
        CASE str_var IS
          WHEN 'a'    => a <= TRANSPORT data AFTER t;
          WHEN 'b'    => b <= TRANSPORT data AFTER t;
          WHEN 'c'    => c <= TRANSPORT data AFTER t;
          WHEN 'r'    => r <= TRANSPORT data AFTER t;
          WHEN OTHERS => NULL;
        END CASE;
      END IF;
    END LOOP;
    WAIT;
  END PROCESS;
  write_errors : PROCESS ----- Schreibe Fehlermeldung ------
    VARIABLE lin : line;
  BEGIN
    WAIT ON r'TRANSACTION;
    IF (y /= r) THEN
      write (lin, string'("y isn't "));  write (lin, r);
      write (lin, string'(" at time ")); write (lin, now);
      writeline (errors,lin);
    END IF;
  END PROCESS;
END arch_1;
```


11.5.3 Handhabung von Files in ✓93

Die ursprüngliche Definition der Syntax zur Handhabung von Files (✓87) enthält einige unklare Punkte und schränkt den Datenim- und -export von und zu Files erheblich ein.

Die wesentlichen Neuerungen in ✓93 bezüglich Files sind:

- Neben den Signalen, Variablen und Konstanten erhalten auch die Files explizit den Status einer eigenen Objektklasse.
- Files können als Argumente an Unterprogramme übergeben werden.
- Files können durch Angabe eines Prozeduraufrufes explizit geöffnet oder geschlossen werden. Damit wird auch eine Abfrage auf Existenz der Datei möglich.
- Files können neben "read" und "write" den Modus "append" annehmen.

Mit diesen Neuerungen ist es, zusammen den "impure functions", nun möglich, mit Hilfe verschiedener Funktionen aus einem File zu lesen.

11.6 Zeiger

Wie in vielen Programmiersprachen kann auch in VHDL-Modellen mit Zeigern gearbeitet werden. Dadurch lassen sich sehr abstrakte, implementierungsunabhängige Modelle für elektronische Systeme erstellen. Beispiele für die Anwendung von Zeigern sind dynamische Warteschlangen oder Kellerspeicher.

In VHDL sind Zeiger spezielle Variablen, die die Adresse für ein Objekt speichern, das selbst nicht direkt ansprechbar ist, also keinen eigenen Bezeichner (identifier) besitzt. Da Zeiger immer Variablen sind, können sie nur innerhalb von Prozessen oder Unterprogrammen eingesetzt werden.

Typdeklaration von Zeigern

Bevor eine Zeigervariable deklariert werden kann, ist die Deklaration des Zeigertyps (sog. "access-type") unter Angabe des Typs, auf den gezeigt werden soll (`type_name`), erforderlich:

```
TYPE pointer_type IS ACCESS type_name ;
```

Deklaration von Zeigern

Die Zeigervariable kann dann in drei Varianten deklariert werden:

```
VARIABLE pointer_name : pointer_type
        := NEW type_name ;

VARIABLE pointer_name : pointer_type
        := NEW type_name'(def_value) ;

VARIABLE pointer_name : pointer_type
        := NULL ;
```

Bei der letzten Variante wird lediglich ein Zeiger angelegt, der auf kein Objekt zeigt. Die beiden anderen Varianten hingegen reservieren durch das Schlüsselwort `NEW` für ein Objekt vom Typ `type_name` den notwendigen Speicherplatz, legen dieses Objekt an und weisen ihm einen Defaultwert zu. Wird dieser Wert nicht wie in der zweiten Variante explizit angegeben, so entspricht der Defaultwert dem am weitesten links stehenden Wert in der Deklaration von `type_name`.

Um den Speicherplatz eines Objektes wieder freizugeben, existiert die Prozedur `deallocate`, deren einziges Argument der entsprechende Zeigername ist:

```
deallocate (pointer_name) ;
```

Anwendung von Zeigern

Folgendes Beispiel soll die Anwendung von Zeigern verdeutlichen:

```
ENTITY acc_types IS
END acc_types;
```



```
ARCHITECTURE behavioral OF acc_types IS
BEGIN
  PROCESS
    TYPE pos2 IS ARRAY (1 DOWNTO 0) OF positive;   -- (1)
    TYPE access_pos2 IS ACCESS pos2;               -- (2)
    VARIABLE p1 : access_pos2 := NEW pos2'(2,3);   -- (3)
    VARIABLE p2 : access_pos2 := NEW pos2;         -- (4)
    VARIABLE p3,p4 : access_pos2 := NULL;          -- (5)
  BEGIN
    p2.ALL (0) := 8;                               -- (6)
    p2 (1) := 7;                                   -- (7)
    p3 := p2;                                      -- (8)
    p4 := NEW pos2;                                -- (9)
    p4.ALL := p2.ALL;                              -- (10)
    deallocate (p1);                               -- (11)
    WAIT;
  END PROCESS;
END behavioral;
```

In Zeile (2) wird ein Zeigertyp für den in Zeile (1) deklarierten Vektortyp deklariert. Die Zeilen (3) bis (5) deklarieren die vier Zeigervariablen p1 bis p4. Zusätzlich wird in Zeile (3) und (4) jeweils ein nicht benanntes Objekt von Typ pos2 angelegt und initialisiert. Nach Abarbeitung dieser Zeilen, d.h. nach Ausführung der Prozeßinitialisierung, ergibt sich die in Abb. B-22 links dargestellte Situation.

In den Zeilen (6) und (7) werden den Elementen des Objektes, auf das der Zeiger p2 zeigt, Werte zugewiesen. Das Schlüsselwort ALL bewirkt ein sog. "Dereferenzieren" des Zeigers, d.h. daß nicht der Zeiger, sondern das Objekt, auf das gezeigt wird, angesprochen ist. Das Schlüsselwort ALL kann entfallen, wenn eine Bereichseinschränkung (wie in Zeile (7)) verwendet wird.

In Zeile (8) wird dem Zeiger p3 die Adresse des Objektes übergeben, auf das der Zeiger p2 zeigt. Zeile (9) bewirkt durch das Schlüsselwort NEW eine Speicherplatzreservierung für ein Objekt. Die Adresse dieses Objekts wird im Zeiger p4 gehalten. Sein initialer Wert wird durch die Zuweisung in Zeile (10) überschrieben.

Mit der Prozedur deallocate() wird in Zeile (11) der Speicherbereich des Objekts freigegeben, auf das p1 gezeigt hat.

Nach Abarbeitung der Zeilen `(6)` bis `(11)` stellt sich die in Abb. B-22 rechts dargestellte Situation ein.

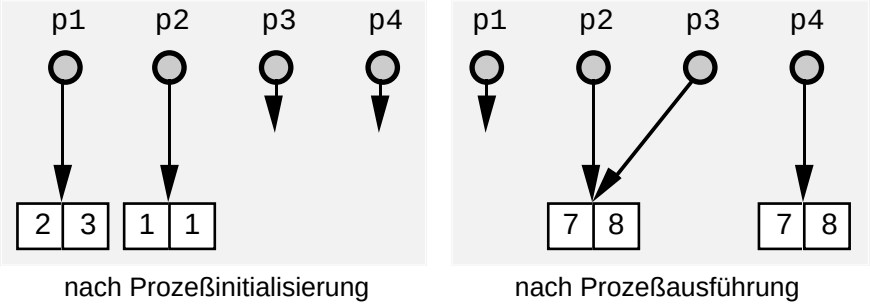

Abb. B-22: Verwendung von Zeigern

Unvollständige Typdeklaration

Um mit VHDL auch rekursive Strukturen modellieren zu können, sind neben den Zeigern auch die sog. "**incomplete types**" notwendig. Wie der Name besagt, stellen diese eine unvollständige Typdeklaration dar:

```
TYPE type_name ;
```

Die zugehörige vollständige Typdeklaration muß innerhalb desselben Deklarationsbereiches nachfolgen.

Anwendung von unvollständigen Typdeklarationen

Die Anwendung einer unvollständigen Typdeklaration zur Modellierung einer verketteten Liste wird am Beispiel einer Warteschlange gezeigt. Sie besteht aus beliebig vielen Elementen des Typs `chain_element`, die aus dem eigentlichen Datenelement vom Typ `integer` und einem Zeiger bestehen. Die Verkettung wird dadurch realisiert, daß der Zeiger jedes Elementes auf das nächste Element zeigt.

Zur Handhabung der Funktionalität sind zusätzlich zwei Zeiger (`first` und `last`) erforderlich, die auf das erste bzw. letzte Element der Warteschlange zeigen (Abb. B-23).

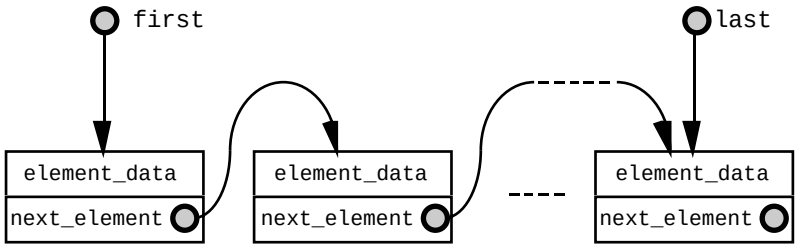

Abb. B-23: Aufbau der Warteschlange

Das VHDL-Modell zur Realisierung einer Warteschlange mit diesem Aufbau hat zwei Schnittstellen: Am Port `command` wird der Befehl für eine Operation auf der Warteschlange übergeben. Die Werte der Datenelemente werden über den Port `value` ausgegeben bzw. eingelesen.

```
ENTITY queue IS
  PORT (command : IN    queue_access;
        value   : INOUT integer);
END queue;
```

Die Alternativen für das Kommando sind im Typ `queue_access` deklariert:

- `nul`: keine Aktion,
- `add_element`: füge Element hinten (in Abb. B-23 rechts) an die Warteschlange an, wobei der aktuell anliegende Wert des Ports `value` abgespeichert wird,
- `delete_element`: lösche das erste Element (ganz links) der Warteschlange nachdem dessen Wert am Port `value` ausgegeben wurde.
- `read_element`: lese das erste Element, ohne es zu löschen. Auch hier wird dessen Wert über den Port `value` bereitgestellt.

```
ARCHITECTURE demo OF queue IS
  TYPE chain_element;              -- unvollstaendige Dekl.
  TYPE pointer IS ACCESS chain_element;
  TYPE chain_element IS RECORD     -- vollst. Deklaration
    next_element : pointer;
    element_data : integer;
  END RECORD;
BEGIN
  handle_queue : PROCESS (command)
    VARIABLE empty_flag : boolean := true;
    VARIABLE first, last, help : pointer := NULL;
  BEGIN
    CASE command IS
      WHEN add_element =>
        IF empty_flag THEN
          first := NEW chain_element; last  := first;
          empty_flag := false;
        ELSE
          last.next_element := NEW chain_element;
          last := last.next_element;
        END IF;
        last.element_data := value;
      WHEN delete_element =>
        IF empty_flag THEN
          ASSERT false REPORT "Empty queue!" SEVERITY note;
        ELSE
          value <= first.element_data, 0 AFTER 10 ns;
          help := first;
          IF first = last THEN empty_flag := true;
          ELSE first := first.next_element; END IF;
          deallocate (help);
        END IF;
      WHEN read_element =>
        IF empty_flag THEN
          ASSERT false REPORT "Empty queue!" SEVERITY note;
        ELSE
          value <= first.element_data, 0 AFTER 10 ns;
        END IF;
      WHEN nul => NULL;
    END CASE;
  END PROCESS;
END demo;
```


Das Beispiel zeigt, wie der Typ `chain_element` mit Hilfe der unvollständigen Typdeklaration im Typ `pointer` bereits verwendet werden kann, bevor er vollständig deklariert wird. Gleichzeitig wird deutlich, wie ein Record und dessen Elemente durch seinen Zeiger mit Hilfe von "selected names" referenziert werden kann.

Funktionsweise des Modells:

Bei der Prozeßinitialisierung werden keine Speicherplätze für Elemente der Warteschlange allokiert. Dies geschieht erstmals durch das Kommando `add_element`. In diesem Fall wird der Zeiger `first` für das hinzuzufügende Element verwendet. Sind bereits Elemente vorhanden, wird das neue Element durch `last.next_element` referenziert. Der Zeiger `last` wird im Anschluß daran auf das jeweils letzte Element gesetzt.

Das Löschen eines Elementes (`delete_element`) erfordert den temporären Zeiger `help`, der das später zu entfernende Element referenziert. Hier muß geprüft werden, ob es sich um das letzte Warteschlangenelement handelt. Der Wert des zu löschenden Elementes wird an den Port `value` angelegt und der Zeiger `first` wird bei noch vorhandenen weiteren Elementen auf das nächste Element gesetzt.

Das Kommando `read_element` gibt lediglich den Datenwert des ersten Elementes an den Ausgang weiter, ohne daß ein Element gelöscht wird.

11.7 Externe Unterprogramme und Architekturen

VHDL bietet sehr viele Funktionen zur Handhabung von vordefinierten Typen. Für eigene Typen können Routinen selbst verfaßt werden. Trotzdem kann es zur Handhabung komplexer Operationen oder großer Datenmengen sinnvoll sein, die Fähigkeiten von höheren Programmiersprachen zu nutzen. Prinzipiell ist es daher möglich, durch geeignete Schnittstellen externe Unterprogramme mit VHDL-Modellen zu verbinden. Eine weitere Anwendung von Schnittstellen wäre die Anbindung von fremden Architekturen oder gar von kompletten Modellen anderer Simulatoren.

Das LRM von ✓87 sieht zwar die Möglichkeit von Schnittstellen zu externen Funktionen über Packages vor, gibt aber über nähere Details keine Auskunft, d.h. es erlaubt die Verwendung, gibt aber keine Vorgaben, wie die Schnittstelle aussehen muß, wie die externen Funktionen anzusprechen sind, usw.

Die Situation hat sich heute dahingehend entwickelt, daß manche Softwarefirmen simulatorspezifische, und damit uneinheitliche Schnittstellen zu externen Funktionen (üblicherweise in "C") vorsehen, deren Handhabung aber alles andere als intuitiv ist.

Mit ✓93 wird durch die Definition eines Attributes zumindest der Versuch einer einheitlichen Handhabung unternommen. Die Definition der Schnittstelle (Signal-/Datentypen und -modi) bleibt aber weiterhin dem Softwarehersteller überlassen, ist also nach wie vor implementierungsabhängig.

Im Package `standard` von ✓93 ist dazu das Attribut `FOREIGN` deklariert, das als einziges vordefiniertes Attribut vom Benutzer zugewiesen werden kann:

```
ATTRIBUTE FOREIGN : string;
```

Externe Unterprogramme und Architekturen erhalten nun einheitlich einen VHDL-Rahmen und werden mit einem Attribut `FOREIGN` dekoriert. Dieses Attribut hat implementierungsabhängige Bedeutung und stellt die Verbindung zwischen dem VHDL-Bezeichner und dem Namen der externen Architektur bzw. des Unterprogramms her. Diese können dann wie herkömmliche Architekturen instantiiert oder wie normale Unterprogramme verwendet werden. Ein Beispiel:

```
ARCHITECTURE ext OF anything IS
  ATTRIBUTE FOREIGN OF ext : ARCHITECTURE IS "lib_z/aoi223";
BEGIN
END ext;
```