VHDL Kurzbeschreibung

A. Mäder
Universität Hamburg – Fachbereich Informatik
Arbeitsbereich Technische Grundlagen der Informatik

Inhaltsverzeichnis

1 Konzepte von VHDL

VHDL ist eine Hardwarebeschreibungssprache; dabei steht der Name für:
VHSIC **H**ardware **D**escription **L**anguage
Very **H**igh **S**peed **I**ntegrated **C**ircuit

VHDL wurde 1983 vom amerikanischen Department of Defense initiiert und ist seit Ende 1987 als IEEE Standard 1076 genormt. Inzwischen ist VHDL die *quasi* Standard-Hardwarebeschreibungssprache.

1.1 Grundkonzepte

Die Sprache VHDL dient der Beschreibung und Simulation digitaler Systeme (ASICs) und deren Umgebung. Alle, während des Entwurfsvorgangs anfallenden Beschreibungen der Schaltung, werden von Sprachumfang abgedeckt. Dabei ist die Schaltung jederzeit simulierbar. Dementsprechend sind in VHDL die folgenden Konzepte verwirklicht:

Hierarchie: die Unterteilung des Entwurfs in (Teil-)Komponenten wird unterstützt. Der Gesamtentwurf wird dann hierarchisch aus diesen Teilen zusammengesetzt. Die Komplexität dieser Teile kann vom einfachen Gatter (z.B. NAND) bis hin zu komplexen Funktionseinheiten (z.B. Prozessorkern) reichen.

Abstraktion: jede Design-Einheit (Entity) kann auf unterschiedliche Weise beschrieben sein. Dabei ist grundsätzlich zwischen folgenden Möglichkeiten zu unterscheiden:

- Verhalten : Algorithmische Beschreibung mit den Mitteln einer höheren Programmiersprache.
- Datenfluß : Mischform zwischen Struktur- und Verhaltensbeschreibung. Beschreibt die Struktur von Datenpfaden, während die Operationen auf den Daten als *elementare* Funktionen vorhanden sind.
- Struktur : Darstellung der strukturellen Verbindung von Elementen im Sinne einer Hierarchie.

Durch die Definition benutzereigener Datentypen kann die Aufgabe unabhängig von konkreten Hardwarerealisierungen spezifiziert werden.

Datenhaltung: das in der Sprache benutzte Bibliothekskonzept erlaubt:

- den Zugriff auf gemeinsame Datenbestände durch Arbeitsgruppen
- die Wiederverwendung bestehender (Teil-)Entwürfe
- das Einbinden herstellerspezifischer Bibliotheken (z.B. für Standardzellen)

1.2 Design-Paradigmen

Der eigentliche Design-Prozeß, bei Verwendung von VHDL, entspricht einem *Top-Down* Vorgehen.

Algorithmendesign: Ausgangspunkt ist die Verhaltensbeschreibung der Schaltung, deren Funktionalität durch Simulationen geprüft wird. So können verschiedene Algorithmen implementiert und miteinander verglichen werden.

Auf oberster Ebene ist dies eine Beschreibung der zu entwerfenden ICs oder Systems, sowie eine Testumgebung, die das Interface zu der Schaltung darstellt.

Top-Down Strukturierung: Im weiteren Vorgehen wird die Schaltung in Funktionsblöcke gegliedert, bis man letztendlich eine Strukturbeschreibung erhält.

Dieser Prozeß — Algorithmischer Entwurf von Funktionseinheiten, hierarchische Verfeinerung und Umsetzung in Strukturbeschreibungen — wird rekursiv ausgeführt, bis man letztendlich bei Elementen einer Zellbibliothek angekommen ist und die Schaltung praktisch realisiert werden kann.

Dadurch, daß die Schnittstelle der Schaltung und deren eigentliche Realisierung voneinander getrennt sind, werden Alternativen im Entwurf unterstützt — *exploring the design-space.*

Durch den Einsatz von Synthesewerkzeugen wird die Entwurfsaufgabe (auf den unteren Abstraktionsebenen) dabei zunehmend vereinfacht: ausgehend von Verhaltensbeschreibungen werden Netzlisten für Zielbibliotheken generiert. Derzeitiger Stand der Technik ist, daß die Synthese für Logik (Schaltnetze) und für endliche Automaten problemlos beherrscht wird. Für die Synthese komplexerer Algorithmen gibt es viele gute Ansätze, die zumindest bei Einschränkungen auf bestimmte Anwendungsfelder (Einschränkung des Suchraums), mit den Entwürfen guter Designer konkurrieren können.

1.3 Bibliotheken und compilierbare Einheiten

Die Entwürfe sind in Bibliotheken organisiert, wobei die Bibliotheken jeweils compilierten und durch den Simulator ausführbaren VHDL-Code enthalten. Bibliotheken können folgende vier Teile enthalten:

`package`	: globale Deklarationen
`entity`	: Design – Sicht von Außen (black box)
`architecture`	: Design Implementation
`configuration`	: Festlegung einer Design-Version (Zuordnung: entity – architecture)

Neben herstellereigenen- und benutzerdefinierten Bibliotheken gibt es zwei Standardbibliotheken:

- `WORK` : default-Bibliothek des Benutzers. Wenn nichts anderes angegeben wird, ist `WORK` die Bibliothek, mit der der VHDL-Compiler und -Simulator arbeiten.
- `STD` : enthält die beiden Packages `STANDARD` und `TEXTIO` mit vordefinierten Datentypen und Funktionen.

Compilation und Simulation
VHDL-Beschreibungen werden in zwei Schritten bearbeitet:

1. Die *Analyse* (Compilation) prüft die Syntax und die Konsistenz des VHDL-Codes und schreibt die Ausgabe in die entsprechende Bibliothek (normalerweise `WORK`).

   Bei Verwendung von SYNOPSYS: `vhdlan`.

2. Bei der *Simulation* von Elementen wird dann die Funktion der eigegebenen Schaltung überprüft.

   Bei Verwendung von SYNOPSYS: `vhdlsim`, `vhdldbx`.

Benutzung von Bibliothekselementen
VHDL Bibliotheken erlauben bis zu drei Namensebenen:

lib_name.package_name.item_name	: volle Derefenzierung
lib_name.item_name	: bei Eindeutigkeit von item_name
item_name	: Bibliothek `WORK`, Eindeutigkeit von item_name

Um Bibliothekselemente im eigenen VHDL-Entwurf benutzen zu können, muß zuerst die Bibliothek definiert werden (`library` lib_name), anschließend kann die Auswahl von Packages aus den Bibliotheken erfolgen (`use` lib_name.package_name.item_name, bzw. `...all`).

Beispiel:
```
C <= not (A xor B) after FREDS_LIB.MY_DEFS.UNIT_DELAY;
```

— oder nach einer `use` Anweisung —
```
use FREDS_LIB.MY_DEFS.UNIT_DELAY;
C <= not (A xor B) after UNIT_DELAY;
```

1.3.1 Package

Üblicherweise dienen Packages dazu Gruppen von Deklarationen zusammenzufassen, die in mehreren Designs benutzt werden, z.B. für
- Komponenten einer Bibliothek
- Unterprogramme (Funktionen, Prozeduren)
- Typen, Konstanten, Dateien...

Die Zellbibliotheken der ASIC-Hersteller werden so über spezielle Bibliotheken und Packages zur Vefügung gestellt. Auf der anderen Seite liefern die Hersteller von CAD-Software Hilfsroutinen und Funktionen, die den Umgang mit den Werkzeugen erleichtern, in Form von Packages mit.

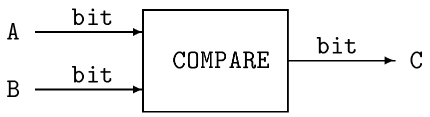
1.3.2 Entity

Ein `entity` definiert für eine Komponente des Entwurfs die externe Sichtweise. Dabei werden der Name, die Ein- und Ausgänge und zusätzliche Deklarationen festgelegt. Nach der Übersetzung in eine Bibliothek kann ein `entity` dann als Teil anderer Entwürfe benutzt werden. Die interne Realisierung wird dann als `architecture` beschrieben.

Syntax:
```
entity entity_name is
   [generics]
   [ports]
   [declarations (constants, ports, signals)]

[begin                                              normalerweise nicht benutzt
   statements]
end [entity_name];
```

Das nachfolgende Beispiel zeigt die Entity-Beschreibung eines Bitvergleichers

Beispiel:
```
entity COMPARE is
   port (  A, B: in bit;                  Ein-/Ausgänge: Richtung und Datentyp
              C: out bit);
end COMPARE;
```



1.3.3 Architecture

Eine Architektur beschreibt das Verhalten, bzw. die interne Realisierung einer vorher analysierten `entity`. Dabei können mehrere Architekturen für eine `entity` deklariert werden (Unterstützung von Alternativen), die Auswahl einer bestimmten `architecture` erfolgt dann später über eine `configuration`.

Syntax:
```
architecture architecture_name of entity_name is
   [arch_declarative_part]

begin
   [arch_statement_part]
end [architecture_name];
```

Der arch_declarative_part beschreibt Typen, Unterprogramme und Konstanten, die nur für diese Architektur benötigt werden, während der arch_statement_part das eigentliche Design beschreibt. Für diese Beschreibung gibt es drei *Stile*: Verhalten, Datenfluß, Struktur. Innerhalb des arch_statement_part kann einer oder eine beliebige Kombination dieser Stile benutzt werden.

Verhalten
Das Grundkonstrukt der Verhaltensbeschreibung ist der `process`.

interner Aufbau: Ein VHDL-Prozeß ist einem Programm einer Programmiersprache vergleichbar, mit den üblichen Konzepten:

- sequentielle Abarbeitung der Anweisungen
- Kontrollanweisungen zur Steuerung des Ablaufs
- Verwendung lokaler Variablen, -Datentypen
- Unterprogrammtechniken (Prozeduren und Funktionen)

Aktivierung: Da das Ziel der VHDL Beschreibung ein durch den Simulator ausführbares Verhalten ist, gibt es spezielle Konstrukte, die festlegen wann der Prozeß zu aktivieren ist — im Gegensatz zu Programmen in herkömmlichen Sinne sind Hardwareelemente *immer*, *gleichzeitig* aktiv.

Ein Prozeß hat deshalb üblicherweise entweder eine *sensitivity list* oder enthält *wait*-Anweisungen. Beide Methoden bewirken, daß bei der Änderung von Eingangswerten der Architektur der Prozeß von Simulator aktiviert wird, die Anweisungen sequentiell abgearbeitet werden und dadurch neue Ausgangswerte erzeugt werden.

Ein-/Ausgabe: Nur über *Signale* können Prozesse nach außen hin aktiv werden. Diese Signale können zum einen Ein- und Ausgänge der Schaltung sein, zum anderen können auch mehrere Prozesse über (architektur-)interne Signale kommunizieren.

In Gegensatz zu den Signalen können die *Variablen* des Prozesses nur prozessintern verwendet werden und stellen so etwas wie lokalen Speicher dar.

Um Parallelitäten im Verhalten einer `architecture` zu beschreiben, können innerhalb des Anweisungsteils beliebig viele Prozesse beschrieben werden.

Beispiel:
```
architecture ARCH_BEHAV of COMPARE is
begin
  process (A, B)                        Prozeß, sensitiv zu Eingängen A und B
  begin
    if (A = B) then
      C <= '1' after 1 ns;
    else
      C <= '0' after 2 ns;
    end if;
  end process;
end ARCH_BEHAV;
```

Datenfluß

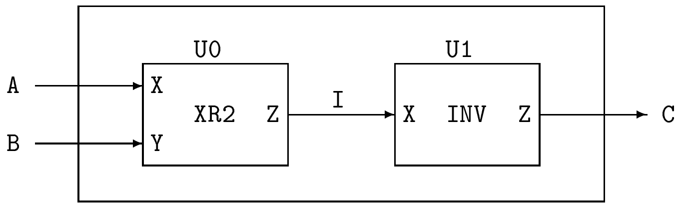
Bei dieser Beschreibung wird der Datenfluß über kombinatorische logische Funktionen (Addierer, Komparatoren, Decoder, Gatter…) modelliert.

Beispiel:
```
architecture ARCH_DATFL of COMPARE is
begin
  C <= not (A xor B) after 1 ns;
end ARCH_DATFL;
```
Datenfluß von Eingängen, über `xor` und `not`, zum Ausgang

Struktur

Strukturbeschreibungen sind Netzlisten aus Bibliothekselementen: diese Elemente werden instanziiert und über Signale miteinander verbunden.

Beispiel:
```
architecture ARCH_STRUC of COMPARE is
signal I: bit;                                   -- lokales Signal

component                                        -- Komponentendeklaration
  XR2 port (X,Y: in bit; Z: out bit);            -- der Bibliothekselemente
end component;
component
  INV port (X: in bit; Z: out bit);
end component;

begin                                            -- Beschreibung der Netzliste
  U0: XR2 port map (A,B,I);                      -- Benutzung der Komponenten
  U1: INV port map (I,C);
end ARCH_STRUC;
```



1.3.4 Configuration

Durch Konfigurationen kann der Entwerfer zwischen verschiedenen Design-Alternativen und -Versionen auswählen. Dabei bestimmt eine Konfiguration, welche Realisierung — von möglicherweise mehreren vorhandenen Architekturen — für ein Entity aus der Bibliothek, bei dessen Instanziierung benutzt wird.

Syntax:
```
configuration conf_name of entity_name is
  conf_specification
end [conf_name];
```

Existiert zu einem `entity` keine expliziten Konfiguration, so wird jeweils die (zeitlich) zuletzt analysierte Architektur benutzt — die *null* Konfiguration.

In dem nachfolgenden Beispiel wird das `entity COMPARE` in einer ALU benutzt (strukturelle Beschreibung der `architecture ... of ALU`).

Beispiel:

```
entity ALU is                                        Entity Beschreibung der ALU
  port ( opcode: ...
end ALU;

architecture FIRST of ALU is          Architektur der ALU (Strukturbeschreibung)
  component COMPARE
    port (A,B: in bit; C: out bit);
  end component;
  ...
begin
  U0: COMPARE port map (S,D,Q);                 Instanz U0 des entity COMPARE
  ...
end FIRST;

configuration FAST_ONE of ALU is
  for FIRST                                      Architektur die konfiguriert wird
    for U0: COMPARE use entity WORK.COMPARE(ARCH_BEHAV)
    ...                          legt Entitynamen und dessen Architektur fest
  ...
end FAST_ONE;
```

Da Konfigurationen seperate Entwurfseinheiten sind, können sie auch direkt für die Simulation benutzt werden. In obigem Beispiel wäre es möglich diese Konfiguration zu simulieren als: `vhdlsim WORK.FAST_ONE`.

1.4 zeitliche Kontrolle

Bei der Einführung eines Zeitbegriffs innerhalb der Simulation, muß man unterscheiden zwischen der Art und Weise, wie Anweisungen im VHDL-Code sequentiell abgearbeitet werden und dem Verstreichen von simulierter Zeit.

Verhaltensbeschreibungen : Wie in Programmiersprachen, wird das Verhalten durch einen *sequentiellen Ablauf* von Anweisungen beschrieben. Die Abarbeitung dieser Anweisungen *beeinflußt die simulierte Zeit nicht*.

Strukturbeschreibungen : Entsprechend realen Schaltungen sind alle Strukturelemente *parallel aktiv*, eine Reihenfolge der entsprechenden Anweisungen im Code ist irrelevant. Die Verzögerungen der einzelnen *Elemente bestimmen die simulierte Zeit.*

Üblicherweise wird für die Simulation von VHDL ein ereignisgesteuerter Simulator mit zwei-Listen Technik benutzt[1]. Eine verwaltet die vorherigen Werte der Signale, die andere Liste die neu berechneten.

Eine fiktive Zeiteinheit (*delta-time*) erlaubt die Behandlung von Signalzuweisungen ohne Verzögerungszeit. Dadurch kann der Simulator die Grundschleife der Simulation mehrfach durchlaufen, ohne daß die simulierte Zeit fortschreitet. Um die Wirkungsweise von Signalzuweisungen besser zu verstehen, sei hier kurz der Simulationszyklus skizziert:

1. Aktivierung des Zyklus zu einem Zeitpunkt t_0 durch Signalzuweisungen, die für diesen Zeitpunkt im Schedule sind.

2. Durch diese Signaländerungen werden alle parallelen Anweisungen und Prozesse, die zu diesen Signalen sensitiv sind, aktiviert.

3. In einer ersten Phase werden die Signale der aktiven Prozesse ausgewertet und Ausdrücke berechnet.

4. In der zweiten Phase werden dann neue Werte an die Signale zugewiesen. Entsprechend den Verzögerungszeiten werden die Werteänderungen in die Scheduling-Tabelle übernommen.

5. Damit ist ein Durchlauf für den Zeitpunkt t_0 abgeschlossen. Waren bei den Signalzuweisungen in Schritt 4 auch welche ohne Verzögerungszeit, so wird der Simulationszyklus beginnend bei Schritt 2 wiederholt — der Zeitschritt ist *delta-time*, die simulierte Zeit bleibt bei t_0.

[1]In Bezug auf Simulationsalgorithmen sei auf die entsprechenden Vorlesungen (CAD-Algorithmen...) verwiesen

2 Datentypen

VHDL ist eine stark *typisierte* Sprache, d.h. Konstante, Variablen und Signale haben einen, durch deren Deklaration festgelegten Typ. Bei der Codeanalyse wird die Konsistenz der Datentypen bei Operationen und Zuweisungen überprüft. Gegebenenfalls müssen *Konvertierungsfunktionen* benutzt werden.

Neben den Typen des Package `STANDARD`, die *immer* bekannt sind, werden hier auch noch Typen aus den folgenden zwei Packages vorgestellt:

`std_logic_1164`	in Bibliothek `IEEE`
`textio`	in Bibliothek `STD`

2.1 Skalare

Die einfachen Datentypen in VHDL sind denen in Standard-Programmiersprachen vergleichbar:

Character : die `character` Literale entsprechen dem Standard ASCII Zeichensatz, die darstellbaren Zeichen werden dabei in einfache Hochkommas eingeschlossen: `'0'`–`'9'`, `'a'`–`'z'`, `'A'`–`'Z'`.[2]

Bit : die beiden logischen Werte `'0'` und `'1'` sind `bit` Literale.[2]

Std_Logic : der IEEE Standard 1164 ist in einem extra Package `std_logic_1164` in der Library `IEEE` definiert. Dabei wird ein Logiksystem mit neun Signalwerten, bzw. Treiberstärken definiert, das für die Simulation und Synthese von Hardware besser geeignet ist als der Typ Bit. Die Werte sind im einzelnen:

`'U'`	noch nicht initialisiert
`'X'`	treibend *unbekannt*
`'0'`	treibend *logische 0*
`'1'`	treibend *logische 1*
`'Z'`	hochohmig – für Busse mit three-state
`'W'`	schwach *unbekannt*
`'L'`	schwach *logische 0*
`'H'`	schwach *logische 1*
`'-'`	don't care – für Logiksynthese

Zusammen mit diesen Datentyp, der ja auch Werte für die Modellierung von three-state Bussen beinhaltet, ist eine Auflösungsfunktion definiert, die mehrere Treiber auf einer Leitung zuläßt (siehe 6.3, Seite 36).

Boolean : die beiden boole'schen Werte `true` und `false` sind `boolean` Literale.

[2] Wegen der Typbindung in VHDL kann es notwendig sein, zur Unterscheidung der Typen deren Werte explizit zu klassifizieren: `character'('1')`
`bit'('1')`

Integer : Zahlen sind möglich von $-2^{31}-1$ bis $+2^{31}-1$ ($-2\,147\,483\,647$ bis $+2\,147\,483\,647$). Die default-Zahlendarstellung ist dezimal; bei der Benutzung anderer Zahlenbasen wird explizit angegeben:

binär	`2#...#`
octal	`8#...#`
hexadecimal	`16#...#`

Format: [+|-][2#|8#|16#]number[#]

Meist sind zu `integer` noch Untertypen definiert, wie:

`POSITIVE`	: 1...n
`NATURAL`	: 0...n

Real : Zahlen sind möglich von $-1.0e+38$ bis $+1.0e+38$, die default-Zahlendarstellung ist dezimal.

Format: [+|-][2#|8#|16#]number.number[#][e[+|-]number]

Physikalische Literale : zusätzlich zu den eigentlichen Wertangaben erlaubt VHDL noch die Angabe physikalischer Maßeinheiten. In dem `package STANDARD` sind als Typ `TIME` folgende Maßeinheiten definiert: `fs, ps, ns, us, ms, sec, min, hr`. Dadurch können Verzögerungszeiten modelliert werden: `C <= ... after 2 ns;`

Aufzählungstypen : um die Aufgabenstellung unabhängig von speziellen technischen Codierungen beschreiben zu können, kann der Benutzer beliebige Aufzählungstypen definieren.[3]

Syntax: `type enum_name is (enum_liter {, enum_liter});`

Dabei können für enum_liter beliebige Bezeichner (erstes Beispiel) aber auch Literale (`character` Literale im zweiten Beispiel) benutzt werden.

Beispiel:

`type AMPEL is (ROT, GELB, GRUEN);`	für Ampelsteuerung
`type FOURVAL is ('X', '0', '1', 'Z');`	vierwertige Logik für Simulation

Untertypen : zu vordefinierten, bzw. zu eigenen Typen lassen sich Untertypen bilden, dies geschieht üblicherweise durch Einschränkung der Wertebereiche. Solche Bereichseinschränkungen lassen sich zwar auch bei der Variablen- und Signaldeklaration angeben, Untertypen bieten aber den Vorteil, daß solche Begrenzungen zentral vorgenommen werden können.

[3] Die Typen `character`, `bit` und `boolean` sind im `package STANDARD` als Aufzählungstypen vordefiniert.

Beispiel:

```
subtype DIGIT is integer range 0 to 9;        -- Untertyp zu integer
...
variable MSD, LSD: DIGIT;
```

— ist äquivalent zu —

```
variable MSD, LSD: integer range 0 to 9;
```

2.2 komplexe Typen

Array : Wie in Programmiersprachen bestehen Arrays aus durchnumerierten Elementen gleichen Typs. Dabei sind in entsprechenden Packages schon folgende Array-Typen vordefiniert:

String : Array Typ zu `character` — in Package `STANDARD`

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

Bit_Vector : Array Typ zu `bit` — in Package `STANDARD`

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;
```

Std_Logic_Vector : Array Typ zu `std_logic` — in Package `STD_LOGIC_1164`

```
type STD_LOGIC_VECTOR is array ( NATURAL range <>) of STD_LOGIC;
```

Da die Hersteller für diesen Datentyp auch gleich logische und arithmetische Operatoren anbieten, kann es notwendig sein die Zahlendarstellung genau zu spezifizieren, um beispielsweise zwischen *unsigned* und *signed* Darstellung zu unterscheiden (siehe 4, Seite 23). Gegebenenfals muß die Zahlendarstellung über die Bindung an Untertypen explizit dargestellt werden. In dem Package `STD_LOGIC_ARITH` sind deshalb definiert:

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

Die Deklaration eigener Array Typen besteht dabei aus der Angabe des Arraytyp-Namens, der Beschreibung des Index (der Indices bei mehrdimensionalen Arrays) durch Index-Typ(en) und Index-Bereich(e) und der Angabe des Element-Typs.

Syntax: `type array_name is array (index_description) of element_type;`

Möglichkeiten für index_description

`index_range |`	`integer` Bereich
`index_type |`	Aufzählungstyp als Index
`index_type range index_range |`	allg. Typ-Bereich
`index_type range <>`	unbegrenzter Typ; Bereichsangabe bei Var-Dekl.

Nachfolgend werden einige Eigenschaften von Arrays anhand von Beispielen genauer erläutert.

Index-Typen : neben den üblichen Integer-Indices können auch eigene Aufzählungstypen (-untertypen) benutzt werden.

Beispiel:
```
type INSTRUCTION is (ADD, SUB, LDA, LDB, STA, STB, OUTA);
subtype FLAGS is integer range (0 to 7);
...
type INSTR_FLAG is array (INSTRUCTION) of FLAGS;
```
Array von Flag-Werten

Benutzung als Laufindex : Indices können innerhalb von Schleifen über Variablen incrementiert/decrementiert werden.

Beispiel:
```
...
process ...
  variable INFO : bit_vector (0 to 49);
  variable START : integer;
  variable OUTBYTE : bit_vector (0 to 7);
begin
  for I in 0 to 7 loop
    OUTBYTE(I) := INFO(I + START);
  end loop;
end process;
```

Unbegrenzte Indices : oft werden Indices über den gesamten Wertebereich des Aufzählungstypen deklariert und dann später bei der Variablendeklaration werden erste Bereichseinschränkungen vorgenommen.

Beispiel:
```
type BIT_VECTOR is array (NATURAL range <>) of BIT;
...                                          Deklaration aus STANDARD
variable BYTE: BIT_VECTOR (0 to 7);
```

Index-Bereiche : vergleichbar den Wertebereichseinschränkungen von Variablen ist die Reihenfolge des Index wichtig.

Beispiel:
```
type AVEC is array (0 to 3) of bit;
type BVEC is array (3 downto 0) of bit;
...
variable AV: AVEC;
variable BV: BVEC;
...
AV := "0101";       ⇒ AV(0)='0' AV(1)='1' AV(2)='0' AV(3)='1'
BV := "0101";       ⇒ BV(0)='1' BV(1)='0' BV(2)='1' BV(3)='0'
```

Array-Zuweisungen : können entweder über die *Position*, den *Namen* oder gemischt erfolgen.

Syntax bei Namen:
```
[typ_name']                               optionale Typ-Qualifizierung
( selector => expression{,
  selector => expression}[,
  others => expression] )
```

Beispiel:
```
variable C: BIT_VECTOR (0 to 3);
variable H, I, J, K: bit;

mögliche Zuweisungen
C := "1010";                                          4-bit string
C := H & I & J & K;                                   Konkatenation
C := ('1', '0', '1', '0');                            Aggregat

Aggregatzuweisungen
C := ('1', I, '0', J or K);                           Position
C := (0 => '1', 3 => J or K, 1 => I, 2 => '0');       Name
C := ('1', I, others => '0');                         gemischt
```

Auschnitte : können über die Indizes gebildet werden.

Beispiel:
```
variable A: BIT_VECTOR (3 downto 0);
variable B: BIT_VECTOR (8 downto 1);
...
B(6 downto 3) := A;                                   4 bit Slice von A
```

Mehrdimensionale Arrays : können über mehrere Indices erzeugt werden.

Beispiel:
```
type MEMORY is array (0 to 7, 0 to 3) of bit;         8 × 4 bit Array
...
constant ROM: MEMORY := ( ('0','0','0','0'),
                          ('0','0','0','1'),
                          ('0','0','1','0'),
                          ('0','0','1','1'),
                          ('0','1','0','0'),
                          ('0','1','0','1'),
                          ('0','1','1','0'),
                          ('0','1','1','1'));
variable DATA_BIT: bit;
...
DATA_BIT := ROM (5,3);                                ist '1'
```

Es lassen sich auch Arrays von Arrays bilden; die Indices werden hierbei getrennt behandelt.

Beispiel:
```
type WORD is array (0 to 3) of bit;                 4 bit Zeile
type MEMORY is array (0 to 7) of WORD;              8 × 4 bit Array
...
constant ROM: MEMORY := ( ('0','0','0','0'),
                          ('0','0','0','1'),
                          ...
                          ('0','1','1','1'));
variable DATA: WORD;
variable DATA_BIT: bit;
variable ADDR, INDEX: integer;
...
DATA := ROM (ADDR);
DATA_BIT := ROM (ADDR)(INDEX);
```

Array Untertypen : lassen sich zu bestehenden Arrays, bzw. zu unbegrenzten Arrays definieren.

Beispiel:
```
subtype BYTE is BIT_VECTOR (7 downto 0);        unbegrenzter Typ
                                                BIT_VECTOR
```

Record : Elemente verschiedener Typen (Skalare oder zusammengesetzte Typen) können mit Hilfe von Records zusammengefaßt werden, um abstrakte Datenmodelle zu bilden. Die einzelnen Felder des Records werden über die Namen dereferenziert.

Beispiel:
```
type TWO_DIGIT is                               Zahlen von −99 bis +99
  record  SIGN  : bit;
          MSD   : integer range 0 to 9;
          LSD   : integer range 0 to 9;
  end record;
...
process ...
  variable ACNTR, BCNTR: TWO_DIGIT;
begin
  ACNTR.SIGN := '1';                            Zugriff auf Felder
  ACNTR.MSD := 1;
  ACNTR.LSD := ACNTR.MSD;
  ...
  BCNTR := TWO_DIGIT'('0',3,6);                 Aggregat, Typ-qualifiziert
  ...
end process;
```

Alias Deklarationen : Eine zweite Möglichkeit, neben Records, auf Teilstrukturen über Namen zuzugreifen, bieten Alias-Namen. So lassen sich beispielsweise Teile von Arrays über solche selbstdefinierten Namen selektieren.

Nachfolgend wird der im vorigen Beispiel benutzte Record-Typ `TWO_DIGIT` mit Hilfe von `alias` Anweisungen nachgebildet.

Beispiel:

```
signal ACNT: bit_vector(1 to 9);                       vergl. voriges Beispiel
alias SIGN: bit is count(1);
alias MSD:  bit_vector(1 to 4) is count(2 to 5);
alias LSD:  bit_vector(1 to 4) is count(6 to 9);
...
SIGN  := '1';                                          Benutzung
MSD   := "1001";
LSD   := MSD;
...
COUNT := "110011001";
```

Text I/O : In dem `package TEXTIO` sind die Datentypen `text` und `line` deklariert, sowie Funktionen die den (aus Programmiersprachen gewohnten) Zugriff auf Textdateien erlauben; so ist es beispielsweise möglich Testvektoren aus einer Datei einzulesen und in der Simulation zu benutzen.

Beispiel:

```
use std.textio.all;        Standardbeispiel: Datei1 lesen, in Datei2 schreiben
entity COPY4 is                                      entity ohne Anschlüsse
end COPY4;

architecture FIRST of COPY4 is
begin
  process (go)
    file INSTUFF: text is in "\path\test.data";            Eingabedatei
    file OUTFILE: text is in "\path\out.data";             Ausgabedatei
    variable L1, L2 line;
    variable VECT: bit_vector(3 downto 0);

    begin
      while not (endfile(INSTUFF)) loop                 bis zum Dateiende
        readline (INSTUFF, L1);                         liest Zeile aus Datei
        read (L1, VECT);                                liest Vektor aus Zeile
        write (L2, VECT);                               schreibt Vektor in Zeile
        writeline (OUTFILE, L2);                        schreibt Zeile in Datei
      end loop;
    end process;
end FIRST;
```

In dem `package TEXTIO` sind unter anderem vordefiniert:

Typen:	`LINE`, `TEXT`
Standarddateien:	`INPUT`, `OUTPUT`
Prozeduren:	`READ`, `READLINE`, `WRITE`, `WRITELINE`
Funktionen:	`ENDLINE`, `ENDFILE`

Zugriffstypen : Wie Zeiger in Programmiersprachen können Zugriffstypen dazu benutzt werden dynamisch Speicherstrukturen zu alloziieren. Variablen vom Typ `access` sind Zeiger auf skalare oder komplexe Datenstrukturen.

Syntax: `type ptr_type_name is access type_name;`

Für die Arbeit mit Zugriffstypen sind zwei Operatoren definiert.

New : wird bei *Zuweisungen* an eine Variable des Zugriffstyps als Keywort benutzt. Dabei sind Initialisierungen möglich.

Wenn der Zeiger auf einen unbeschränkten Typ zeigt, sind Bereichseinschränkungen notwendig, wie z.B. bei `string`.

Deallocate : um Speicherbereiche wieder freizugeben steht eine *Prozedur* zur Verfügung, die als Parameter die Variable des Zugriffstyps übergeben bekommt.

Beispiel:

```
type CELL;                                   unvollständige Typdeklaration
type LINK is access CELL;                                      Zugriffstyp
type CELL is                                       genaue Typdeklaration
  record
    VALUE : integer;
    NEXTP : LINK;
  end;
variable HEAD, TEMP : LINK;                                Zeiger auf CELL
...
TEMP := new CELL'(0, null);           neues Element, mit Initialisierung
for I in 1 to 5 loop
  HEAD := new CELL;                                      weitere Elemente
  HEAD.VALUE := I;                                    Zugriff auf record
  HEAD.NEXTP := TEMP;
  TEMP := HEAD;
end loop;
...
deallocate(TEMP);                                        Speicherfreigabe
```

Speicheranforderung

```
new CELL;                                                    neues Element
new CELL'(I, TEMP);                                  ... mit Initialisierung
```

... mit notwendiger Bereichsbeschränkung

```
new BIT_VECTOR (15 downto 0);                                 durch Index
new BIT_VECTOR'("001101110");                       durch Initialisierung
```

2.3 Attribute

Symbolische Attribute in VHDL erlauben allgemeineren Code zu schreiben, da Konstanten oder Literale nicht an mehreren Stellen stehen müssen, sondern über Attributierungsmechanismen zum Zeitpunkt der Übersetzung ermittelt werden.

Dimensionierung : Die Attribute ermitteln für Array- und Aufzählungstypen, beziehungsweise Variablen und Signale dieser Typen, Bereichsgrenzen und Längen. Bei mehrdimensionalen Arrays wird die Ordnungsnummer des Index mit angegeben.

Syntax: Bereichsgrenzen

`...'left[(n)]`	linke Grenze (Index `n`)
`...'right[(n)]`	rechte Grenze (Index `n`)
`...'high[(n)]`	Obergrenze (Index `n`)
`...'low[(n)]`	Untergrenze (Index `n`)

Arraylängen

`...'length[(n)]`	Anzahl der Elemente (Index `n`)

Bereiche

`...'range[(n)]`	Bereich `..to / downto..` (Index `n`)
`...'reverse_range[(n)]`	Bereich `..downto / to..` (Index `n`)

Beispiel: Bereichsgrenzen

```
type T_RAM_DAT is array (0 to 511) of integer;
variable RAM_DAT: T_RAM_DAT;
...
for I in RAM_DAT'low to RAM_DAT'high loop
  ...
```

Bereichsgrenzen mehrdimensional

```
variable MEM (0 to 15, 7 downto 0) of MEM_DAT;
...
MEM'left(1)          ist 0
MEM'right(1)         ist 15
MEM'left(2)          ist 7
MEM'right(2)         ist 0
MEM'low(2)           ist 0
MEM'high(2)          ist 7
```

Arraylängen

```
type BIT4 is array (0 to 3) of BIT;
type BIT_STRANGE is array (10 to 30) of BIT;
...
BIT4'length          ist 4
BIT_STRANGE'length   ist 21
```

Bereiche

```
function VEC2INT (INVEC: bit_vector) return integer is
  ...
begin
  for I in INVEC'range loop
    ...
```

Ordnung : Die Attribute ermitteln für Aufzählungstypen Werte, Ordnungszahlen und übergeordnete Typen (bei Untertypen).

Syntax: Wertermittlung

`type'succ(value)`	Wert nach `value`
`type'pred(value)`	Wert vor `value`
`type'leftof(value)`	Wert links von `value`
`type'rightof(value)`	Wert rechts von `value`

Ordnung

`type'pos(value)`	Position von `value`
`type'val(position)`	Wert von `position`

übergeordnete Typen

`type'base`	Basistyp zu `subtype type`

Beispiel:

```
type COLOR is (RED, BLUE, GREEN, YELLOW, BROWN, BLACK);
subtype TLCOL is COLOR range RED to GREEN;
...
```

`COLOR'low`	ist `RED`
`COLOR'succ(RED)`	ist `BLUE`
`TLCOL'base'right`	ist `BLACK`
`COLOR'base'left`	ist `RED`
`TLCOL'base'succ(GREEN)`	ist `YELLOW`

3 Bezeichner und Deklarationen

Identifier dienen dazu Objekte zu benennen. Mit Ausnahme einiger reservierter Wörter kann der Benutzer beliebige Namen vergeben, dabei gilt:

1. Zeichensatz `'a'`... `'z'`, `'0'`... `'9'`, `'_'`.
2. das erste Zeichen muß ein Buchstabe sein.
3. keine Unterscheidung zwischen Groß- und Kleinschreibung in VHDL

Bei Verwendung von Bibliotheken und Packages müssen die Elemente gegebenenfalls über komplette Namen derefenziert werden, wie: lib_name.package_name.item_name

Kommentare : beginnen mit zwei `--` Zeichen und gehen bis zum Ende der Zeile.

Konstanten : legen einmalig Werte innerhalb eines `package`, einer `entity` oder einer `architecture` fest.

Syntax: `constant` identifier: type [range_expr][:= expression];

Beispiel:
```
constant Vcc:   real       := 4.5;
constant CYCLE: time       := 100 ns;
constant PI:    real       := 3.147592;
constant FIVE:  bit_vector := "0101";
```

Variablen : speichern Werte innerhalb eines `process` und werden dort, durch den Kontrollfluß gesteuert, sequentiell benutzt. Variablen können *nicht* benutzt werden, um Informationen zwischen Prozessen auszutauschen.

Syntax: `variable` identifier_list: type [range_expr][:= expression];

Bei der Deklaration können die Wertebereiche der Variablen eingeschränkt werden und die Initialisierung mit Werten ist möglich.

Beispiel:
```
variable INDEX:      integer range 1 to 50 := 10;
variable CYCLE_TIME: time range 10 ns to 50 ns := 10 ns;
variable REGISTER:   std_logic_vector (7 downto 0);
variable X, Y:       integer;
```

Signale : verbinden *Design-Entities* untereinander und übertragen Wertewechsel innerhalb der Schaltung. Die Kommunikation zwischen Prozessen findet über Signale statt.[4]

Syntax: `signal` identifier_list: type [range_expr][:= expression];

[4]Wegen der besonderen Bedeutung von Signalen in VHDL wird auf sie später noch genauer eingegangen in Abschnitt 6.

Bei der Deklaration können die Wertebereiche der Signale eingeschränkt werden und die Initialisierung mit Werten ist möglich.

Beispiel:
```
signal COUNT:     integer range 1 to 50;
signal GROUND:    bit := '0';
signal INT_BUS:   std_logic_vector (1 to 8);
```

Achtung:
Signale können *nicht* innerhalb eines Prozesses deklariert werden. Sie können zwar innerhalb des `process` benutzt werden, aber Signalzuweisungen werden in der Simulationszeit abgearbeitet. Das heißt, daß Signalzuweisungen *nicht* in der sequentiellen Reihenfolge im Prozess wirksam werden, sondern erst, wenn der Prozess ein `wait`-Statement erreicht.

Um den zeitlichen Charakter der Signalzuweisung hervorzuheben, wird auch ein anderer Zuweisungsoperator als bei Variablen benutzt, für den Ablauf der Simulationszeit können außerdem Verzögerungen bei der Signalzuweisung modelliert werden:

```
signal xyz: bit;
...
xyz <= '1' after 5 ns;
```

Die Benutzung von Signalen im sequentiellen Fluß des Prozesses führt daher oft zu (unerwartet) fehlerhaften Ergebnissen. Deshalb empfiehlt es sich im sequentiellen Ablauf eines VHDL-Prozesses (mit *Schreib- und Leseoperationen*) mit *Variablen* zu arbeiten und den berechneten Wert dann vor dem nächsten `wait` an Signale weiterzugeben.

Abschließend noch zwei Anmerkungen zum Umgang mit Variablen und Signalen:

Initialisierung : werden Variable oder Signale bei der Deklaration nicht explizit initialisert, so werden sie bei der Simulation folgendermaßen vorbesetzt:

Aufzählungstypen	: der erste Wert der Aufzählungsliste
`integer`, `real`	: der niedrigste darstellbare Wert

Dies wird gerade bei den Aufzählungstypen oft ausgenutzt, beispielsweise indem man den Startzustand eines endlichen Automaten als ersten definiert oder bei dem Typ `std_logic` wo mit `'U'` (*nicht initialisiert*) begonnen wird.

Bereichseinschränkung : um Variable oder Signale für die Hardwaresynthese zu benutzen, sollte entsprechend der geplanten Bitbreite eine Bereichseinschränkung vorgenommen werden — dies ist gerade bei `integer`-Typen notwendig, da sonst 32-bit Datenpfade generiert werden.

Beispiel:
```
signal CNT100:    integer range 0 to 99;          unsigned 7-bit
signal ADDR_BUS:  std_logic_vector (7 to 0);      8-bit
```

4 Ausdrücke

Um Ausdrücke zu bilden, hat man in VHDL die in der Tabelle aufgeführten Operatoren zur Verfügung, die Operatorprioritäten steigen an. Gegebenenfalls muß die Reihenfolge der Auswertung durch explizite Klammerung festgelegt werden:

Operator	Funktion	Operanden Typ1	- Typ2
logische Operatoren			
`and`	$a \wedge b$	bit, bit_vector, boolean	- =
`or`	$a \vee b$	bit, bit_vector, boolean	- =
`nand`	$\neg(a \wedge b)$	bit, bit_vector, boolean	- =
`nor`	$\neg(a \vee b)$	bit, bit_vector, boolean	- =
`xor`	$a \neq b$	bit, bit_vector, boolean	- =
relationale Operatoren			
`=`	$a = b$	gleicher Typ	
`/=`	$a \neq b$	gleicher Typ	
`<`	$a < b$	gleicher Typ	
`<=`	$a \leq b$	gleicher Typ	
`>`	$a > b$	gleicher Typ	
`>=`	$a \geq b$	gleicher Typ	
arithmetische Operatoren - additiv			
`+`	$a + b$	integer, real	- =
`-`	$a - b$	integer, real	- =
`&`	$a\&b$	bit, bit_vector, character, string	- passend
arithmetische Operatoren - vorzeichen			
`+`	$+a$	integer, real	
`-`	$-a$	integer, real	
arithmetische Operatoren - multiplikativ			
`*`	$a * b$	integer, real	- =
`/`	a/b	integer, real	- =
`mod`	a div b	integer	- =
`rem`	a mod b	integer	- =
weitere Operatoren			
`**`	a^b	integer, real	- integer
`abs`	$\mid a \mid$	integer, real	
`not`	$\neg a$	bit, bit_vector, boolean	

Da VHDL streng typisiert ist, müssen zum Teil explizite Angaben des Typs, sowie Typkonvertierungen vorgenommen werden.

Qualifizierungen : erlauben die explizite Angabe eines Typs, dies ist beispielsweise notwendig, wenn keine eindeutige Zuordnung möglich ist.

Syntax: `type'(expression)`

Beispiel:

```
type MONTH is (APRIL, MAY, JUNE);
type NAMES is (APRIL, JUNE, JUDY);

... MONTH'(JUNE) ...                          für Monat
... NAMES'(JUNE) ...                          für Namen
```

Konvertierungen : sind auf Grund der Typbindungen teilweise notwendig. Für die Standardtypen sind Konvertierungsfunktionen vordefiniert, bei eigenen Typen müssen sie bei Bedarf durch den Benutzer angegeben werden.[5]

Beispiel:

```
type FOURVAL is ('X', 'L,', 'H', 'Z');        vierwertige Logik, die erste
type VALUE4 is  ('X', '0,', '1', 'Z');        ..., die zweite
...
function CONVERT4VAL (S: FOURVAL) return VALUE4 is
begin                                         Konvertierungsfunktion
  case S is
    when 'X' => return 'X';
    when 'L' => return '0';
    when 'H' => return '1';
    when 'Z' => return 'Z';
  end case;
end COVERT4VAL;
...
process (ABC)                                 ...benutzt die Konvertierungsfunktion
  variable ABC: FOURVAL;
  variable XYZ: VALUE4;
...
  XYZ := CONVERT4VAL (ABC);
...
```

[5] Funktionen sind in Abschnitt 5.2, Seite 29 beschrieben.

IEEE 1164 – Std_Logic_Vector

Für den Datentyp `std_logic`, bzw. `std_logic_vector`, sind in extra Packages Operatoren auf diesen Typen definiert. Um dabei zwischen *unsigned* und *signed* (2-Komplement) Zahlendarstelungen zu unterscheiden – speziell für die Auswertung der Vergleichsoperatoren ist dies wichtig –, werden zwei Packages in der Bibliothek `IEEE` definiert, sie enthalten:

`std_logic_1164`

logisch	`and nand or nor xnor not`

`std_logic_unsigned` / `std_logic_signed`

relational	`= /= < <= > >=`
arithmetisch	`+ -`, `+ - abs`, `*`

Ausserdem befinden sich in diesen Packages noch weitere Operatoren für das shiften von Vektoren, sowie Konvertierungsfunktionen.

Beispiel:

```
library IEEE;                          Bibliothek benutzen
use IEEE.STD_LOGIC_1164.ALL;           Packages benutzen
use IEEE.STD_LOGIC_SIGNED.ALL;
...
  VARA := "1011";                      = -5
  VARB := "0011";                      = 3
  if (VARA > VARB) then                false
  ...
```

Ist die Zuordnung au einer Zahlendarstellung nicht eindeutig möglich, wie beispielsweise bei Literalen, so können die Subtypen (`unsigned`, `signed`) explizit angegeben werden.

Beispiel:

```
signed'("1011") > signed'("0011")      false
```

5 Sequentielle Beschreibungen

Die zentrale Rolle bei sequentiellen Beschreibungen spielt der Prozeß. Das `process`-Statement wird für die Verhaltensbeschreibung von Architekturen benutzt, und begrenzt einen Bereich, in dem Anweisungen sequentiell abgearbeitet werden.

Das `process`-Statement selber ist eine konkurrente Anweisung, d.h. es können beliebig viele Prozesse gleichzeitig aktiv sein, ihre Reihenfolge im VHDL-Code ist irrelevant.[6]

Syntax:
```
[proc_label:] process [(sensitivity_list)]
    [subprogram_decl|subprogram_body]
    [type_decl]
    [subtype_decl]
    [constant_decl]
    [variable_decl]
    [file_decl]
    [alias_decl]
    [attribute_decl]
    [attribute_spec]
    [use_clause]
begin
    [sequential_statements]
end process [proc_label];
```

Hier noch eine Anmerkung: das optionale Label (proc_label) ist bei der Simulation für das Debugging der Schaltung nützlich und sollte deshalb vergeben werden.

Das Beispiel simuliert die Auswahl des Minimums und Maximums aus drei Eingangswerten mit Hilfe von zwei Prozessen.

Beispiel:
```
entity LOW_HIGH is
  port (  A, B, C: in integer;                          Eingänge
          MI, MA: out integer);                         Ausgänge
end LOW_HIGH;

architecture BEHAV of LOW_HIGH is
begin
  L: process                                            Auswahl des Minimum
    variable LOW: integer := 0;
    begin
      wait on A, B, C;
      if A < B    then LOW := A;
                  else LOW := B;
      end if;
      if C < LOW  then LOW := C;
      end if;
```

[6] Weitere Erläuterungen zu der `process`-Anweisung und deren Abarbeitung in der Simulation finden sich in Abschnitt 7, Seite 41.

```
      MI <= LOW after 1 ns;
    end process;

  H: process                                       Auswahl des Maximum
    variable HIGH: integer := 0;
    begin
      wait on A, B, C;
      if A > B    then HIGH := A;
                  else HIGH := B;
      end if;
      if C > HIGH then HIGH := C;
      end if;
      MA <= HIGH after 1 ns;
    end process;
end BEHAV;
```

5.1 Anweisungen

Signalzuweisung : Die Signalzuweisung innerhalb von Prozessen wird in dem extra Abschnitt 6.2, Seite 35 behandelt, da sie zwar in der sequentiellen folge eines Prozesses steht, aber bei der Simulationsabarbeitung anders behandelt wird.

Variablenzuweisung : Bei der Zuweisung müssen die Typen zusammenpassen, gegebenenfalls müssen Attributierungen oden Konvertierungsfunktionen benutzt werden.

Syntax:
```
variable := expression;
```

If : Verzweigung wie in Programmiersprachen; einfache Decodierung bei Hardwareumsetzung.

Syntax:
```
if condition then
  sequential_statements
{elsif condition then
  sequential_statements}
[else
  sequential_statements]
end if;
```

Case : mehrfach-Verzweigung wie in Programmiersprachen; Decodierung komplexer Codes (z.B. auf Bussen) bei Hardwareumsetzung.

Syntax:
```
case expression is
  {when choices => sequential_statements}
end case;
```

Möglichkeiten für choices

value => sequential_statements	genau ein Wert
value1 | value2 ... => sequential_statements	Aufzählung von Werten
value1 to value2 => sequential_statements	Wertebereich
others => sequential_statements	Voreinstellung

Für expression müssen alle möglichen Werte aufgezählt werden oder die Anweisung muß `when others` als letzte Auswahl enthalten.

Beispiel: Decoder: BCD zu 7-Segment

```
case BCD is
  when "0000" => LED := "1111110";
  when "0001" => LED := "1100000";
  when "0010" => LED := "1011011";
  when "0011" => LED := "1110011";
  when "0100" => LED := "1100101";
  when "0101" => LED := "0110111";
  when "0110" => LED := "0111111";
  when "0111" => LED := "1100010";
  when "1000" => LED := "1111111";
  when "1001" => LED := "1110111";
  when others => LED := "-------";    don't care aus std_logic_1164
end case;
```

Loop : Modellierung verschiedener Schleifen, entsprechend in Programmiersprachen; Wiederholung von Elementen (z.B. entsprechend der bit-Breite) bei Hardwareumsetzung.

Syntax:

```
[loop_label:] while condition loop |                      mit Abfrage
[loop_label:] for identifier in value1 to value2 loop |   mit Zähler
[loop_label:] loop                                        beliebig oft...
  sequential_statements
end loop [loop_label];
```

Die Laufvariable der `for`-Schleife muß nicht extra deklariert werden; identifier gilt als lokale Variable in der Schleife, Zuweisungen sowie externer Zugriff sind nicht möglich.

Next : bewirkt den vorzeitigen Abbruch eines Schleifendurchlaufs, die zusätzliche Angabe einer Bedingung ist möglich.

Syntax: `next [loop_label][when condition];`

Beispiel:

```
for I in 0 to MAX_LIM loop
  if (DONE(I) = true) then next;      Sprung zu end loop
  end if;
  Q(I) <= A(I);
end loop;
```

```
L1: while J < 10 loop                          äußere Schleife
  L2: while K < 20 loop                        innere Schleife
    ...
    next L1 when J = K;                        Sprung aus innerer Schleife
    ...
  end loop L2;
end loop L1;                                   Sprungziel
```

Exit : beendet die Ausführung einer Schleife, die zusätzliche Angabe einer Bedingung ist möglich.

Syntax: `exit [loop_label][when condition];`

Beispiel:

```
for I in 0 to MAX_LIM loop
  if (Q(I) <= 0) then exit;                    Sprung aus Schleife
  end if;
  Q(I) <= (A * I);
end loop;
...                                            Sprungziel
```

Assert : ermöglicht die Überprüfung von Bedingungen zur Laufzeit des VHDL-Simulators. Dies ist bespielsweise sinvoll um Zeitbedigungen zu gewährleisten (set-up, hold ...), um Bereichsgrenzen zu prüfen u.s.w.

Syntax:

```
assert condition
  [report string_expr]
  [severity failure|error|warning|note];
```

Wenn condition nicht erfüllt ist, wird als Meldung string_expr ausgegeben.

Beispiel:

```
process (CLK, DIN)                             Verhaltensmodell eines D-FF
  variable X: integer;
  ...
begin
  ...
  assert (X > 3)
    report "setup violation"
    severity warning;
  ...
end process;
```

Wait : kontrolliert dynamisch die Ausführung/Unterprechung des Prozesses. Dadurch können in dem Verhaltensmodell — das der Prozess letztendlich darstellt — signalabhängige Aktivitäten modelliert werden. Außerdem werden erst durch ein `wait` neu berechnete Werte von Signalen in der Schaltung wirksam[7].

Syntax:
```
wait
   [on signal_name {, signal_name}]
   [until condition]
   [for time_expr];
```

Die sensitivity_list eines `process` entspricht einem `wait on ...` am Ende des Prozesses. Es gibt vier Grundformen der `wait`-Anweisung:

- `wait on A, B;` : Unterbrechen des Prozesses, bis ein Ereignis geschieht — hier: ein Wertewechsel auf einem der Signale `A` oder `B`.
- `wait until X > 10;` : Unterbrechen des Prozesses, bis eine Bedingung erfüllt ist — hier: das Signal `X` einen Wert `> 10` hat.
- `wait for 10 ns;` : Unterbrechen des Prozesses, bis eine Zeitspanne verstrichen ist — hier: `10 ns` simulierter Zeit.
- `wait;` : Unendlich langes Warten... Da ein VHDL-`process` immer *aktiv* ist, bietet diese Anweisung am Ende des Prozesses die einzige Möglichkeit ihn quasi anzuhalten — dies wird beispielsweise bei Initialisierungsprozessen benutzt, die nur einmal abgearbeitet werden sollen.

In dem Beispiel wird eine Architektur modelliert, die über zwei Prozesse eine Erzeuger/Verbraucher Situation simuliert. Über ein einfaches Handshake-Protokoll (zwei Leitungen, mit je zwei aktiven Zuständen) werden die Prozesse synchronisiert.

Beispiel:
```
entity PRO_CON is
  ...
end PRO_CON;

architecture BEHAV of PRO_CON is
  signal PROD: boolean := false;              item erzeugt Semaphor
  signal CONS: boolean := true;               item verbraucht Semaphor
begin
  PRODUCER: process                           Erzeuger
    begin
      PROD <= false;
      wait until CONS;
      ..produce item
      PROD <= true;
      wait until not CONS;
    end process;
```

[7] In den Abschnitten 6.2 und 1.4 (Seiten 36,7) wird die Wirkungsweise des `wait` genauer erklärt.

```
  CONSUMER: process                              Verbraucher
    begin
      CONS <= true;
      wait until PROD;
      CONS <= false;
      ..consume item
      wait until not PROD;
    end process;
end BEHAV;
```

5.2 Unterprogramme

An Unterprogrammen stehen in VHDL Prozeduren (mehrere Return-Werte via Parameter) und Funktionen (gibt einen Wert zurück) zur Verfügung. Funktionen werden beispielsweise zur Typkonvertierung oder als Auflösungsfunktion (siehe Abschnitt 6.3, Seite 36) benutzt.

In Unterprogrammen können zwar lokale Variablen deklariert werden, deren Werte sind aber nur bis zum Verlassen des Unterprogramms definiert — im Gegensatz zu Variablen im `process`, die einem lokalen Speicher entsprechen.

Die Deklaration von Unterprogrammen muß vor deren Aufruf erfolgt sein, das heißt bei Aufruf in einem `process`: innerhalb der entsprechenden `architecture`, des `entity` oder eines `package`.

Die Übergabe der Parameter kann beim Aufruf entweder über die Position oder über den Namen erfolgen, bei der Angabe des Namens gilt: declaration_name => actual_parameter

Function : hat mehrere Parameter und gibt genau einen Wert zurück — entspricht damit einem Ausdruck.

Syntax:
```
function func_name (parameter_list)
return type_name is
  [variable_declaration]
  [constant_declaration]
  [type_declaration]
  [use_clause]
begin
  [sequential_statements]
  return expression;
  [sequential_statements]
end [func_name];
```

In dem folgenden Beispiel wird ein Bitvektor in eine Integer Zahl umgerechnet.

Beispiel:
```
architecture ...
  ...
  function VEC2INT (S: bit_vector range 1 to 8)          Deklaration
  return integer is
    variable RES: integer := 0;                          lokale Variable
  begin
    for I in 1 to 8 loop
      RES := RES * 2;
      if S(I) = '1'  then RES := RES + 1;
      end if;
    end loop;
    return RES;
  end VEC2INT;
  ...
begin
  ...
  process ...
    ...
    XVAL := VEC2INT (XBUS);                              Aufruf
    ...
  end process;
  ...
end ...
```

Procedure : hat mehrere Parameter, die folgende Modi haben können:

- `in` : nur Eingangswert
- `out` : nur Ausgangswert, d.h. die Benutzung ist *nur* auf die linke Seite von Zuweisungen beschränkt.
- `inout` : Ein-/Ausgangswert, kann innerhalb der Prozedur universell benutzt werden.

Für die Parameter sind außer Variablen auch Signale (nach expliziter Deklaration) zulässig. Im Code werden Prozeduren wie Anweisungen behandelt.

Syntax:
```
procedure proc_name (parameter_list) is
  [variable_declaration]
  [constant_declaration]
  [type_declaration]
  [use_clause]
begin
  sequential_statements
end [proc_name];
```

parameter_list:
[variable] name_list [in|out|inout] type_name [:= expression]; |
signal name_list [in|out|inout] type_name;

Die Prozedur des Beispiels dient, wie die Funktion oben, der Umrechnung eines Bitvektors in eine Integer-Zahl wobei zusätzlich ein Flag gesetzt wird.

Beispiel:
```
architecture ...
  ...
  procedure VEC2INT                                    Deklaration
    ( S: in bit_vector;
      ZFLAG: out boolean;
      Q: inout integer ) is                            wegen Zuweisung
  begin
    Q := 0;
    ZFLAG := true;
    for I in 1 to 8 loop
      Q := Q * 2;                                      Q auf rechter Seite
      if S(I) = '1' then
        Q := Q + 1;                                    ⇒ Modus ist: inout
        ZFLAG := false;
      end if;
    end loop;
  end VEC2INT;

begin
  ...
  process ...
    ...
    VEC2INT (XBUS, XFLG, XVAL);                        Aufruf
    ...
  end process;
  ...
end ...
```

Overloading
Wie in einigen Programmiersprachen gibt es auch in VHDL Overloading Mechanismen für Funktionen und Prozeduren. *Overloading* bezeichnet dabei, daß Unterprogramme mehrfach definiert sind, aber unterschiedliche Typen, beziehungsweise unterschiedlich viele Parameter haben. Bei Aufruf dieser Unterprogramme wird dann, entsprechend Anzahl und Typ der Argumente, die entsprechende Funktion/Prozedur ausgewählt. Dadurch können, trotz starker Typfestlegung, Operatoren und Funktionen allgemein benutzt werden.

Argument-Typ : zwischen den Unterprogrammen wird durch den Typ der Argumente unterschieden.

Beispiel:

```
function DECR (X: integer) return integer is      integer Argument
begin
  ...
end DECR;

function DECR (X: real) return real is            real Argument
begin
  ...
end DECR;

...
variable A, B: integer;
...
B := DECR(A);                                     benutzt erste Funktion
```

Argument-Anzahl : zwischen der Unterprogrammen wird durch die Anzahl der Argumente unterschieden.

Beispiel:

```
function CONV_ADDR (A0, A1: bit) return integer is
begin                                             2 Argumente
  ...
end;

function CONV_ADDR (A0, A1, A2: bit) return integer is
begin                                             3 Argumente
  ...
end;

function CONV_ADDR (A0, A1, A2, A3: bit) return integer is
begin                                             4 Argumente
  ...
end;
```

Mit Overloading lassen sich auch die bestehenden Operatoren, aus dem Default-Package `STANDARD`, erweitern. Gerade bei herstellerspezifischen Packages werden diese Möglichkeiten in VHDL genutzt. Eine der häufigsten Ergänzungen sind Packages die mehrwertige Logik implementieren (STD_LOGIC_1164, MVL7, MVL9... – sie enthalten neben '0' und '1' auch Werte für 'X' (unknown), 'Z' (high-impedance) sowie verscheidene Treiberstärken). Dabei werden neben den Typen auch logische (and, or, not, xor...), arithmetische (+, −, ∗...) und Vergleichsoperatoren (=, / =, >, <...) auf diesen Typen definiert. Beispielsweise gibt es, für das am häufigsten genutzte Package STD_LOGIC_1164, noch zusätzliche Erweiterungen die eine unsigned- oder signed-Zahlendarstellung (2-Komplement) annehmen und die Operatoren entsprechend definieren. Durch das Overloading können diese dann vom Benutzer wie gewohnt gehandhabt werden. Für Funktionen mit zwei Argumenten ist dabei die Benutzung in *infix-Notation* möglich.

In dem Beispiel wird eine Addition für einen 4-bit `bit_vector` definiert.[8]

Beispiel:
```
function "+" (A, B: bit_vector (3 downto 0)) return bit_vector is
  variable SUM: bit_vector (3 downto 0);
  variable CARRY: bit;
begin
  CARRY := '0';
  for I in 0 to 3 loop
    SUM(I)  := A(I) xor B(I) xor CARRY;
    CARRY   := ((A(I) and B(I)) or (A(I) and CARRY) or (B(I) and CARRY));
  end loop;
  return SUM;
end;
```

[8]Standardmäßig ist die Addition auf `bit_vector` nicht möglich!

6 Signale

Während die VHDL-Elemente aus Verhaltensbeschreibungen (Prozesse, Variablen und sequentielle Anweisungen) ihre direkte Entsprechung in Programiersprachen haben, haben Signale und parallele Blöcke Eigenschaften, die typisch sind für Strukturbeschreibungen und deren Simulation.

In VHDL stellen Signale die einzige Möglichkeit dar, quasi als *Leitungen* die Elemente struktureller Beschreibungen miteinander zu verbinden sowie die Kommunikation zwischen Prozessen zu ermöglichen. Bei der Simulation wird eine zeitliche Ordnung von Ereignissen — im Sinne von Ursache und Wirkung — über Signale geregelt.

6.1 Deklaration

Signale können an folgenden Stellen im VHDL-Code deklariert werden:

1. innerhalb eines `package` für globale Signale.
2. als `port ...` der `entity` Deklaration für entity-globale Signale.
3. innerhalb einer `architecture` als architecture-globale Signale.

Syntax: in `package` oder `architecture`

```
signal signal_name: type_name [:= expression];
```

als `port` einer `entity`

```
... signal_name: in|out|inout|buffer type_name;
```

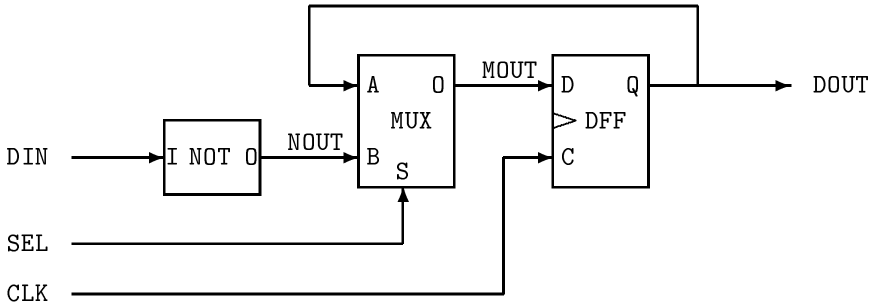
Bei der Deklaration von Ein-/Ausgängen einer `entity` muß eine Signalrichtung spezifiziert werden:

`in`	Eingang, nur auf rechter Seite von Variablen-/Signalzuweisungen zulässig.
`out`	Ausgang, nur auf linker Seite von Signalzuweisungen zulässig.
`inout`	bidirektionale Leitung, kann beliebig in Code benutzt werden.
`buffer`	prinzipiell ein Ausgang (nur ein Treiber), kann intern aber auch auf rechter Seite von Zuweisungen benutzt werden.

Beispiel:

```
package SIGDECL is
  signal VCC: std_logic := '1';                 globale Signale
  signal GND: std_logic := '0';
end SIGDECL;

entity MUXFF is                                 entity-globale Signale
  port (  DIN: in bit;
          SEL: in bit;
          CLK: in bit;
          DOUT: buffer bit);                    wird intern benutzt
  signal NOUT: bit;
end MUXFF;
...
```

```
architecture STRUCT of MUXFF is
  signal MOUT: bit;                          -- architecture-globale Signale
begin
  ...
```



6.2 Signalzuweisungen im Prozeß

Da Prozesse mit der *Außenwelt* (andere Prozesse, Entities...) nur über Signale verbunden werden können, werden Signalen (z.B. bei Ausgängen) Werte durch den Prozeß zugewiesen. Dabei sind allerdings einige Punkte zu beachten.

Verzögerungszeiten : Das Ziel der einer VHDL Beschreibung ist letztendlich die Simulation *realer* Schaltungen mit Verzögerungszeiten (Zeitkonstanten der elektrischen Netze). Zur Modellierung werden entsprechende Werte bei der Zuweisung an Signale angegeben. Für die Abarbeitung durch den Simulator heißt das, daß der neue Wert erst nach Ablauf der Verzögerungszeit auf dem Signal (für nachfolgende Eingänge) wirksam wird.

Syntax: signal_name <= expression [`after` time_expr {, expression `after` time_expr}];

Die Verzögerungszeit bei der Signalzuweisung (`after` ...) ist relativ zu dem aktuellen Wert simulierter Zeit bei Erreichen der Zuweisung; auch eine Verzögerungszeit von 0 ist zulässig. In einer Signalzuweisung können gleich mehrere Werte, als zeitliche Abfolge, zugewiesen werden. Diese zeitliche Folge wird dabei vom Simulationsalgorithmus in die Liste zukünftiger Ereignisse aufgenommen (*scheduling*). [9]

Beispiel:

```
R   <= "1010";
S   <= '1' after 4 ns, '0' after 7 ns;
T   <= 1 after 1 ns, 3 after 2 ns, 6 after 8 ns;
CLK <= not CLK after 50 ns;
```

[9]Der zeitliche Ablauf der Simulation ist in Abschnitt 1.4 genauer erläutert.

Aktivierung der Zuweisung : Obwohl Signalzuweisungen innerhalb eines Prozesses in einer Umgebung stehen, die in der Regel sequentiell abgearbeitet wird, werden solche Zuweisungen *nicht in der Reihenfolge der sequentiellen Anweisungen wirksam.* Signalzuweisungen werden erst bei Erreichen des nächsten `wait` in dem Prozeß oder, bei Verwendung einer *sensitivity-list*, am Ende des Prozesses wirksam. Daraus ergeben sich folgende Konsequenzen:

1. Signale können im Prozeß nicht wie Variable als Zwischenspeicher für Werte benutzt werden.
2. Es sollte in dem Prozeß nur eine einzige Zuweisung pro Signal vorkommen — nur *ein* Treiber.

Wegen der speziellen Eigenschaften der Signalzuweisung kommt es (gerade bei VHDL Anfängern) oft zu Fehlern, deshalb noch einige Beispiele:

<u>Beispiel:</u>	`X <= Y;`	beide Anweisungen werden bei dem `wait` gleichzeitig ausgeführt:
	`Y <= X;`	⇒ die Werte von `X` und `Y` werden vertauscht
	`wait ...`	⇒ die Reihenfolge der Zuweisungen ist irrelevant
	`V := 1;`	`V` wird `1` — sofort
	`S <= V;`	`S` wird `V` (also `1`) — bei `wait`
	`A := S;`	`A` erhält *alten* Wert von `S` — sofort
	`wait ...`	
	`X <= 1;`	Achtung: wird durch zweite Zuweisung unwirksam!
	`Y <= 3;`	`Y` wird `3` — bei `wait`
	`X <= 2;`	überschreibt obige Zuweisung: `X` wird `2` — bei `wait`
	`wait ...`	

6.3 Implizite Typauflösungen und Bustreiber

Alle bisher beschriebenen Signalzuweisungen gingen von einem Treiber pro Signal aus. Um Hardwarestrukturen wie (bidirektionale) Busse, wired-or, wired-and u.s.w. beschreiben zu können, müssen besondere Mechanismen (*resolution function*) eingeführt werden.

Standardmäßig ist die Behandlung von Signalen mit mehreren Treibern in VHDL nicht vorgesehen; der Grund dafür ist, daß sich, abhängig von der verwendeten Schaltungstechnik, entweder wired-or, wired-and oder anderes (logisches) Verhalten ergibt. Dementsprechend ist es den Herstellern überlassen, in entsprechenden Packages, Typdeklarationen für mehrwertige Logik und solche Auflösungsfunktionen zu liefern.

Sollen mehrere Treiber auf ein Signal wirken, muß man folgendermaßen vorgehen — mehrere Treiber heißt, daß dem Signal in konkurrenten Anweisungen (parallel ablaufenden Prozessen, konkurrenten Signalzuweisungen, konkurrenten Prozeduraufrufen u.s.w.) Werte zugewiesen werden:

Typ und Untertyp : Zu einem entsprechenden Datentyp wird ein Untertyp (`subtype`) deklariert, der mit einer Auflösungsfunktion verbunden ist.

Syntax: `subtype subtype_name is resolution_func_name type_name;`

Bei Signalzuweisungen auf ein Signal vom Typ `subtype_name` wird *implizit* die mit dem Typ verbundene Funktion aufgerufen und berechnet den Wert des Signals.

Analog zu dem `subtype` kann auch ein aufzulösendes Signal deklariert werden als: `signal_name : resolution_func_name type_name;`

Auflösungsfunktion : Die *resolution function* wird wie eine normale Funktion deklariert und hat folgende Eigenschaften

wie gewohnt gilt:
- Argumente haben den Modus *input* und werden als *pass by value* übergeben.
- Die Funktion liefert einen Wert zurück.

bei Auflösungsfunktionen gilt zusätzlich:
- Als Argument der Funktion wird ein Array variabler Länge mit Elementen `type_name` übergeben.
- Der Funktionswert ist vom *ursprünglichen* Typ: `type_name`.
- Die Funktion ist mit einem `subtype` verbunden. Bei jeder Signalzuweisung auf diesen Typ wird sie aufgerufen.

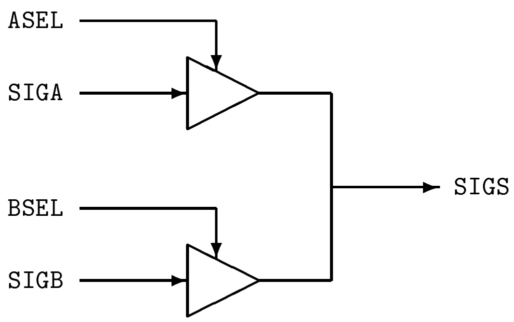
In dem Beispiel wird eine Auflösungsfunktion in Form eines *wired-or* auf einem 4-wertigen Datentyp beschrieben. Zwei Prozesse, die tristate-Treiber modellieren, benutzen ein gemeinsames Ausgangssignal.

Beispiel: 4-wertiger Typ und entsprechender Array-Typ für Funktion
```
type BIT4 is ('X', '0', '1', 'Z');
type BIT4_VECTOR is array (integer range <>) of BIT4;
```

Auflösungsfunktion
```
function WIRED_OR (INP: BIT4_VECTOR) return BIT4 is
  variable RESULT: BIT4 := '0';                 Ergebnis, Bus mit pull-down
begin
  for I in INP'range loop                       für alle Eingänge
    if INP(I) = '1' then
      RESULT := '1';
      exit;                                     'raus aus der Schleife
    elsif INP(I) = 'X' then
      RESULT := 'X';
    else null;                                  INP(I) = 'Z' oder = '0'
    end if;
  end loop;
  return RESULT;
end WIRED_OR;
```

Untertyp mit Auflösungsfunktion

```
subtype RESOLVED_BIT4 is WIRED_OR BIT4;

...
architecture BEHAVE of TRISTATE is
  signal ASEL, BSEL: boolean;                    Selektoren
  signal SIGA, SIGB: BIT4;                       Eingänge
  signal SIGS: RESOLVED_BIT4;                    Ausgangssignal
begin
  SOURCE1: process (ASEL, SIGA)                  erste Quelle
  begin
    SIGS <= 'Z';
    if (ASEL) then
      SIGS <= SIGA;
    end if;
  end process;

  SOURCE2: process (BSEL, SIGB)                  zweite Quelle
  begin
    SIGS <= 'Z';
    if (BSEL) then
      SIGS <= SIGB;
    end if;
  end process;
end BEHAVE;
```



Für den Datentyp `std_logic` sind in dem Package `STD_LOGIC_1164` Auflösungsfunktionen definiert: die Typen `std_logic` / `std_logic_vector` sind, mit einer Auflösungsfunktion versehene, Untertypen zu `std_ulogic` / `std_ulogic_vector` (**u**nresolved).

Als Beispiel für den Umgang mit bidirektionalen Bussen, wird nachfolgend ein 4-bit Bustreiber/-empfänger modelliert.

Beispiel:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity BUSIO is
  port( OEN:    in std_logic;
        IBUS:   in std_logic_vector (3 downto 0);
        OBUS:   out std_logic_vector (3 downto 0);
        IOBUS:  inout std_logic_vector (3 downto 0));
end BUSIO;

architecture BEHAV of BUSIO is
  begin
    P: process (OEN, IBUS, IOBUS)
    begin
      if (OEN = '1') then                          Bus treiben
        IOBUS <= IBUS;
      else                                         Bus nur lesen
        IOBUS <= "ZZZZ";                           explizite Zuweisung von 'Z'
      end if;
      OBUS <= IOBUS;
    end process;
  end BEHAV;
```

6.4 Attribute

Neben den typgebundenen Attributen gibt es in VHDL auch Attribute, die sich auf Signale beziehen. Mit Hilfe dieser Attribute können VHDL-Beschreibungen erstellt werden, die dynamisches Signalverhalten berücksichtigen. So können zur Laufzeit des Simulators über Attribute, Simulationsereignisse und Zeitpunkte ausgewertet werden.

Syntax:

aktueller Zeitpunkt	
`signal'event`	`true`, wenn *event* (Signaländerung)
`signal'active`	`true`, wenn *transaction* (Signalzuweisung)
vorheriger Zeitpunkt	
`signal'last_event`	Zeitdifferenz zur letzten Signaländerung
`signal'last_value`	Wert vor der letzten Signaländerung
`signal'last_active`	Zeitdifferenz zur letzten Signalzuweisung

Beispiel: D-Type FF

```
entity DFLOP is
  port (  CLK, D: in std_logic;
          Q: out std_logic)
end DFLOP;

architecture BEHAV of DFLOP is
begin
  process (CLK)
  begin
    if  (CLK = '1') and             -- CLK ist 1 und der Wert hat sich geändert
        (CLK'event) and             -- (kann wegfallen da CLK in sensitivity-list)
        (CLK'last_value = '0')      -- und der letzte Wert war 0 (wegen 'X'...)
      then  Q <= D;                 -- ⇒ Vorderflanke
    end if;
  end process;
end BEHAV;
```

7 Konkurrente Beschreibungen

Um die gleichzeitige Aktivität von Hardwarekomponenten auszudrücken, dienen konkurrente Beschreibungsformen.

Prozeß : Die wichtigste konkurrente Anweisung, der `process`, wurde schon als Umgebung sequentieller Anweisungen angesprochen,[10] seine Merkmale sind:

- alle Prozesse sind *parallel* aktiv.
- ein Prozeß definiert einen Bereich in dem Anweisungen (programmiersprachenähnlich) sequentiell ausgeführt werden, um Verhalten zu beschreiben.
- ein Prozeß muß entweder eine sensitivity-list oder explizite `wait`-Anweisungen beinhalten.
- innerhalb des Prozesses können Signale aus `entity` und `architecture` gelesen und verändert werden.

Prozeßabarbeitung
Da ein Prozeß in VHDL letztendlich das Verhalten eines Hardwareelementes modellieren soll, das ständig aktiv ist, hat ein Prozeß einige spezielle Eigenschaften:

Abarbeitung : Ein Prozeß kann als eine unbegrenzte Schleife verstanden werden. Bei Beginn der Simulation wird, quasi als Initialisierung, jeder Prozeß aktiviert und bis zu einem `wait` ausgeführt. Anschließend wird die Prozeßausführung entsprechend der Bedingung der `wait`-Anweisung unterbrochen.

Wird der Prozeß später durch Erfüllung der `wait`-Bedingung wieder aktiviert, werden die Anweisungen von dort ausgehend sequentiell weiter ausgeführt bis ein nächstes `wait` erreicht wird. Ist der Prozeßcode zu Ende (`end process;`), so *beginnt die Abarbeitung von vorn.* Man kann sich dies vorstellen als:

Beispiel:
```
process ...
begin
  loop;                          Start der Schleife
    ...
    wait ...                     mindestens ein wait, bzw. sensitivity-list
    ...
  end loop;                      Ende der Schleife
end process;
```

Aktivierung : Wie oben schon erläutert, wird ein Prozeß durch den Simulator sequentiell abgearbeitet, dann an einer oder mehreren Stellen unterbrochen und bei Eintreten bestimmter Ereignisse *event* wieder aktiviert.

Daraus ergibt sich, daß ein Prozeß mindestens eine `wait`-Anweisung oder eine *sensitivity-list* enthalten muß. Die sensitivity-list entspricht einem `wait on ...` am Ende des Prozesses.

[10] in Abschnitt 5, Seite 24

Beispiel: sensitivity-list

```
process (A, B)
begin
  ...
end process;
```

— ist äquivalent zu —

```
process
begin
  ...
  wait on A, B;
end process;
```

Soll ein Datenfluß beschrieben werden, so entspräche jede Operation einem Prozeß der jeweils nur eine einzige Anweisung enthält. Als vereinfachte Form stehen dafür konkurrente Anweisungen zur Verfügung. Diese Anweisungen stehen innerhalb der `architecture` und entsprechen jede einem eigenen Prozeß. Ihre Reihenfolge im VHDL-Code ist irrelevant.

konkurrente Signalzuweisungen : sind zu einem Prozeß äquivalent, der nur aus einer Signalzuweisung mit entsprechender sensitivity-list besteht.

Syntax: [label:] signal_name <= expression [after time_expr];

Die Anweisung wird aktiviert, wenn sich eines der Signale ändert, das auf der rechten Seite der Zuweisung steht.

Beispiel:

```
architecture VER1 of MUX is
begin
  OUTPUT <= A (INDEX);
end VER1;
```

— ist äquivalent zu —

```
architecture VER1 of MUX is
begin
  process (A, INDEX)
  begin
    OUTPUT <= A (INDEX);
  end process;
end VER1;
```

geschützte Signalzuweisungen : sind eine besondere Form der konkurrenten Signalzuweisungen. Die Zuweisung wird nur dann durchgeführt, wenn ein Signal `GUARD` von Typ `boolean` den Wert `true` hat. Dieses Signal kann explizit vom Designer deklariert und benutzt worden sein, kommt aber üblicherweise als implizites Signal aus einem *geschützten Block* (siehe Seite 44).

Syntax: [label:] signal_name <= guarded expression [after time_expr];

bedingte Signalzuweisungen : sind zu einem Prozeß äquivalent, der nur aus einer `if`-Anweisung und Signalzuweisungen nach der Verzweigung besteht. Auch geschachtelte `if` ... `elsif` ... sind möglich.

Syntax:
```
[label:] signal_name <= expression when condition else
                        {expression when condition else}
                        expression;
```

Die Anweisung wird aktiviert, wenn sich eines der Signale ändert, das auf der rechten Seite der Zuweisung steht (in Ausdruck oder Bedingung).

Beispiel:
```
Z <=  A when (X > 3) else
      B when (X < 3) else
      C;
```

selektive Signalzuweisungen : sind zu einem Prozeß äquivalent, der nur aus einer `case`-Anweisung und Signalzuweisungen nach der Verzweigung besteht.

Syntax:
```
[label:] with expression select
  signal_name <=  expression when value{,
                  expression when value};
```

Die Anweisung wird aktiviert, wenn sich eines der Signale der Auswahlbedingung oder der Ausdrücke auf der rechten Seite der Zuweisung ändert.

Beispiel:
```
with MYSEL select
  Z <=  A when 15,
        B when 22,
        C when 28;
```

konkurrente Prozeduraufrufe : entsprechen einem Prozeß der nur einen Prozeduraufruf beinhaltet. Die Prozedur hat Signale als `in`, `out` und `inout` Parameter. Der Aufruf wird aktiviert, wenn sich eines der Parametersignale ändert.

Beispiel:
```
architecture ...
  procedure VEC2INT
    ( signal S: in bit_vector;
      signal ZFLAG: out boolean;
      signal Q: inout integer;) is
  ...
begin
  VEC2INT (BITVEC, FLAG, NUMBER);
  ...
```

— ist äquivalent zu —

```
architecture ...
  procedure VEC2INT ...                    Deklaration wie oben
  ...
begin
  process (BITVEC, NUMBER)
  begin
    VEC2INT (BITVEC, FLAG, NUMBER);
  end process;
  ...
```

Block : Um konkurrente Anweisungen (hierarchisch) zu gruppieren, kann der `block`-Befehl benutzt werden. Ein Block enthält Deklarationen von Datentypen, Signalen u.s.w. die nur lokal benötigt werden; der eigentliche Anweisungsteil besteht aus konkurrenten Anweisungen wie oben beschrieben.

Geschützte Blöcke beinhalten zusätzlich einen boole'schen Ausdruck guard_expression der ein *implizit* vorhandenes Signal `GUARD` (Typ `boolean`) treibt. Dieses Signal kann innerhalb des Blocks für die Kontrolle von Anweisungen genutzt werden — wegen des direkten Bezugs auf `GUARD` sind dies meist *geschützte Signalzuweisungen*.

Syntax:
```
label: block [(guard_expression)]
  [use_clause]
  [subprogram_decl subprogram_body]
  [type_decl]
  [subtype_decl]
  [constant_decl]
  [signal_decl]
  [component_decl]
begin
  [concurrent_statements]
end block [label];
```

8 Strukturbeschreibungen

Bei dem strukturellen Beschreibungsstil in VHDL werden die Komponenten einer Architektur und deren Verbindungen untereinander beschrieben. Dazu werden Komponenten deklariert und dann Instanzen dieser Komponenten erzeugt, wobei die verbindenden Signale auf die Anschlüsse abgebildet werden. Die Instanziierung einer Komponente wird vom Simulator wie eine konkurrente Anweisung behandelt.

Syntax: component declaration

```
component component_name
  [generic (generic_list: type_name [:= expression] {;
            generic_list: type_name [:= expression]} );]
  [port (   signal_list: in|out|inout|buffer type_name {;
            signal_list: in|out|inout|buffer type_name} );]
end component;
```

component instantiation

```
component_label: component_name port map (signal_mapping);
```

Die Abbildung der Signale an den Anschlüssen kann bei der Instanziierung entweder über die Position oder über den Namen erfolgen, bei der Angabe des Namens gilt wie üblich: declaration_name => signal_name.

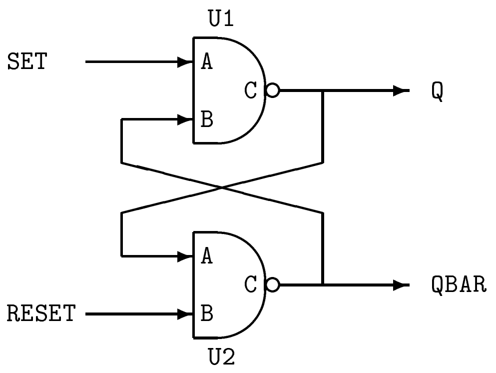
Wird an einen der Ports kein Signal angeschlossen (z.B. bei nicht benutzten Ausgängen), so kann der reservierte Bezeichner `open` benutzt werden. Anstelle der Signalnamen ist auch ein Funktionsaufruf möglich, dadurch können Typkonvertierungen direkt bei der Instanziierung von Komponenten vorgenommen werden.

Beispiel:

```
entity RSFF is
port (  SET, RESET: in bit;
        Q, QBAR: inout bit);
end RSFF;

architecture NETLIST of RSFF is
  component NAND2
    port (A, B: in bit; C: out bit);
  end component;
begin
  U1: NAND2 port map (SET, QBAR, Q);
  U2: NAND2 port map (Q, RESET, QBAR);
end NETLIST;
```

— bei Instanziierung mit Signalabbildung über Namen: —

```
  U1: NAND2 port map (A => SET, C => Q, B => QBAR);
```



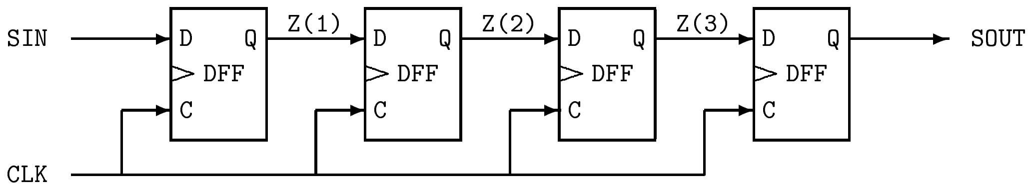
8.1 Erzeugung von Instanzen

Oft werden bei Hardwareaufbauten array-artige Anordnungen von Komponenten benötigt (z.B. entsprechend der Bitbreite, Größe des Speichers...). Um solche Anordnungen zu beschreiben, kann in VHDL die `generate`-Anweisung benutzt werden. Sie erlaubt:

1. die Wiederholung von Strukturen entsprechend einer `for ... loop` Schleife.
2. die Auswahl bestimmter Instanziierungen durch `if ... then` Bedingungen.

Syntax:

```
generate_label: for variable in range generate
  concurrent_statement                                i.A. Instanziierung
end generate [generate_label];

generate_label: if (condition) generate
  concurrent_statement
end generate [generate_label];
```



Beispiel:

```
entity SHIFT is
port (  SIN, CLK: in bit;
        SOUT: out bit);
end SHIFT;
```

gleichartiger Aufbau des Schieberegisters

```
architecture NETLIST1 of SHIFT is
  component DFF
    port (D, CLK: in bit; Q: out bit);
  end component;
  signal Z: bit_vector (0 to 4);
```

```
begin
  Z(0) <= SIN;
  GF: for I in 0 to 3 generate
    UI: DFF port map (Z(I), CLK, Z(I+1));
  end generate;
  SOUT <= Z(4);
end NETLIST1;
```

Ausnahmebehandlung von Ein- und Ausgang des Schieberegisters

```
architecture NETLIST2 of SHIFT is
  component DFF
    port (D, CLK: in bit; Q: out bit);
  end component;
  signal Z: bit_vector (1 to 3);
begin
  GF: for I in 0 to 3 generate
    GI1: if (I = 0) generate
      U0: DFF port map (SIN, CLK, Z(I+1));
    end generate;

    GI2: if ((I > 0) and (I < 3)) generate
      UI: DFF port map (Z(I), CLK, Z(I+1));
    end generate;

    GI3: if (I = 3) generate
      U3: DFF port map (Z(I), CLK, SOUT);
    end generate;
  end generate;
end NETLIST2;
```

8.2 Benutzung von Packages

Bei den obigen Beispielen wurde die Deklaration der Komponenten immer innerhalb der `architecture` gemacht. Bei sich dauernd wiederholenden Komponenten, zum Beispiel aus der Zellbibliothek eines Herstellers, kann man sich durch die Benutzung von Packages (die meist schon von den Herstellern mitgeliefert werden) diese Arbeit sparen.

Dazu wird mit Hilfe einer `use`-Anweisung das entsprechende Package für die Benutzung verfügbar gemacht. Das `package` selber enthält nur die Komponentendeklaration, die Entity-Deklaration und die zugehörige(n) Architektur(en) können sich in einer anderen Design-Unit befinden.

Beispiel:
```
architecture S of COMPARE is
  component XR2
    port (X, Y: in bit; Z: out bit);
  end component;
  component INV
    port (X: in bit; Z: out bit);
  end component;
  signal I: bit;
begin
  U0: XR2 port map (A, B, I);
  U1: INV port map (I, C);
end S;
```

— ist äquivalent zu —

```
package XYZ_GATES                  package mit Deklaration der Komponenten
  component XR2
    port (X, Y: in bit; Z: out bit);
  end component;
  component INV
    port (X: in bit; Z: out bit);
  end component;
end XYZ_GATES;

use WORK.XYZ_GATES.ALL;            Benutzung des package XYZ_GATES
architecture T of COMPARE is
  signal I: bit;
begin
  U0: XR2 port map (A, B, I);
  U1: INV port map (I, C);
end T;
```

8.3 Konfigurationen

Dadurch, daß zu einem `entity` verschiedene Realisierungen als `architecture` möglich sind, werden die folgenden Eigenschaften von VHDL erst ermöglicht:

- Schrittweise top-down Verfeinerung (von black-box Verhalten zu Struktur)
- Untersuchung von Alternativen
- Unterstützung von Versionen

Durch Konfigurationen werden den Komponenten in strukturellen Beschreibungen konkrete Architekturen zugeordnet. Diese Konfigurationen können an zwei Stellen vorgenommen werden:

1. innerhalb der `architecture` : in Form einer Konfigurationsanweisung.

Syntax: `for label: entity_name use entity [lib_name.]entity_name(architecture_name);`

Wurde keine explizite Konfiguration vorgenommen, so wird die jeweils (zeitlich) zuletzt analysierte `architecture` benutzt, die *null* Konfiguration.

<u>Beispiel:</u>

```
entity XR2 is                                    Deklaration des entity
  port (X, Y: in bit; Z: out bit);
end XR2;

architecture SLOW of XR2 is                      erste architecture
begin
  Z <= X xor Y after 1.0 ns;
end SLOW;

architecture FAST of XR2 is                      alternative architecture
begin
  Z <= X xor Y after 0.5 ns;
end FAST;

Benutzung von XR2 in COMPARE
architecture U of COMPARE is
  for U0: XR2 use entity WORK.XR2(FAST);         Konfiguration für XR2
  signal I: bit;
begin
  U0: XR2 port map (A, B, I);                    explizite Konfiguration
  U1: INV port map (I, C);                       implizite Konfiguration
end U;
```

2. außerhalb der `architecture` : können mit Hilfe der `configuration`-Anweisung Architekturen ausgewählt werden. Dabei ist eine Konfiguration eine seperate Design-Einheit, die analysiert und simuliert werden kann. Mit Konfigurationen lassen sich folgende Zuordnungen treffen:

Architecture — Entity

Wenn für ein `entity` mehrere Architekturen definiert sind, kann ausgewählt werden, welche `architecture` in der Simulation der Schaltung benutzt wird.

<u>Syntax:</u>

```
configuration configuration_name of entity_name is
  for architecture_name
  end for;
end configuration_name;
```

In dem Beispiel werden für das xor-Gatter zwei Konfigurationen erstellt, die dann als `ONE` und `TWO`, beispielsweise zur Simulation des XOR, benutzt werden können.

Beispiel:
```
configuration ONE of XR2 is              ONE wählt FAST aus
  for FAST
  end for;
end ONE;

configuration TWO of XR2 is              TWO wählt SLOW aus
  for SLOW
  end for;
end TWO;
```

Architecture — Component

In diesem Fall wird die Auswahl der Architekturen bei hierarchischen Beschreibungen festgelegt. Ist eine Schaltung strukturell (⇒ Hierarchie) beschrieben und sind für die instanziierten Komponenten mehrere Architekturen vorhanden, so kann bei der `configuration` festgelegt werden, welche der möglichen Architekturen für eine Instanz benutzt wird.

Syntax:
```
configuration configuration_name of entity_name is
  for architecture_name
    for label|others|all: comp_entity_name use
      entity [lib_name.]comp_entity_name(comp_architecture_name); |
      configuration [lib_name.]configuration_name;
    end for;
    ...
  end for;
  ...
end configuration_name;
```

In dem Beispiel sei `MCOMP` eine Schaltung die mehrere Instanzen von `XR2` und `INV` enthält.

Beispiel:
```
configuration TRY1 of MCOMP is
  for STRUCT
    for U0: XR2 use entity WORK.XR2(FAST);
    end for;

    — oder —
    for U0: XR2 use configuration WORK.ONE;
    end for;

    for others: XR2 use configuration WORK.TWO;
    end for;

    for all: INV use configuration WORK.INV(FAST);
    end for;
  end for;
end TRY1;
```

Achtung:
Bei den meisten VHDL-Simulatoren ist die Simulation eines Entity, das Komponenten instanziiert, nur über eine Konfiguration möglich. Diese ist auch dann notwendig, wenn für die Komponenten des (hierarchischen) Entwurfs nur *eine* Architektur vorhanden ist.
Üblicherweise ist dies bei Testumgebungen, die den eigentlichen Entwurf referenzieren, der Fall. Hier genügt dann die *default* Konfiguration.

Syntax:
```
configuration configuration_name of entity_name is
  for architecture_name
  end for;
end configuration_name;
```

Konfigurationen erlauben weiterhin eine neu-Abbildung der Anschlüsse der Komponenten-Deklaration (`component ...`) zu denen des zugrundeliegenden Entwurfs (`entity ...`). Üblicherweise verwendet man zwar für die Deklaration der Komponenten die entsprechenden Zeile (`port ...`) des `entity`, aber in einigen Fällen sind solche Umordnungen notwendig.

- Beispielsweise kann mit *generischen* Zellbibliotheken gearbeitet werden, die dann durch Konfigurationen auf Zielbibliotheken verschiedener Hersteller abgebildet werden.
- Außerdem können Elemente des Entwurfs, durch Spezialisierung anderer, ähnlicher Teile ersetzt werden. In dem Beispiel werden alle Inverter durch entsprechend beschaltete Nand-Gatter ersetzt.

Beispiel:
```
configuration NANDY of COMPARE is
  for S
    for all: INV use entity WORK.NAND2(BEHAVE)
      port map (A => A, B => Vcc, Z => Z);
    end for;
  end for;
end NANDY;
```

8.4 Parametrisierung durch generische Werte

Während VHDL-Entwürfe über Signale an den Ein- und Ausgängen im Sinne einer Struktur in der Umgebung eingebunden werden, kann deren Verhalten über generische Werte (quasi als Variablen) verändert werden.

Dieser Mechanismus wird beispielsweise dazu benutzt, Verzögerungszeiten von Elementen außerhalb des `entity` zu definieren. So kann ein Hersteller bei Änderung der Bibliotheken (z.B. Umstellung von 1.0 μ- auf 0.7 μ-Prozeß) seine Verhaltensmodelle beibehalten und nur die Schaltzeiten der einzelnen Elemente werden neu übergeben.

Generische Werte werden bei der Deklaration des `entity` vor den Ein-/Ausgängen (`port`) angegeben und können in der entsprechenden `architecture` wie Konstanten benutzt werden.[11] Die Übergabe, bzw. die Festlegung konkreter Werte kann an folgenden Stellen stattfingen:

1. default-Wert bei der `entity`-Deklaration
2. default-Wert bei der `component`-Deklaration in der `architecture` oder in einem `package`
3. aktueller Wert bei der Instanziierung in der `architecture`
4. aktueller Wert bei einer Konfiguration der `architecture`

Die Abbildung generischer Werte erfolgt jeweils über den Namen, als: declaration_name => actual_value.

Syntax: Deklaration in `entity` und `component`

```
generic ( generic_name : type_name [:= default_value]{;
          generic_name : type_name [:= default_value]} );
```

Instanziierung

```
component_label: component_name
  generic map (value_mapping)
  port map (signal_mapping);
```

Gerade bei Zellbibliotheken sind die generischen Werte, zusammen mit der Deklaration der Komponenten, in seperaten `packages` festgelegt. Weiterhin ist es möglich, generische Werte in Konfigurationen zu spezifizieren. So können verschiedene Konfigurationen benutzt werden, um unterschiedliche Geschwindigkeitsklassen (min-, typ-, max-delay) auszuwählen.

Beispiel:

```
entity XR2 is
  generic (M: time := 1.0 ns);          Verzögerungszeit als generischer Wert
  port (X, Y: in bit; Z: out bit);      Vorbesetzung mit Default
end XR2;
...
architecture GENERAL of XR2 is
begin
  Z <= X xor Y after m;                 Benutzung wie Konstante
end GENERAL;
```

[11] In den Beispielen werden generische Werte nur in strukturellen Beschreibungen, zur Angabe der Verzögerungszeit, eingesetzt. Sie können aber genauso auch in Verhaltensmodellen (`process`) wie ein konstanter Wert verwendet werden.

generischer Wert bei Instanziierung

```
architecture S of COMPARE is
  signal I: bit;
  component XR2
    generic (M: time);
    port (X, Y: in bit; Z: out bit);
  end component;
  ...
begin
  U0: XR2 generic map (M => 1.5 ns)
          port map (A, B, I);
  ...
end S;
```

generischer Wert bei `component`-Deklaration

```
architecture S of COMPARE is
  signal I: bit;
  component XR2
    generic (M: time := 1.5 ns);
    port (X, Y: in bit; Z: out bit);
  end component;
  ...
begin
  U0: XR2 port map (A, B, I);
  ...
end S;
```

generischer Wert bei `component`-Deklaration in extra `package`

```
package XYZ_COMPONENTS is
  component XR2
    generic (M: time := 1.5 ns);
    port (X, Y: in bit; Z: out bit);
  end component;
  ...
end XYZ_COMPONENTS;
```

generischer Wert in der `configuration`

```
configuration LATE of COMPARE is
  for S
    for U0: XR2 use entity WORK.XR2(GENERAL)
      generic map (M => 1.5 ns);
    end for;
  end for;
end LATE;
```

9 Libraries und Packages

Ein `package` ist eine eigene VHDL-Einheit, kann also compiliert werden und dient dazu, oft benutzte Deklarationen und Unterprogramme zu sammeln. Dies sind beispielsweise Deklarationen für: Typen, Untertypen
Konstante
Komponenten
Unterprogramme (Prozeduren und Funktionen)
...

Packages können (müssen aber nicht unbedingt) weiter in *header* und *body* unterteilt werden. Der *header* enthält dabei die nach außen sichtbaren Teile, während Implementationen in dem *body* stehen; beide Teile können in getrennten Dateien enthalten sein — die Idee ist auch hier wieder, daß bei Änderungen nur möglichst kleine Teile des VHDL-Codes ausgetauscht und neu analysiert werden müssen: hier der Package-*body*. Diese Unterteilung ist in folgenden zwei Fällen sinnvoll, bei Unterprogrammen sogar notwendig:

1. zurückgestellte (deferred) Konstante: Die Deklaration der Konstanten befindet sich im *header*, während die Festlegung eines Wertes im *body* stattfindet.
2. Unterprogramme — Funktionen und Prozeduren: Im Package-*header* ist nur die Deklaration des Unterprogramms, der Programmrumpf befindet sich im *body*.

Syntax:
```
package package_name is
   [type_decl]
   [subtype_decl]
   [constant_decl]
   [deferred_constant_decl]
   [subprogram_decl]
   [subprogram_header_decl]
   [component_decl]
end [package_name];

package body package_name is
   [deferred_constant_value]
   [subprogram_body]
end [package_name];
```

Um auf die Deklarationen aus Packages zuzugreifen, werden diese mit der `use` Anweisung in anderen Entwürfen benutzbar gemacht. Befinden sich diese Packages nicht in der Bibliothek `WORK` (Voreinstellung), so muß außerdem die Bibliothek mit der `library` Anweisung bekanntgegeben werden.

Syntax:
```
[library library_name_list;]                Voreinstellung für die Bibliothek ist WORK
use [library_name.]package_name.item_name; |
     [library_name.]package_name.all;
```

Nach obigen Deklarationen kann auf Elemente aus Bibliotheken und Packages direkt über deren Name `item_name` zugegriffen werden. Als zweite, allgemeine Möglichkeit kann auch der komplette Name angegeben werden: [library_name.]package_name.item_name.

Beispiel: Benutzung einer Konstantendeklaration

```
package MY_DEFS is
  constant UNIT_DELAY: time := 1 ns;
end MY_DEFS;

entity COMPARE is
  port (  A, B:in bit;
          C: out bit);
end COMPARE;
...
library DEMO_LIB;                                    wenn nicht WORK
use DEMO_LIB.MY_DEFS.all;
...
architecture DFLOW of COMPARE is
begin
  C <= not (A xor B) after UNIT_DELAY;
end DFLOW;
```

...als deferred constant

```
package MY_DEFS is
  UNIT_DELAY: time;                                  nur Deklaration
end MY_DEFS;

package body MY_DEFS is
  constant UNIT_DELAY: time := 1 ns;                 Festlegung des Wertes
end MY_DEFS;
```

Unterprogramm

```
package TEMPCONV is
  function C2F (C: real) return real;                nur Deklaration
  function F2C (F: real) return real;
end TEMPCONV;

package body TEMPCONV is
  function C2F (C: real) return real is              Funktionsrumpf
  begin
    return (C * 9.0 / 5.0) + 32.0;
  end C2F;

  function F2C (F: real) return real is
  ...
end TEMPCONV;
```

Zusätzliche Libraries und Packages werden im VHDL-Entwurf aus folgenden Gründen benutzt:

Zusammenfassung eigener Deklarationen : wie schon in den Beispielen gezeigt, können Deklarationen, die in mehreren (Sub-) Designs benötigt werden, in Form von Packages gesammelt werden — nur für einen Entwerfer bei Benutzung der Default-Library: `WORK` oder sogar für eine Arbeitsgruppe, Abteilung o.ä. bei zusätzlichen Bibliotheken.

Erweiterungen von VHDL : die Hersteller von VHDL-Werkzeugen bieten über eigene Bibliotheken und in ihnen enthaltene Packages Ergänzungen zu dem „Standard-VHDL" an. Dies sind meist zusätzliche Datentypen und Funktionen für:

mehrwertige Logik (`std_logic_1164`) und Operationen darauf.
mathematische Funktionen (Wurzel, Exponential-, trigonometrische- ...).
Hilfsroutinen (Zufallszahlengeneratoren, Queue-Modellierung...).
Konvertierungsfunktionen zwischen den einzelnen Datentypen.

Benutzung von Zellbibliotheken : die ASIC-Hersteller stellen ihre Zellbibliotheken für die Simulation von Strukturbeschreibungen, bzw. deren Synthese, in Form von VHDL-Libraries zur Verfügung.

Die Abbildung von Bibliotheken, wie sie in VHDL benutzt werden, auf das Dateisystem eines Rechners geschieht außerhalb der Sprache VHDL (meist in Konfigurationsdateien).

10 VHDL und Synthese

VHDL als Hardwarebeschreibungssprache deckt mit seinen Ausdrucksmöglichkeiten alle im (digital) IC-Entwurf verwendeten Abstraktionsebenen ab. So kann das Verhalten kompletter Systeme mit abstrakten Datentypen modelliert (und simuliert) werden, während andererseits Netzlistenbeschreibungen mit Datentypen `'0'` und `'1'` einen Aufbau aus Logikgattern nachbilden. Damit ist VHDL der ideale Ausgangspunkt für Synthesewerkzeuge, da Ein- und Ausgabe der Programme in VHDL erfolgen kann, wobei abhängig von deren Möglichkeiten eine mehr oder minder abstrakte VHDL-Beschreibung verarbeitet wird.

Synthese bezeichnet hier die Transformation von Verhaltensbeschreibungen in Strukturbeschreibungen aus Elementen niedrigerer Abstraktionsebenen — im Idealfall aus Elementen einer Zellbibliothek eines Herstellers. Da sich mit zunehmender Abstraktion mehr Freiheitsgrade in Bezug auf Entwurfsentscheidungen ergeben, wird der Suchraum für *mögliche* Lösungen immer größer. Aber selbst auf der Logikebene hat man schon den *tradeoff* zwischen Platz und Geschwindigkeit: kleine Lösungen mit wenig Elementen, die aber durch Mehrfachbenutzung der Gatter viele Stufen umfassen ⇔ schnelle Schaltnetze mit wenigen Stufen (im Idealfall: zweistufig), die aber sehr groß sind.

In kommerziellen Werkzeugen wird heutzutage die Logiksynthese (Schaltnetze), die Synthese von Datenpfaden (Register, Funktionseinheit, Register, ...) und die Synthese endlicher Automaten (synchrone Schaltwerke) relativ problemlos beherrscht. Bei funktionalen Beschreibungen höherer Ebenen zeigen Werkzeuge, die überwiegend aus dem universitären Bereich kommen, eine (notwendige) Spezialisierung — bezüglich der *Zielarchitekturen*, wie auch der Einsatzbereiche bzw. Aufgabenstellung ⇒ Einschränkung des Suchraums.

Wie schon angedeutet wurde, ist nicht jede VHDL-Beschreibung sinnvoller und/oder möglicher Ausgangspunkt für Synthesetools. Man spricht deshalb auch oft von *synthesis-ready code*. Das heißt:

1. Die Eingabe muß sich auf einem Level befinden, daß sie von den CAD-Werkzeugen verarbeitet werden kann — zur Zeit sind dies Beschreibungen, die im Bereich der Register-Transfer Ebene anzusiedeln sind. Gegebenenfalls müssen, im Sinne eines Top-Down Entwurfs, erste Verfeinerungen schon „von Hand“ gemacht worden sein.

2. Die Eingabe muß *gut synthetisierbar* sein. Hinter dieser Aussage verbirgt sich die ganze Kunst beim Umgang mit Synthesewerkzeugen. Wie in Programmiersprachen lassen sich auch in VHDL Verhaltensweisen auf viele verschiedene Arten ausdrücken, wobei *abhängig vom Werkzeug* einige besonders gut für die Synthese geeignet sind, während andere als „nicht synthethisierbar“ eingestuft werden oder nur zu ineffizienten Hardwarelösungen führen.

Neben der Art der Sprachbeschreibung wirken sich vor allem noch zwei weitere Einflüsse auf das Syntheseergebnis aus. Dies ist zum einen die Zielbibliothek – welche Elemente stellt ein Hersteller zur Verfügung –, zum anderen die Art und Weise, wie der Benutzer den Syntheseprozeß gesteuert hat. So bieten die meisten Systeme vielfältige Möglichkeiten um Randbedingungen einzustellen, wie:

- Schranken für die Fläche (*area*)
- Schranken für Signallaufzeiten (*delay, transition time, arrival time*)
- Zeitabhängigkeiten zwischen Signalen (*clock to data*)
- Belastung vorangeschalteter Stufen (*load*)
- Treiberleistung (*fanout*)

Durch die Angabe solcher Randbedingungen wird einerseits der Suchraum des Synthesesystems eingeschränkt, zum anderen können Anforderungen der späteren Schaltungsumgebung berücksichtigt werden ⇒ schneller, gezieltere Ergebnisse.

Da der „synthetisierbare“ Sprachumfang von VHDL von dem verwendeten Werkzeug abhängt, folgen jetzt nur einige allgemeine Anmerkungen zu Strategien bei der Synthese und zu der Umsetzung von Sprachkonstrukten.

Verzögerungszeiten : sind bei Signalzuweisungen, in Bezug auf die Synthese, unsinnig, da sich die absoluten Zeitangaben nicht in Hardware realisieren lassen.

Beispiel: `C <= A xor B after 5 ns;`

Datentypen : werden bei der Synthese folgendermaßen verarbeitet:

Bit, Bit_vector	: werden direkt umgesetzt
Std_logic, … _vector	: werden direkt umgesetzt
Aufzählungstypen	: werden automatisch durchnumeriert
Integer	: werden umgesetzt (meist in 2-Komplement Zahlen). Durch Bereichseinschränkungen wird die Wortbreite festgelegt. Beispielsweise impliziert die Angabe `range 0 to 15` eine 4-bit unsigned Darstellung.
Real	: abhängig vom Synthesesystem

Prozesse und konkurrente Signalzuweisungen : Die meisten Werkzeuge erzeugen bei der Synthese für jeden Prozeß und jede Signalzuweisung entweder ein eigenes Schaltnetz oder ein eigenes Schaltwerk. Die Unterscheidung ist abhängig von bestimmten Beschreibungsformen, die eine zeitliche Abhängigkeit implizieren.

Schaltnetze : werden unter folgenden Bedingungen erzeugt:

Prozeß	: beschreibt nur Ausdrücke
	: Variablen werden bei jeder Prozeßaktivierung initialisiert und dienen nicht als Zwischenspeicher
	: es kommt nur ein `wait`, bzw. eine sensitivity-list vor
Signalzuweisung	: das Ziel der Zuweisung kommt nicht auf der rechten Seite von Ausdrücken vor

Als Operatoren können alle logischen und arithmetischen Operationen ausgeführt werden. Bei den Anweisungen erzeugen `if` und `case` typischerweise Multiplexerstrukturen, Schleifen führen zu Wiederholungen von Gattern — sofern sie nicht zeitliches Verhalten im Sinne von Schaltwerken ausdrücken.

Schaltwerke : werden unter folgenden Bedingungen erzeugt:

Prozeß	: beschreibt ein zeitliches Verhalten
	: Signalzuweisungen sind von Bedingungen abhängig
	: Variablen beinhalten Werte vorheriger Prozeßaktivierungen
	: es werden typische, ereignisbezogene Signalabfragen durchgeführt, wie `if (CLK'event) and (CLK = '1')`
	: es kommen mehrere `wait` Anweisungen vor
Signalzuweisung	: das Ziel der Zuweisung steht auch auf der rechten Seite in einem Ausdruck

Um die zeitliche Zwischenspeicherung zu realisieren, werden bei der Synthese Flipflops, bzw. Register generiert. Die Operatoren und Anweisungen werden als Schaltnetze (s.o.) an den Ein- und Ausgängen der Zeitglieder realisiert.

endliche Automaten
Eine besonders häufig verwendete Form von Schaltwerken sind *endliche Automaten* (Finite State Machine), die dazu benutzt werden, als Kontrolleinheit andere Werke zu steuern.

Sie werden in VHDL typischerweise dadurch beschrieben, daß eine Variable (oder ein Signal mit Zwischenspeicherung) existiert, die den aktuellen Zustand speichert. In dem Prozeß wird mit Hilfe einer einzigen großen `case` Anweisung, abhängig vom aktuellen Zustand verzweigt; dabei werden dann innerhalb dieser Verzweigung aus der Eingabe ein Nachfolgezustand und die Ausgabewerte berechnet.

10.1 Synopsys

Da die Art der Beschreibung stark von den eingesetzten Werkzeugen abhängt – wie schon oben angedeutet: durch unterschiedliche Synthesestrategien werden unterschiedliche Formulierungen bevorzugt –, können Beispiele nur exemplarisch zu bestimmten Syntheseprogrammen und deren Versionen angegeben werden. Dieser Abschnitt bezieht sich auf die Synthese mit dem Synopsys Design Compiler, V 3.1a.

Art der VHDL-Beschreibung
Hinsichtlich der Abstraktion ist eine Synthese der Eingabe bis in der Bereich der Register-Transfer Ebene, teilweise sogar bis hin zur Hauptblockebene, möglich. Beispielsweise kann eine Strukturbeschreibung aus Elementen der RT-Ebene problemlos synthetisiert werden, wobei die einzelnen Komponenten durch Verhaltensbeschreibungen spezifiziert sind.

Bei der Wahl zwischen Verhaltens- oder Strukturbeschreibungen (einer niedrigeren Abstraktionsebene) gilt folgende Grundregel: Strukturen, die sich ein Entwerfer während des Designprozesses vorstellt, sollten auch als solche in die VHDL-Beschreibung eingebracht werden. Andernfalls ist es vergleichsweise schwierig eine komplexe Verhaltensbeschreibung so zu ändern, daß bestimmte „Zielarchitekturen“ durch die Synthese erzeugt werden.

10.1.1 Behandlung von Beschreibungsformen in der Synthese

Für die Beschreibungen in einer `architecture` gelten folgende Grundregeln:

Instanziierungen bilden die logische Grundlage von (vorgegebenen) Hierarchien, sie werden direkt in die synthetisierte Struktur übernommen. `generate`-Anweisungen werden entsprechend wie einfache Instanziierungen von Komponenten direkt in Hardwarestrukturen umgesetzt.

Beispiel: einfache Instanziierungen: $Z = \overline{\overline{A \wedge B} \wedge \overline{C \wedge D}}$

```
component ND2                                        Komponentendeklaration
  port (A, B: in bit; C: out bit);
end component;
...
signal TMP1, TMP2: bit;
...
begin
  U1: ND2 port map(A, B, TMP1);                      Instanziierungen
  U2: ND2 port map(C, D, TMP2);
  U3: ND2 port map(TMP1, TMP2, Z);
end ARCHI1;
```

Instanziierungen mit `generate`: 8-bit Register

```
component FF1                                        1-bit FF
  port (D, CLK: in bit; Q: out bit);
end component;
...
signal CLK:   bit;
signal D, Q:  bit_vector(0 to 7);
...
begin
  GEN: for I in D'range generate                     8-fache Instanziierung
    U: FF1 port map (D(I), CLK, Q(I));
  end generate GEN;
end ARCHI2;
```

Werden Entities in der Hierarchie mehrfach instanziiert, so sind bei deren Behandlung durch den Syntheseprozeß drei Strategien möglich:

Generalisierung : das instanziierte Entity wird einmal synthetisiert, anschließend verweist jede Instanziierung auf dieses Element (`Design Attribute: Don't Touch`).

Individualisierung : entsprechend der Anzahl der Instanziierungen werden *unterscheidbare* Entities erzeugt, die dann individuell synthetisiert werden (`Uniquify Hierarchy`).

Der Vorteil dieser Methode ist, daß unterschiedliche Umgebungen (des Entities) berücksichtigt werden können und deshalb der Syntheseprozeß für einzelne Instanzen auch unterschiedlich optimierte Ergebnisse liefert. Solche Unterschiede können hervorgerufen werden durch: Ausgangslasten ($\Rightarrow$ andere Dimensionierung von Treibern und Logik), Belegung von Eingängen mit konstanten Werten ($\Rightarrow$ Logikminimierung) usw.

Auflösen der Hierarchie : führt prinzipiell zu vergleichbaren Ergebnissen wie die Individualisierung, ist aber wegen des Verlusts der Übersichtlichkeit *nicht* zu empfehlen (`ungroup -all -flatten`).

Direkte Signalzuweisungen auf der Architekturebene (konkurrente, selektive und bedingte Signalzuweisungen) werden in entsprechende Schaltnetze umgesetzt.

Beispiel: konkurrente Signalzuweisung (mit Operation)

```
signal A, B, Z: bit;
...
begin
  Z <= A and B;
```

bedingte Signalzuweisung

```
signal A, B, C, Z: bit;
...
begin
  Z <=  A when ASSIGN_A = '1' else
        B when ASSIGN_B = '1' else
        C;
```

selektive Signalzuweisung (Multiplexer)

```
signal A, B, C, D, Z: bit;
signal CONTROL: bit_vector(1 down to 0);
...
begin
  with CONTROL select
    Z <=  A when "00"
          B when "01"
          C when "10"
          D when "11"
```

Konkurrente Prozeduraufrufe entsprechen in ihrer Wirkung Prozessen in VHDL (mit entsprechender *sensitivity list*) und werden bei der Synthese wie diese behandelt.

Prozesse können, abhängig von der Verwendung der enthaltenen Signalzuweisungen, sowohl Schaltnetze als auch Schaltwerke beschreiben. In dem jetzt folgenden Abschnitt wird auf die Synthese von Prozessen noch genauer eingegangen.

10.1.2 VHDL-Prozesse

Das entscheidende Kriterium, wie der VHDL-Code in Prozessen synthethisiert wird, bzw. ob überhaupt eine Synthese möglich ist, ist die Art der Zuweisungen an Variablen des Prozesses oder an die Signale, die den Prozeß mit seiner Umgebung in der `architecture` verbinden. Dabei wird unterschieden, ob bei der Abarbeitung eines Prozesses Zuweisungen *immer* oder nur *bedingungsabhängig* wirksam werden. Im letzteren Fall werden speichernde Elemente (Register) erzeugt — inferred devices.

1. Schaltnetze ⇔ keine impliziten Register

Signale : Für ein Signal wird nur sequentielle Logik synthetisiert, wenn dem Signal bei *jeder* Prozeßaktivierung Werte zugewiesen werden. Dementsprechend müssen folgende Bedingungen gelten: entweder ist die Signalzuweisung *nicht* bedingungsabhängig oder bei Verzweigungen im Programmlauf des Prozesses (`if` und `case`) findet in allen Fällen eine Zuweisung statt. Außerdem darf eine `if`-Anweisung keine Bedingung enthalten, die eine *Taktflanke* impliziert.

Beispiel: `if`-Anweisung

```
signal A, B, C, P1, P2, Z: bit;
...
P1: process
begin
  if (P1 = '1') then
    Z <= A;
  elsif (P2 = '0') then
    Z <= B;
  else
    Z <= C;
  end if;
  ...
```

`case`-Anweisung

```
signal INVAL is integer range 0 to 15;
signal Z1, Z2, Z3, Z4: bit;
...
P2: process
begin
  Z1 <= '0';    Z2 <= '0';
  Z3 <= '0';    Z4 <= '0';
  case INVAL is
    when 0 =>           Z1 <= '1';          0
    when 1 | 3 =>       Z2 <= '1';          1, 3
    when 4 to 7 | 2 =>  Z3 <= '1';          2, 4, 5, 6, 7
    when others =>      Z4 <= '1';          8...15
  end case;
  ...
```

Achtung:
Gerade bei *case*-Anweisungen wird oft vergessen, daß in allen Fällen eine Signalzuweisung vorkommen muß. Gibt es einen „Standard-Wert", so empfiehlt es sich erst eine unbedingte Zuweisung dieses Wertes vor der Verzweigung zu machen (s. obiges Beispiel). Dann brauchen in der Verzweigung nur die davon abweichenden „Spezialfälle" behandelt zu werden.

Beispiel: 1-aus-4 Decoder

```
signal DECIN is integer range 0 to 3;
signal D1, D2, D3, D4: bit;
...
P: process
begin
```

so nicht! — für `D1` bis `D4` werden FFs synthetisiert

```
  case DECIN is
    when 0 => D1 <= '1';
    when 1 => D2 <= '1';
    when 2 => D3 <= '1';
    when 3 => D4 <= '1';
  end case;
```

korrekte Beschreibung — lange Form

```
  case DECIN is
    when 0 => D1 <= '1';     D2 <= '0';
              D3 <= '0';     D4 <= '0';
    when 1 => D1 <= '0';     D2 <= '1';
              D3 <= '0';     D4 <= '0';
    when 2 => D1 <= '0';     D2 <= '0';
              D3 <= '1';     D4 <= '0';
    when 3 => D1 <= '0';     D2 <= '0';
              D3 <= '0';     D4 <= '1';
  end case;
```

korrekte Beschreibung — kurze Form

```
  D1 <= '0';                                default-Zuweisungen
  D2 <= '0';
  D3 <= '0';
  D4 <= '0';
  case DECIN is
    when 0 => D1 <= '1';                    davon abweichende Fälle
    when 1 => D2 <= '1';
    when 2 => D3 <= '1';
    when 3 => D4 <= '1';
  end case;
```

Variablen : Als Zwischenspeicher bei der sequentiellen Abarbeitung *einer Aktivierung* des Prozesses, entsprechen sie in ihrer Wirkung logischen Ausdrücken und werden auch bei der Synthese so behandelt. Entscheidend ist dabei, daß die Variablen *vor* jeder Leseoperation Werte zugewiesen bekommen.

Beispiel:

```
P: process
  variable TMP: bit;
begin
  ...                                   kein Lesezugriff auf TMP
  TMP := (A and B) or C;                gemeinsamer Teilausdruck
  ...
  Y <= D and TMP;
  Z <= (E or TMP) and F;
  ...
```

2. Schaltwerke ⇔ implizite Register werden eingefügt

Signale : Ist die Signalzuweisung an eine Bedingung der (sequentiellen) Prozeßabarbeitung gekoppelt, so werden speichernde Elemente als Register für Ausgangssignale erzeugt. Abhängig von der Art der Bedingung werden die unterschiedlichen Registertypen ausgewählt.

Zeitverhalten : Taktpegel- oder taktflankengesteuerte Elemente werden durch die Aktivierung im Prozeß unterschieden.

Bei flankengesteuerten Elementen können dabei sowohl `wait`-Anweisungen als auch die sensitivity list eines Prozesses zusammen mit einer `if`-Anweisung benutzt werden.

Schema:

```
process
begin
  wait until edge;
  ...
end process;

— oder —
process (sensitivity_list)
begin
  if edge then
    ...
  end if;
end process;
```

Dabei können für den Ausdruck edge folgende Bedingungen eingesetzt werden:

Schema: nach wait-Anweisung

```
sig_name = '1'                                   Vorderflanke
sig_name = '0'                                   Rückflanke
```

in if-Abfrage

```
sig_name'event and sig_name = '1'                Vorderflanke
not sig_name'stable and sig_name = '1'
sig_name'event and sig_name = '0'                Rückflanke
not sig_name'stable and sig_name = '0'
```

Kontrollverhalten : Sind mehrere steuernde Signale vorhanden, so bestimmt die Schachtelung der einzelnen Abfragen, ob es sich um synchron oder asynchron wirkende Leitungen handelt (s. Beispiele).

Variablen : Bei der Benutzung als Speicher für Werte *über mehrere Aktivierungen* des Prozesses hinweg, werden für Variablen implizite Register erzeugt. Dies ist immer dann der Fall, wenn es Pfade in der sequentiellen Abarbeitung Prozesses gibt, auf denen die Variablen als Operanden gelesen werden, ohne daß *vorher* Zuweisungen ausgeführt wurden. Hinsichtlich der Art der erzeugten Register gelten die oben bei den Signalen beschriebenen Regeln.

Die Unterscheidung, wann Signale und wann Variablen für implizite Register benutzt werden, ergibt sich im allgemeinen aus der Art der Beschreibungsform. Wird der gesamte Speicher außerhalb des Prozesses sichtbar, so sollten die dazugehörigen Signale direkt benutzt werden. Ist dies nicht der Fall, wie beispielsweise bei Schieberegistern (s. Beispiel: parametrisierbares Schieberegister), so müssen notwendigerweise Variablen benutzt werden.

Bei der Beschreibung impliziter Register sollte innerhalb eines Prozesses nur eine Art von speicherndem Element, bezüglich Zeit- und Kontrollverhalten beschrieben werden, da man so die bestmögliche Kontrolle über die Art der erzeugten Elemente hat. Nachfolgend sind die verschiedenen Registertypen exemplarisch vorgestellt:

Schema: Latch

```
                                LATCH_ENABLE: boolean
process (LATCH_ENABLE, D)
begin
  if LATCH_ENABLE then
    Q <= ...
  end if;
end process;
```

Latch mit asynchronem Set `SET, LATCH_ENABLE: boolean`

```
process (SET, LATCH_ENABLE, D)
begin
  if SET then
    Q <= '1';
  elsif LATCH_ENABLE then
    Q <= ...
  end if;
end process;
```

Flipflop `CLK: bit`

```
process (CLK)
begin
  if edge then
    Q <= ...
  end if;
end process;
```

Flipflop mit asynchronem Reset `RESET, CLK: bit`

```
process (RESET, CLK)
begin
  if (RESET = '1') then
    Q <= '0';
  elsif edge then
    Q <= ...
  end if;
end process;
```

Flipflop mit synchronem Reset `RESET, CLK: bit`

```
process (RESET, CLK)
begin
  if edge then
    if (RESET = '1') then
      Q <= '0';
    else
      Q <= ...
    end if;
  end if;
end process;
```

Durch generische Werte (`generic`) können Register beschreiben werden, die hinsichtlich der Bitbreite parametrisierbar sind.

Beispiel:

```
entity SREG_N is                                          -- n-bit Schieberegister
  generic (REG_LEN: integer := 16);                       -- Länge
  port( CLK:      in std_logic;
        SER_IN:   in std_logic;
        SER_OUT:  out std_logic);
end SREG_N;

architecture BEHAV of SREG_N is
begin
  process (CLK)
    variable VALUE: std_logic_vector(REG_LEN-1 downto 0);     -- Speicher
  begin
    if (CLK = '1') and CLK'event then
      VALUE := SER_IN & VALUE(REG_LEN-1 downto 1);            -- shift down
      SER_OUT <= VALUE(0);                                    -- Ausgangsleitung
    end if;
  end process;
end BEHAV;
```

Bei der Beschreibung von *flankengesteuerten* Registern gelten die folgenden Einschränkungen, um die Synthetisierbarkeit des VHDL-Codes zu gewährleisten:[12]

Lesezugriff auf Variablen : nach einer Zuweisung darf kein Lesezugriff auf die Variable mehr stattfinden. Diese Bedingung ergibt sich aus der (zeitlich) unterschiedlichen Wirksamkeit der Zuweisung an Variablen (⇒ sofort) und Signale (⇒ erst bei nächster `wait`-Anweisung).

Beispiel:

```
process (CLK)
  variable EDGE_VAR, ANY_VAR: BIT;
begin
  if (CLK'event and CLK = '1') then
    EDGE_VAR  := Y;                 -- Variablenzuweisung
    ANY_VAR   := EDGE_VAR;          -- erlaubt, da innerhalb der Bedingung
    ...
  end if;
  ANY_VAR := EDGE_VAR;              -- Fehler!
  ...
end process;
```

[12]**Anmerkung:** Die den Takt betreffenden Fälle modellieren Schaltungen die in einem *guten Design* ohnehin zu vermeiden sind!

nur einen Takt : die Benutzung mehrerer Ausdrücke für Taktflanken innerhalb eines Prozesses ist nicht erlaubt auch wenn diese unabhängig voneinander auf verschiedene Signale / Variablen wirken.

Die Auswahl mehrerer Takte für ein Ziel der Zuweisung – explizit durch Multiplexstrukturen oder implizit durch die Schachtelung von Bedingungen – ist nicht synthesefähig und kann höchstens durch externe Logik modelliert werden.

Beispiel: zwei in einem Prozeß beschriebene Flipflops

```
process(CLK_1, CLK_2)
begin
  if (CLK_1'event and CLK_1 = '1') then          erste Clock
    Q1 <= B;
  end if;
  ...
  if (CLK_2'event and CLK_2 = '1') then          Fehler!
    Q2 <= B;
  end if;
end process;
```

keine Ausdrücke : logische Verknüpfungen anderer Signale mit dem Takt, beispielsweise für Taktausblendungen, sind nicht zulässig. Sie müssen entweder durch externe Beschreibungen oder übergeordnete Bedingungen erreicht werden.

Beispiel:

```
if not(CLK'event and CLK = '1') then                     Fehler!

if (CLK_ENA = '1' and (CLK'event and CLK = '1')) then    Fehler!
```

keine Zuweisung bei negierter Flankenbedingung : die Zuweisung im `false`-Zweig einer durch den Takt bedingten Verzweigung ist nicht möglich, da dies eine Hardware voraussetzt, die das Fehlen einer Taktflanke erkennt.

Daraus ergibt sich auch, daß bei geschachtelten Bedingungen die Abfrage der Taktflanke die letzte Bedingung sein muß.

Beispiel:

```
process(CLK)
begin
  if (CLK'event and CLK = '1') then
    SIG <= B;
  else
    SIG <= C;                                    Fehler!
  end if;
end process;
```

10.1.3 Endliche Automaten

Für die VHDL-Modellierung endlicher Automaten gibt es die unterschiedlichsten Möglichkeiten, wovon hier exemplarisch eine Methode vorgestellt werden soll, die die beste Kontrolle über die Syntheseergebnisse verspricht.

Die Grundidee hierbei ist, daß innerhalb der beschreibenden Architektur die sequentielle Logik (δ- und λ-Schaltnetze $\Rightarrow$ Berechnung des Folgezustandes und der Ausgangsfunktion) und die Zeitglieder in den Rückkopplungspfaden (τ-Register $\Rightarrow$ Synchronisierung der Zustandsübergänge) in getrennten Prozessen beschrieben werden. Die Unterscheidung des Automatenmodells nach Mealy- und Moore-Modellierung betrifft dabei nur die Codierung des Prozesses der die Schaltnetze beschreibt.

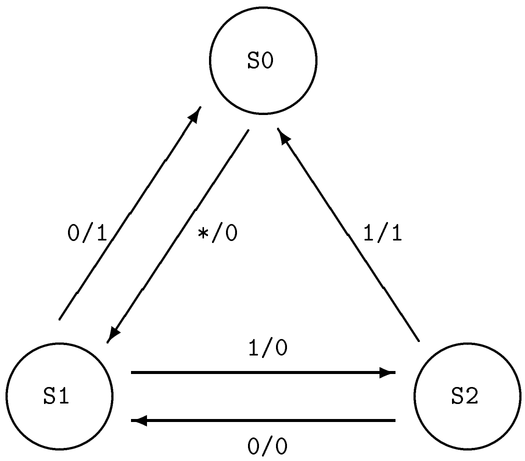
Beispiel:

```
entity MEALY is                                              Mealy-Automat
  port (X, CLK: in bit;
        Z:      out bit);
end;

architecture BEHAVIOR of MEALY is
  type STATE_TYPE is (S0, S1, S2);                           3-Zustände
  signal CURRENT_STATE:  STATE_TYPE;                         aktueller Zustand
  signal NEXT_STATE:     STATE_TYPE;                         Folgezustand
begin
  COMBIN: process(CURRENT_STATE, X)         Prozeß für kombinatorische Logik
  begin
    case CURRENT_STATE is                   Verzweigung durch CURRENT_STATE
    when S0 =>
      Z <= '0';                       Ausgabe '0'
      NEXT_STATE <= S1;               *-Übergang nach S1
    when S1 =>
      if X = '0' then
        Z <= '1';                     Ausgabe '1'
        NEXT_STATE <= S0;             '0'-Übergang nach S0
      else
        Z <= '0';                     Ausgabe '0'
        NEXT_STATE <= S2;             '1'-Übergang nach S2
      end if;
    when S2 =>
      if X = '0' then
        Z <= '0';                     Ausgabe '0'
        NEXT_STATE <= S1;             '0'-Übergang nach S1
      else
        Z <= '1';                     Ausgabe '1'
        NEXT_STATE <= S0;             '1'-Übergang nach S0
      end if;
    end case;
  end process;
```

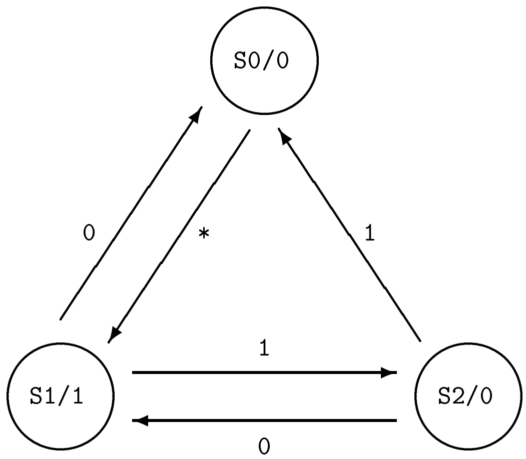
```
  SYNCH: process                              Prozeß für synchrone Elemente (Flipflops)
  begin
    wait until CLK'event and CLK = '1';       Synchronisation auf Vorderflanke
    CURRENT_STATE <= NEXT_STATE;
  end process;
end BEHAVIOR;
```

```
entity MOORE is                                                          Moore-Automat
  port (X, CLK: in bit;
        Z:      out bit);
end;

architecture BEHAVIOR of MOORE is
  type STATE_TYPE is (S0, S1, S2);                                           3-Zustände
  signal CURRENT_STATE:  STATE_TYPE;                                  aktueller Zustand
  signal NEXT_STATE:     STATE_TYPE;                                       Folgezustand
begin
  COMBIN: process(CURRENT_STATE, X)           Prozeß für kombinatorische Logik
  begin
    case CURRENT_STATE is                     Verzweigung durch CURRENT_STATE
    when S0 =>
      Z <= '0';                               Ausgabe '0'
      NEXT_STATE <= S1;                       *-Übergang nach S1
    when S1 =>
      Z <= '1';                               Ausgabe '1'
      if X = '0' then
        NEXT_STATE <= S0;                     '0'-Übergang nach S0
      else
        NEXT_STATE <= S2;                     '1'-Übergang nach S2
      end if;
    when S2 =>
      Z <= '0';                               Ausgabe '0'
      if X = '0' then
        NEXT_STATE <= S1;                     '0'-Übergang nach S1
      else
        NEXT_STATE <= S0;                     '1'-Übergang nach S0
      end if;
    end case;
  end process;
```

```
  SYNCH: process                              Prozeß für synchrone Elemente (Flipflops)
  begin
    wait until CLK'event and CLK = '1';       Synchronisation auf Vorderflanke
    CURRENT_STATE <= NEXT_STATE;
  end process;
end BEHAVIOR;
```



Mealy-Automat



Moore-Automat

10.1.4 Zusätzliche Information

Genauere Beschreibungen der Möglichkeiten und der Funktionsweise der SYNOPSYS Werkzeuge lassen sich am besten mit Hilfe der Online-Manuals nachlesen – Aufruf mit `iview` –, dabei sind folgende Kapitel in diesem Zusammenhang zu nennen:

```
Synthesis           VHDL Compiler Reference
                    Design Compiler Family Reference Manual
Application Notes   Flattening and Structuring: A Look at Optimization Strategies
                    Finite State Machines
                    HDL Coding Styles: Sequential Devices
```

Neben den hier aufgeführten Beispielen, finden sich umfangreiche VHDL Beispiele unter:

```
$SYNOPSYS/doc/vhdl/examples
$SYNOPSYS/doc/syn/examples/vhdl
```

A Syntaxbeschreibung

Anmerkung:
Der Anhang ist *keine* vollständige Syntaxbeschreibung von VHDL — hier sei auf das Reference Manual [IEEE94] verwiesen. Dieser Teil ist vielmehr als Hilfe für den, mit den Grundkonzepten der Sprache vertrauten Designer, gedacht: als Nachschlagehilfe bei Fragen zur Syntax und der Anordnung von VHDL-Konstrukten. Dementsprechend sind nicht alle Produktionen der Sprache dargestellt — die verbleibenden Bezeichner sollten aber für sich sprechen —, dafür wurde Wert darauf gelegt zu zeigen wo welche VHDL-Anweisungen im Code stehen können.

Der Index am Ende dieser Beschreibung ist dient als Cross-Index für die nachfolgenden Syntaxbeschreibungen. Dort sind Verweise für die gängigen Symbole und Bezeichner vorhanden — ständig benutzte Terminale wie `begin`, `end`, `is` usw. sind nicht aufgeführt.

A.1 Übersicht

Da der Aufbau einiger VHDL-Konstrukte aus den Syntaxbeschreibungen nicht klar erkenntlich ist, folgen hier noch einige Diagramme:

VHDL-Entwurf : Ein gesamter Entwurf besteht üblicherweise aus einer Anzahl von Dateien, die wiederum die analysierbaren Einheiten enthalten.

VHDL-Entwurf
- analysierbare VHDL-Datei
  - Entity -Interfacebeschreibung von Entwürfen
  - Architektur -Implementation von Entwürfen
  - Package -allgemeine Deklarationen
  - Konfiguration -Zuordnung von Architekturen zu Entity / Komponente / Instanz

Architekturen : beschreiben die Funktion eines Entity. Sie bestehen aus einem Teil für lokale Deklarationen und einem Anweisungsteil, der konkurrente Anweisungen beinhaltet. Diese können in beliebiger Reihenfolge im VHDL-Code stehen.

Architektur
- 1. Deklarationen
  - Typen
  - Konstanten
  - Signale
  - Unterprogramme
  - Komponenten
- 2. konkurrente Anweisungen
  - Blöcke -Gruppierung konkurrenter Anweisungen
  - Signalzuweisungen -Werteberechnung und -zuweisung
  - Prozeduraufrufe -Aufruf von Algorithmen, die Werte Signalen zuweisen
  - Komponenteninstanziierungen -Hierarchische Verbindung
  - Prozesse -Definition sequentieller Algorithmen, die Signalwerte verarbeiten und die Ergebnisse Signalen zuweisen

Packages : fassen Deklarationen zusammen, die an mehreren Stellen im Entwurf gebraucht werden (insbesondere in mehreren Dateien).

- Package
  - Deklarationen
    - Typen
    - Konstanten
    - Unterprogramme
    - Komponenten

Prozesse und Unterprogramme : dienen der Verhaltensbeschreibung durch einen sequentiellen Ablauf von Anweisungen.

Prozesse verarbeiten die Werte von Signalen und weisen ihnen neue Werte zu. Signalzuweisungen werden *außerhalb der sequentiellen Abarbeitung* wirksam. Die Synchronisation zwischen der Abarbeitung der Anweisungen und dem Verlauf an simulierter Zeit geschieht durch besondere Anweisungen (sensitivity-list, wait).

Unterprogramme strukturieren Verhaltensbeschreibungen und können nur innerhalb von sequentiellen Anweisungen benutzt werden. Signalein- und Ausgaben sind nur nach expliziter Deklaration im Interface möglich.

- Prozeß / Unterprogramm
  - 1. Deklarationen
    - Typen
    - Konstanten
    - Variablen
    - Unterprogramme
  - 2. sequentielle Anweisungen
    - Signalzuweisungen
    - Variablenzuweisungen
    - Prozeduraufrufe
    - Statements
      - Wait
      - If
      - Case
      - Loop
      - Next
      - Exit
      - Null
      - Return -nur bei Unterprogrammen-

A.2 Bibliothekseinheiten

entity_declaration

```
entity identifier is
   [generics]
   [ports]
   [entity_declarative_part]
[begin
   entity_statement_part]
end [identifier];
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

architecture_body

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

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

package_declaration

```
package identifier is
  package_declarative_part
end [identifier];
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

package_body

```
package body identifier is
  package_body_declarative_part
end [identifier];
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

configuration_declaration

```
configuration identifier of entity_name is
  configuration_declarative_part
  block_configuration
end [identifier];
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

A.3 Deklarationen

generic_declaration

```
generic (identifier_list : type [:= expression];
         {identifier_list : type [:= expression];} );
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

port_declaration

```
port (identifier_list : in|out|inout|buffer type [:= expression];
     {identifier_list : in|out|inout|buffer type [:= expression];} );
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

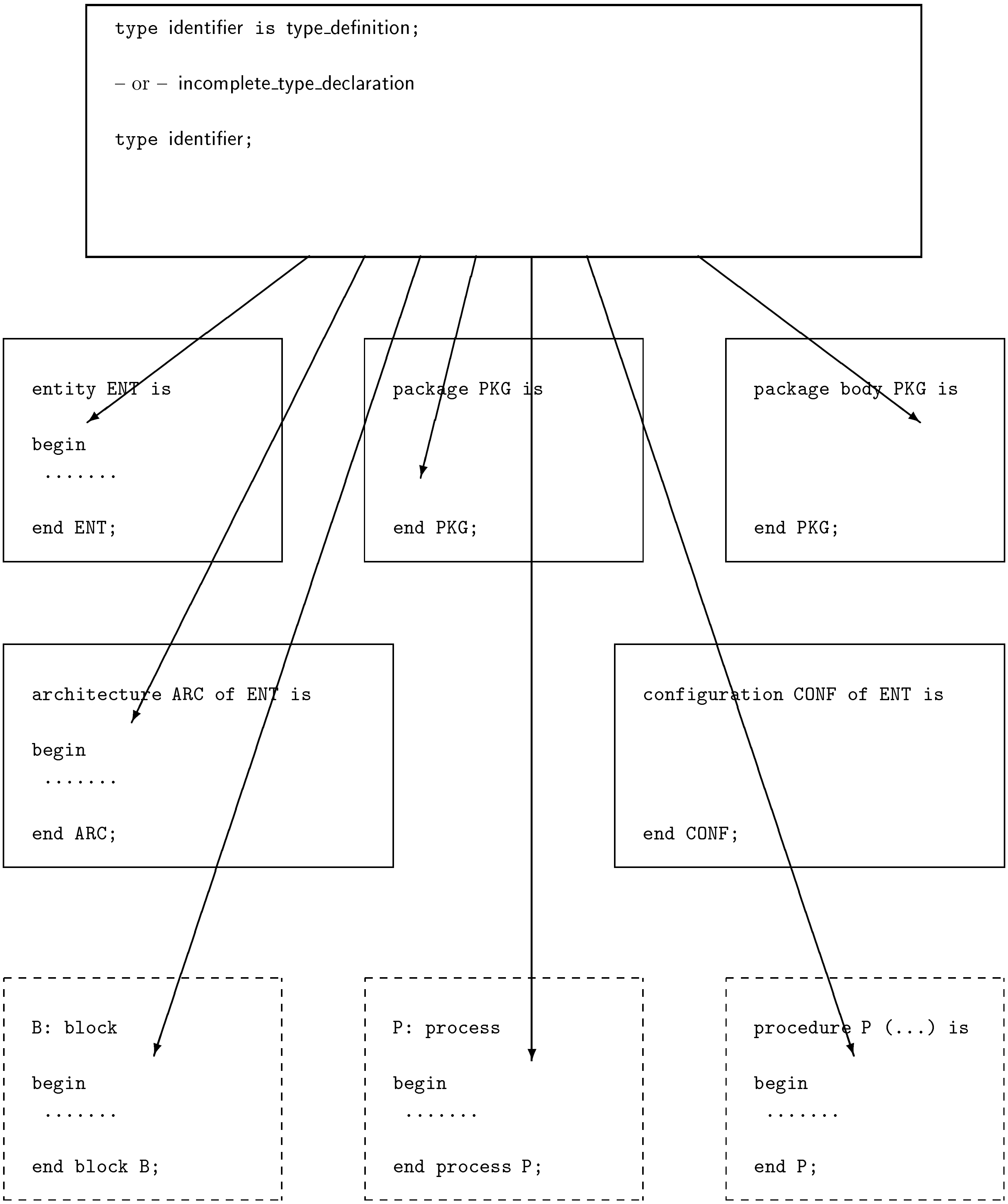
```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

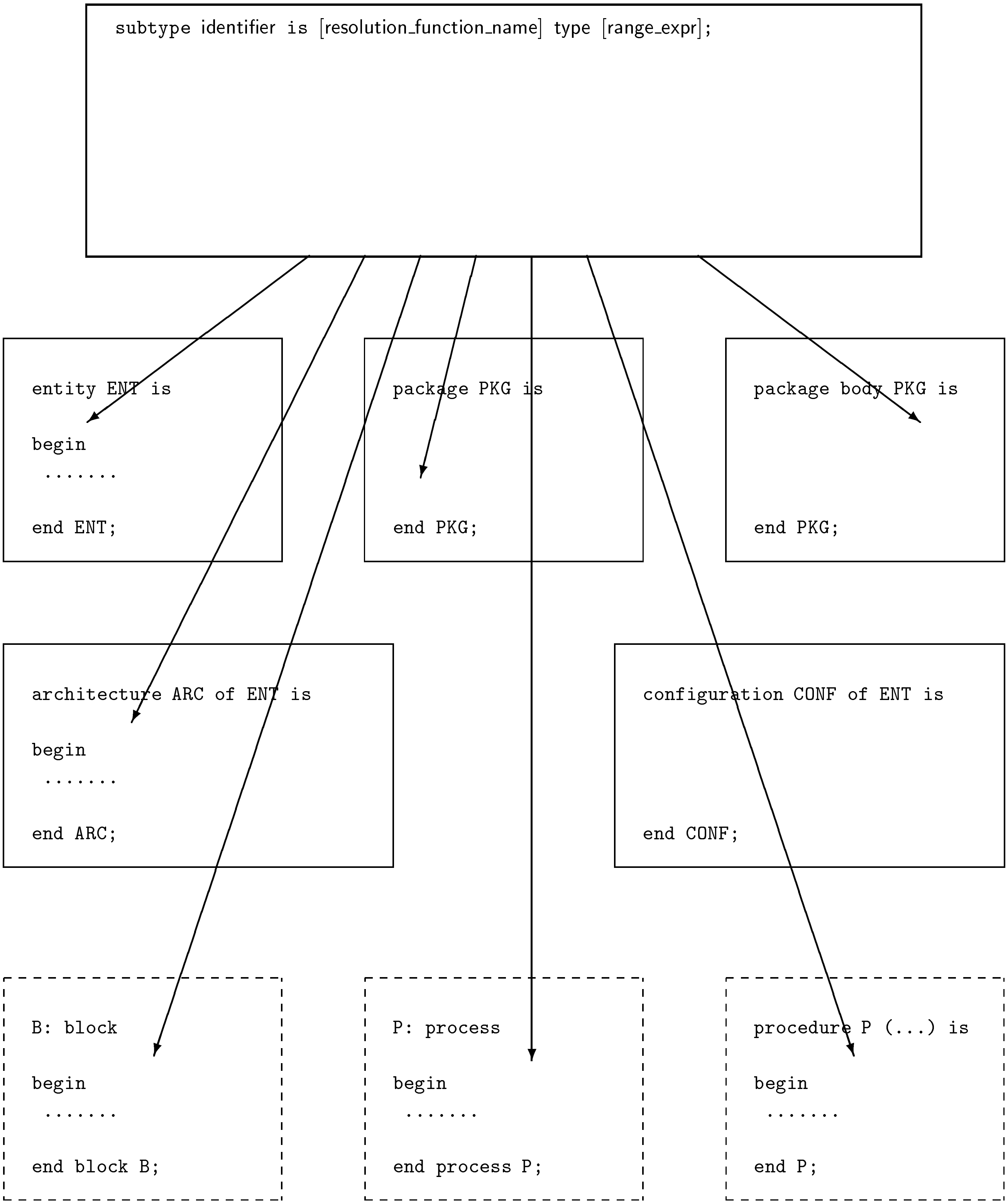
```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

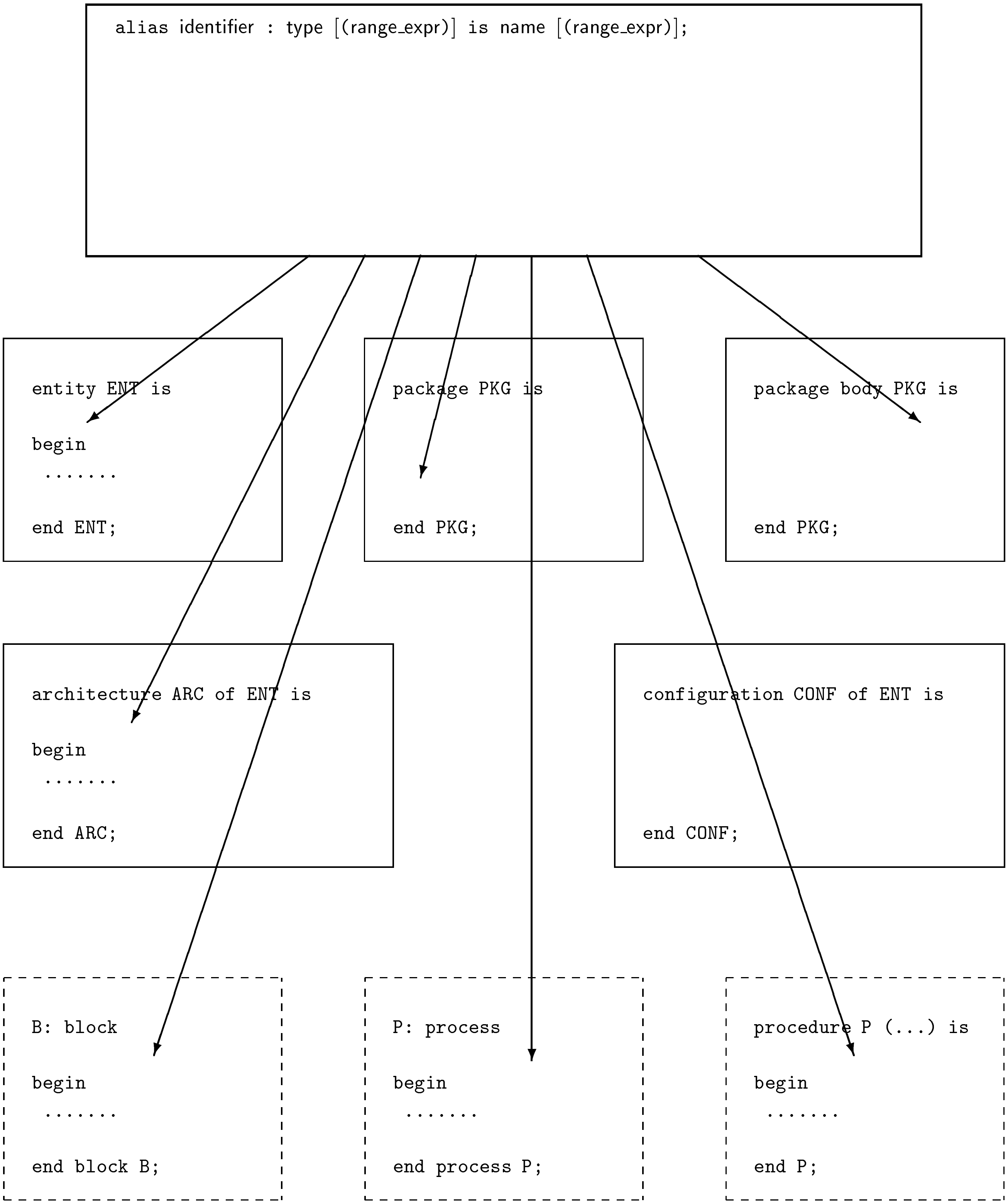
type_declaration



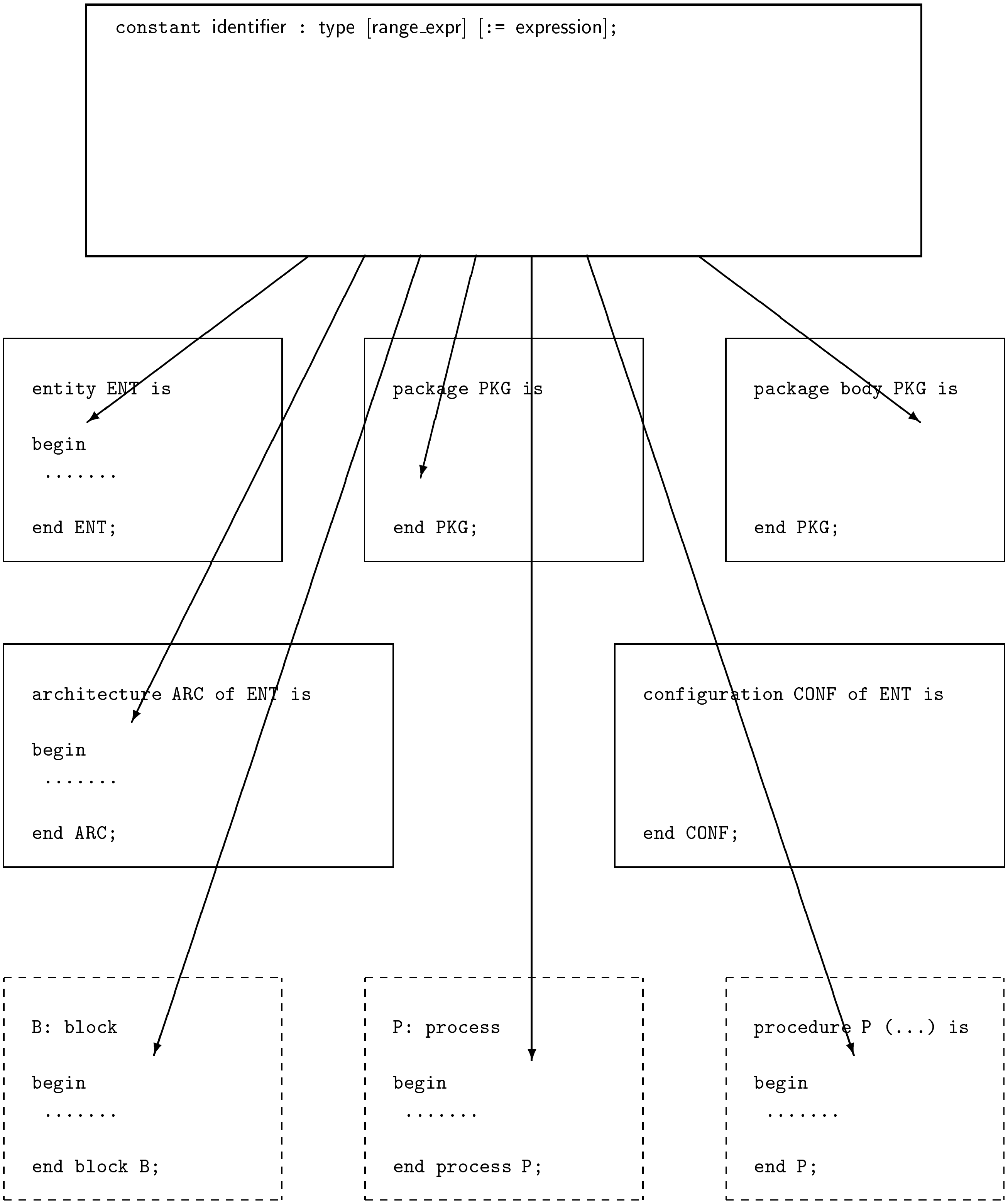
subtype_declaration



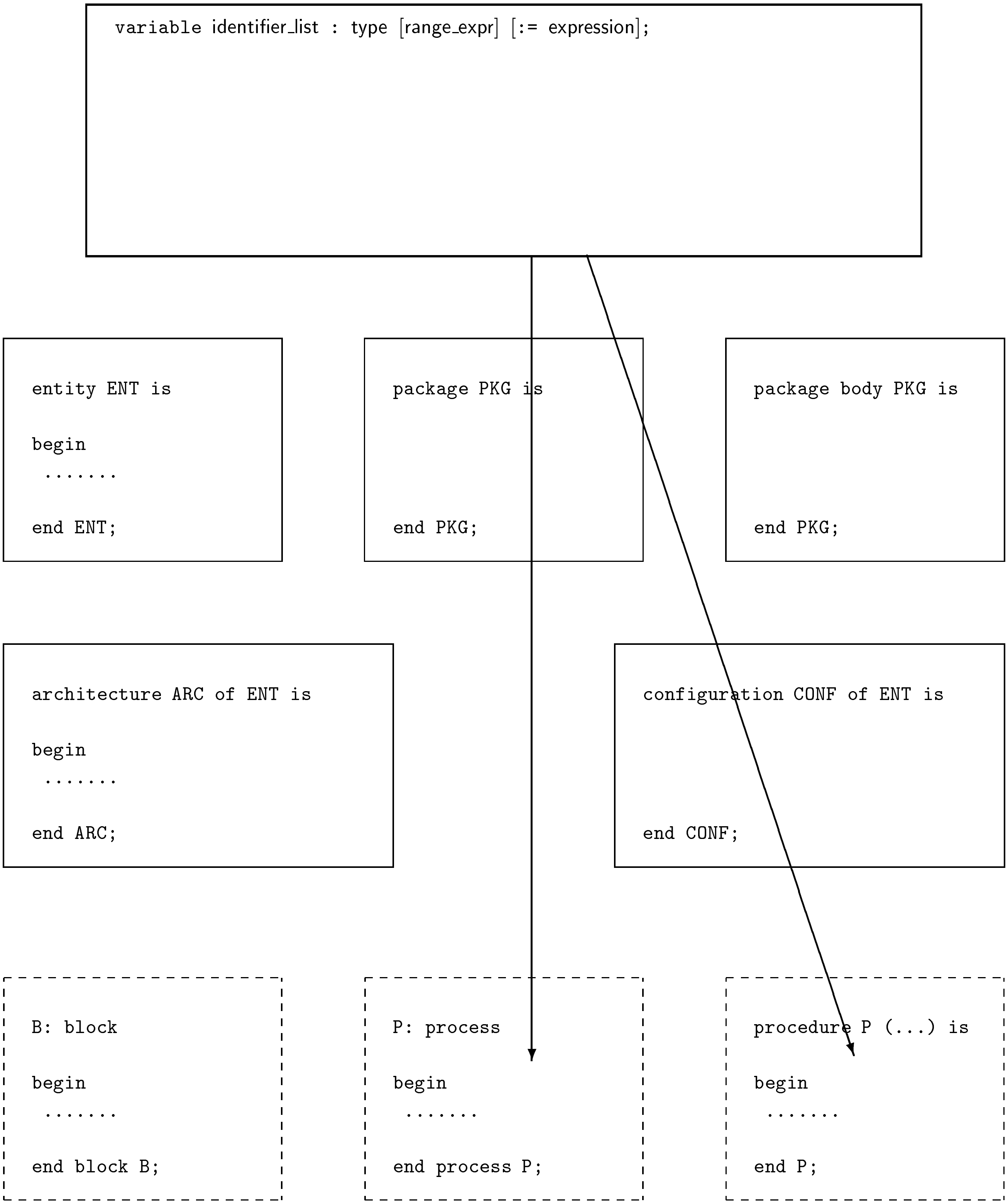
alias_declaration



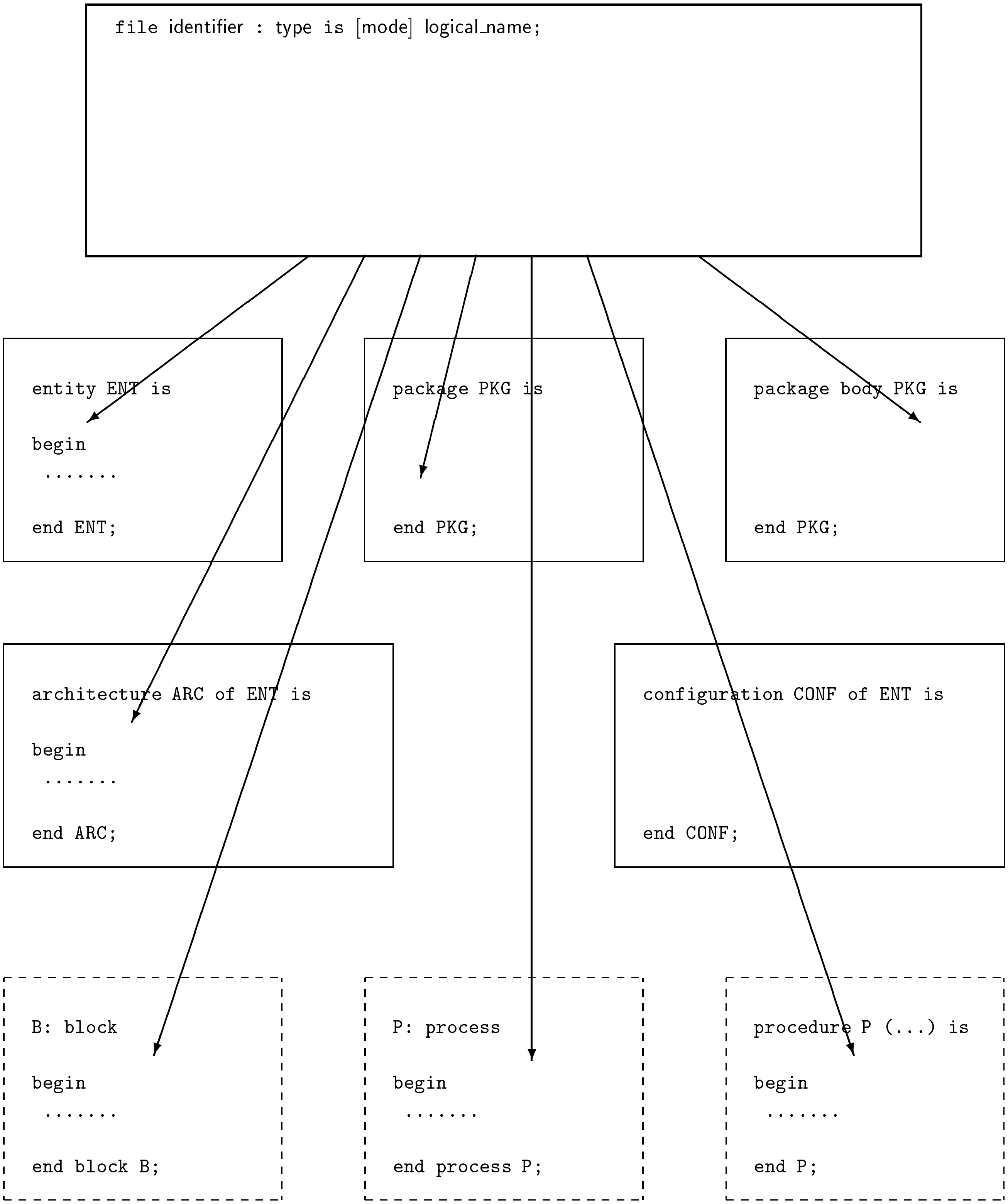
constant_declaration



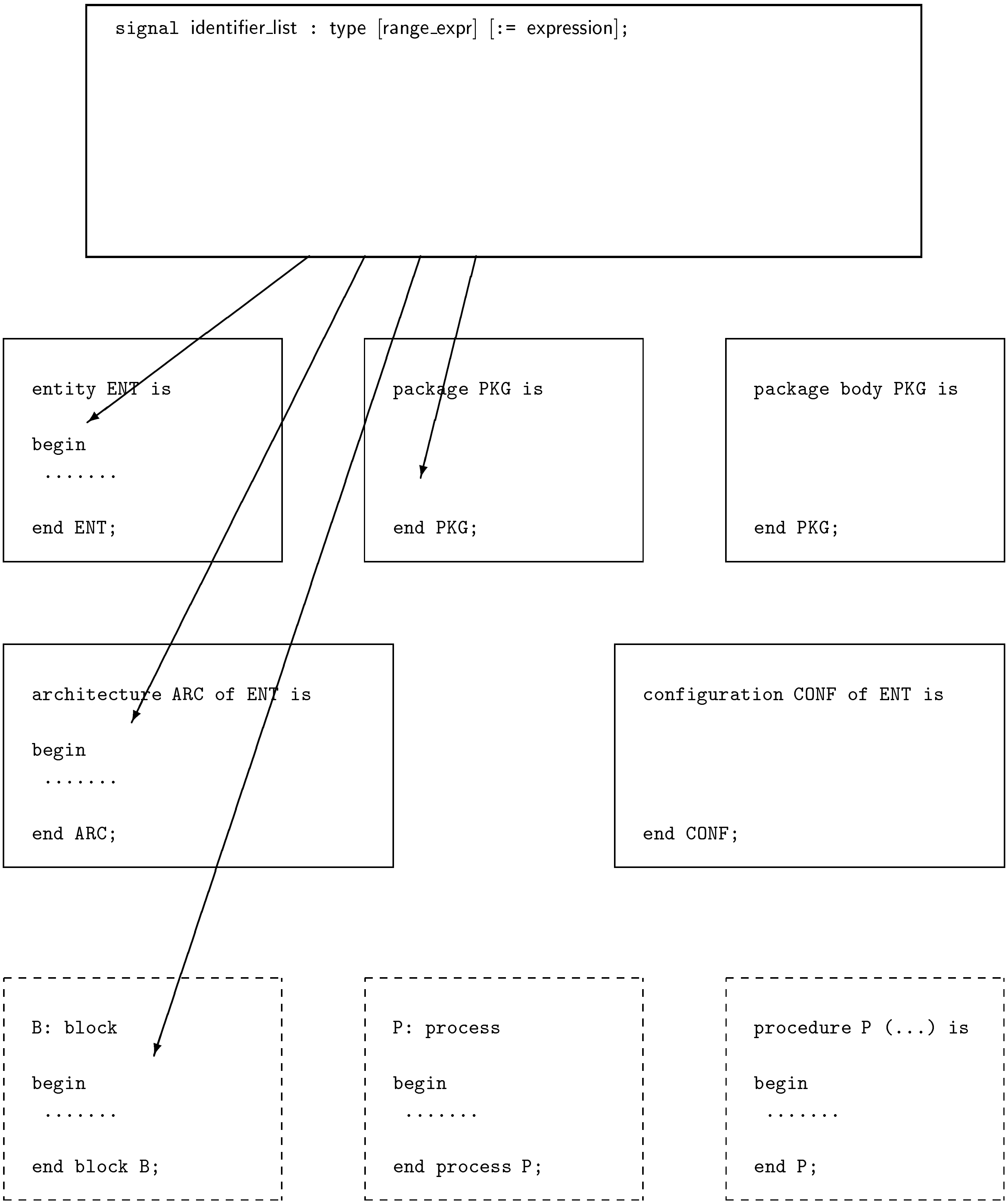
variable_declaration



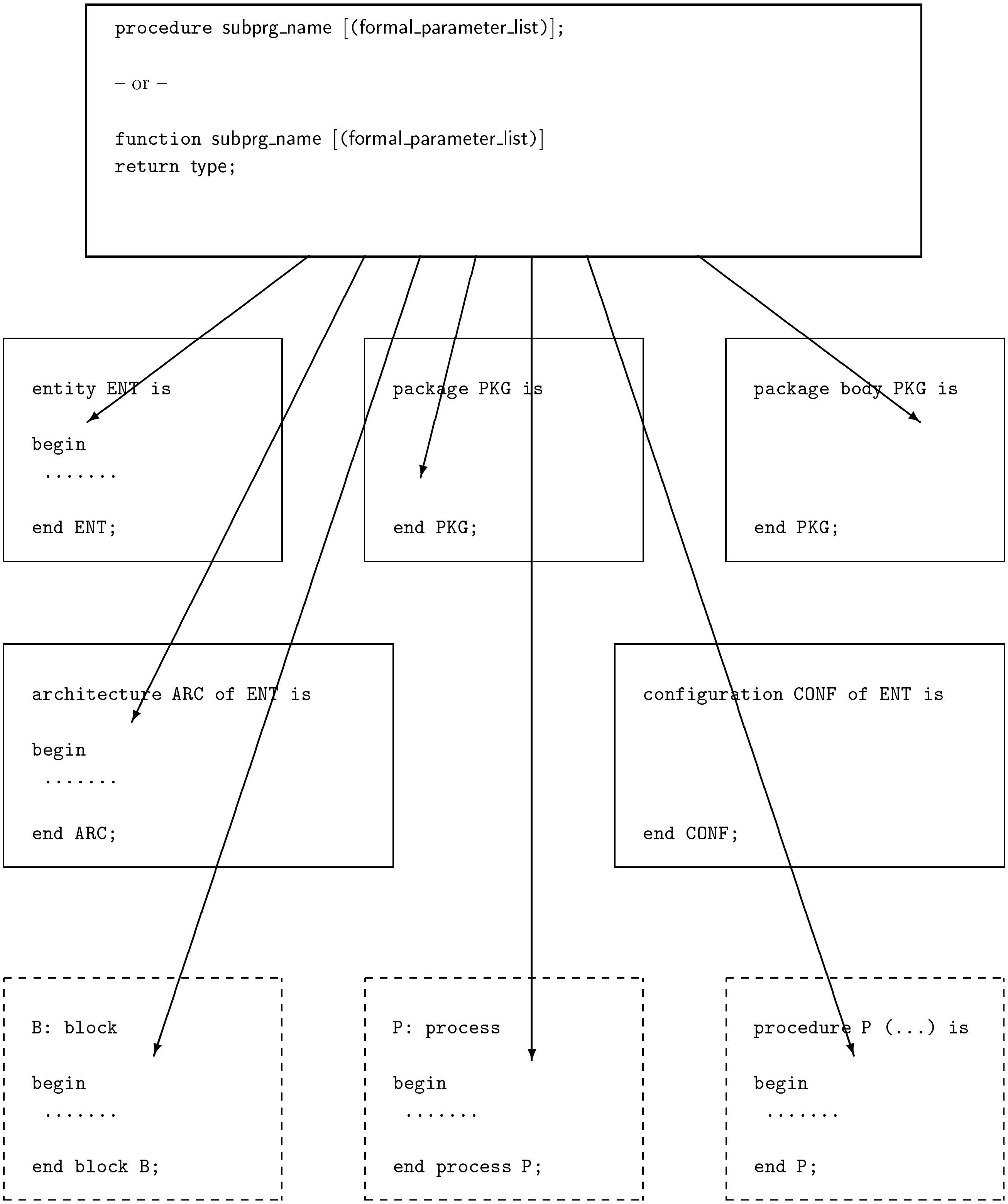
file_declaration



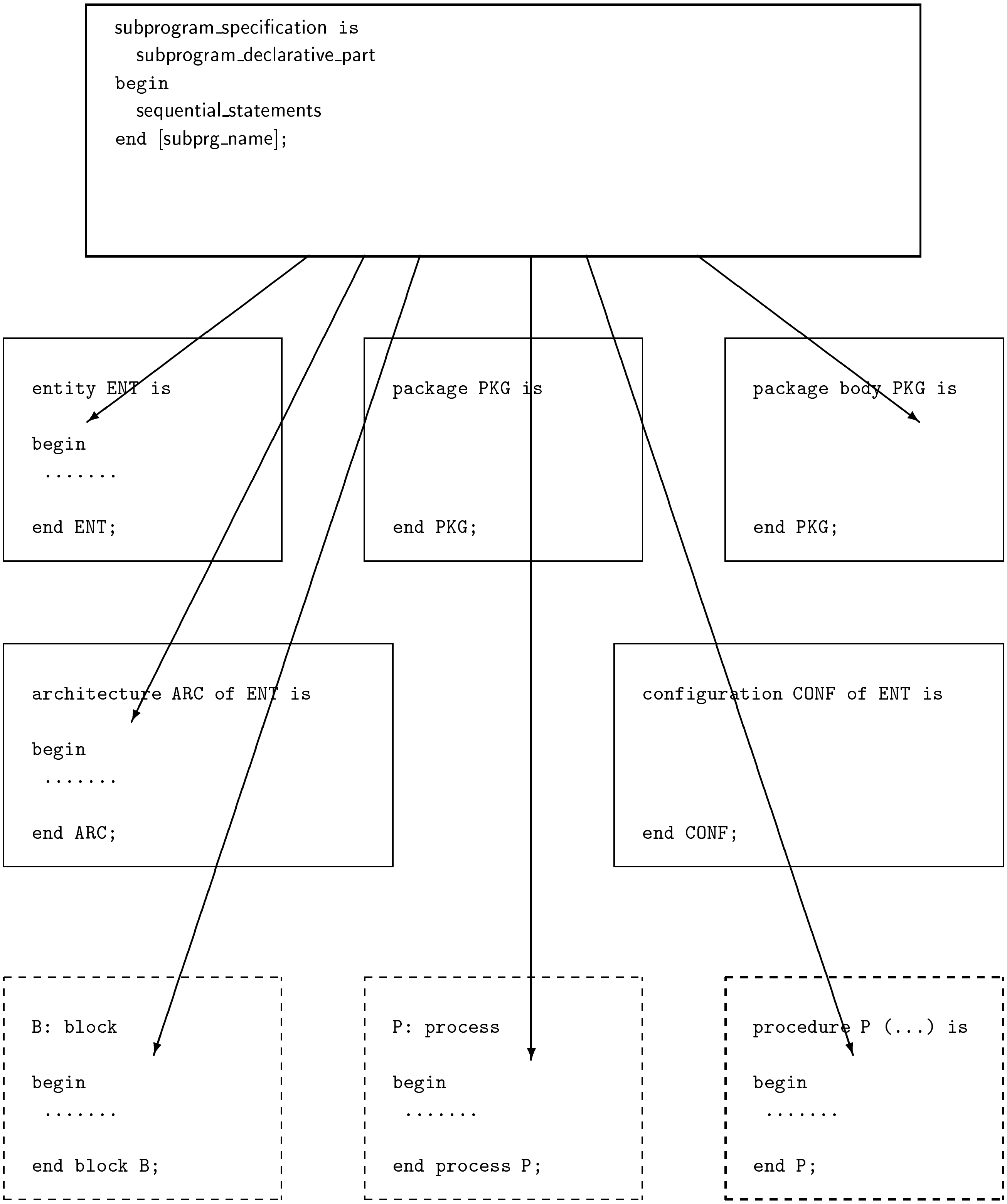
signal_declaration



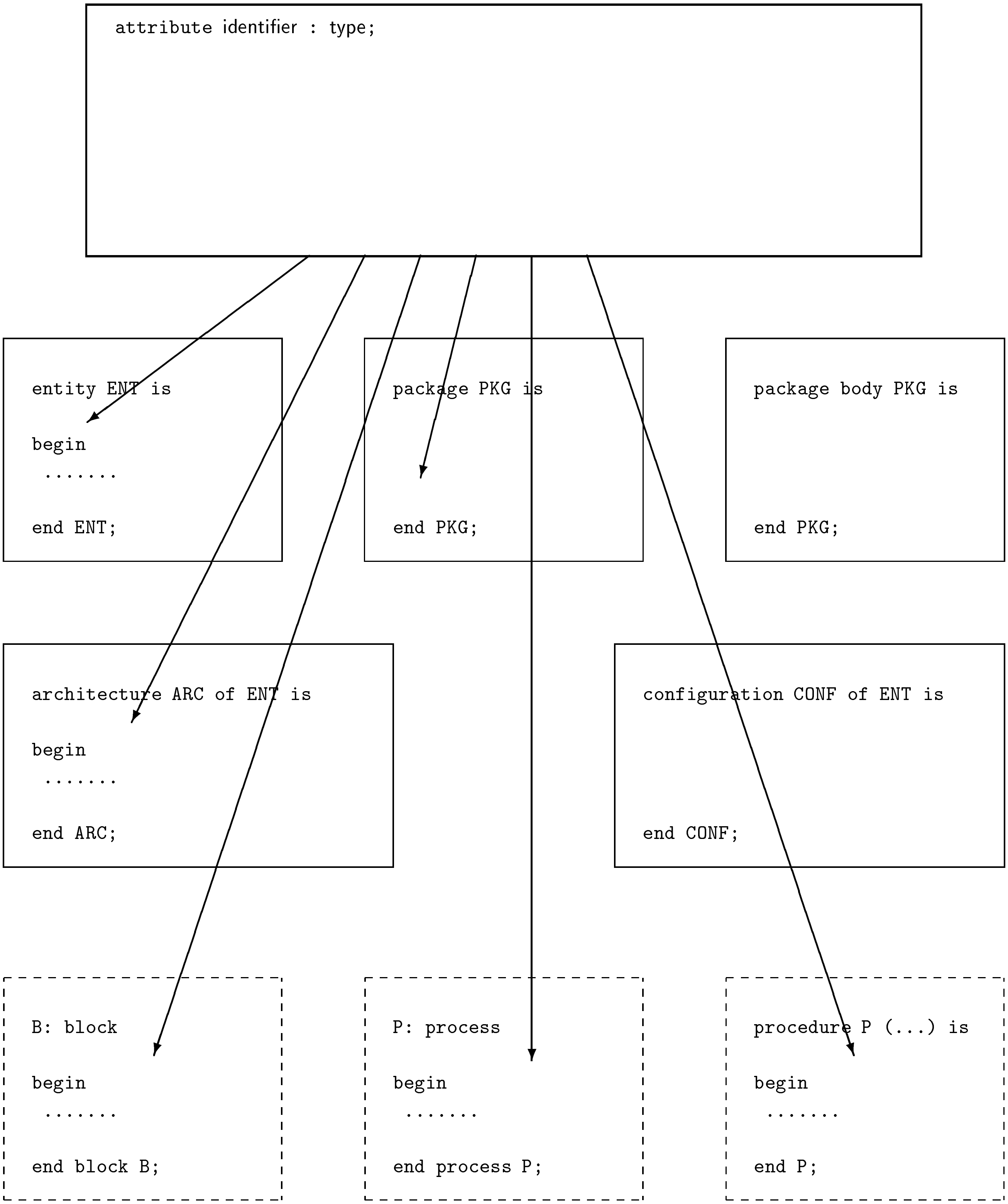
subprogram_declaration



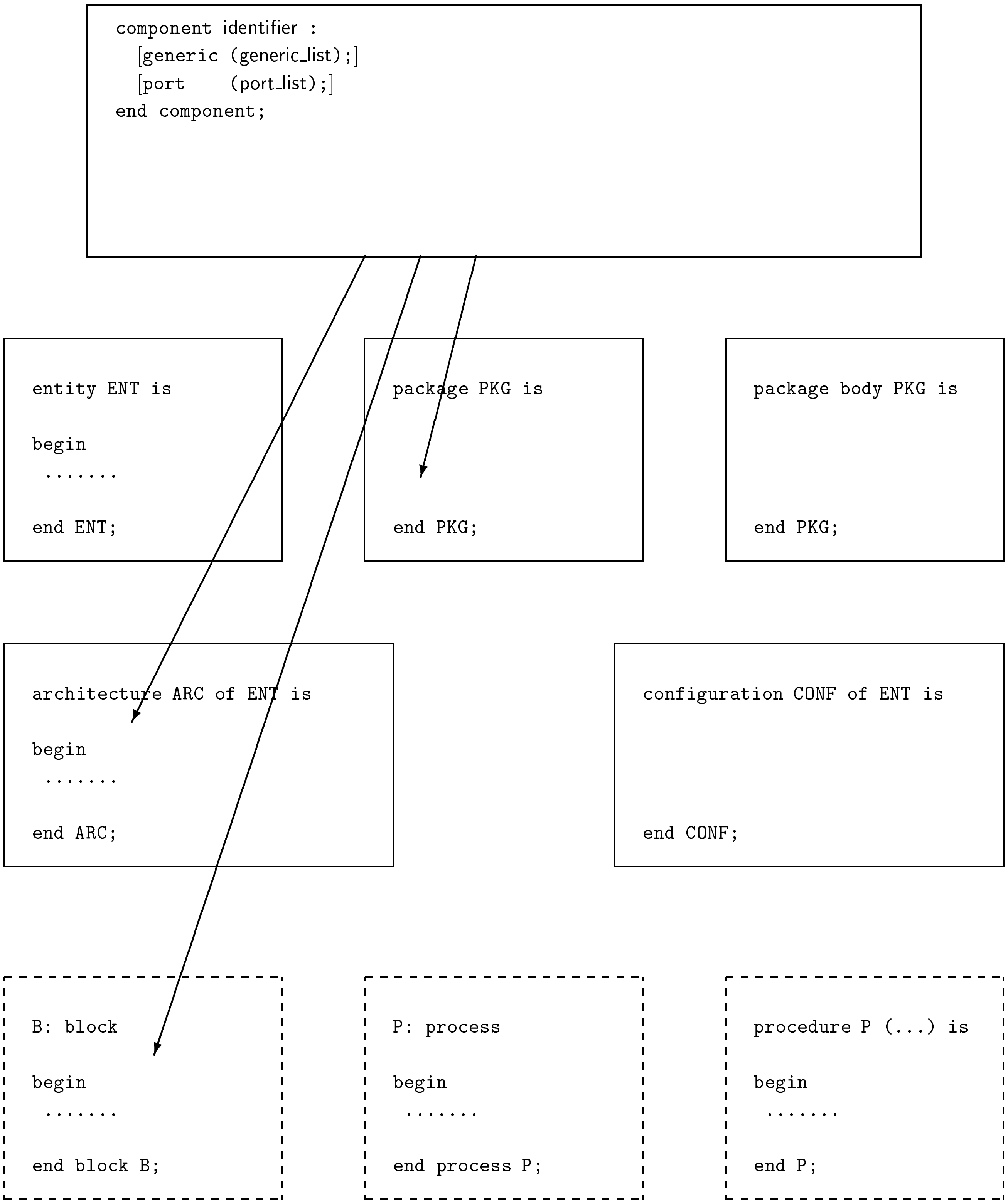
subprogram_body



attribute_declaration

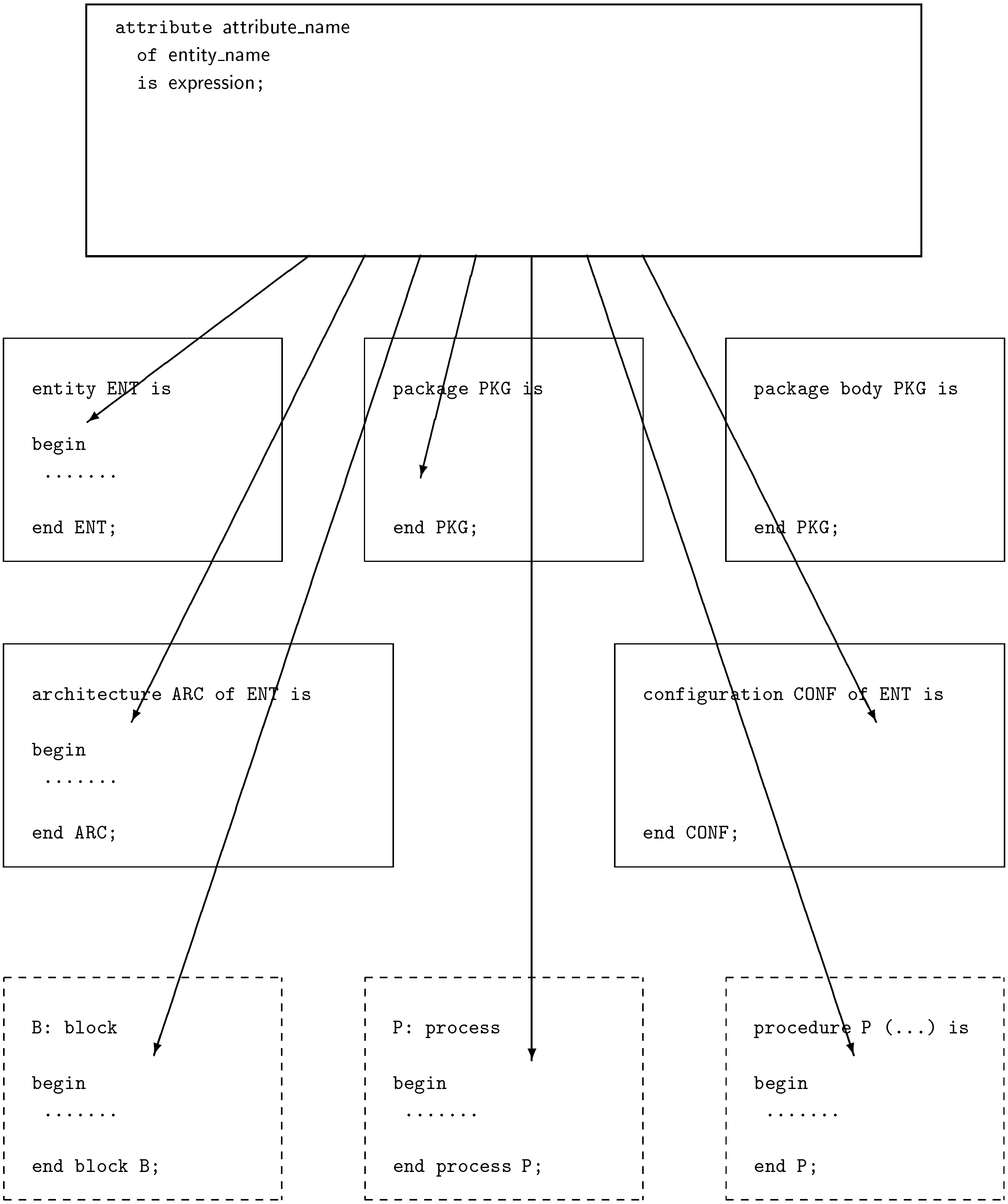


component_declaration

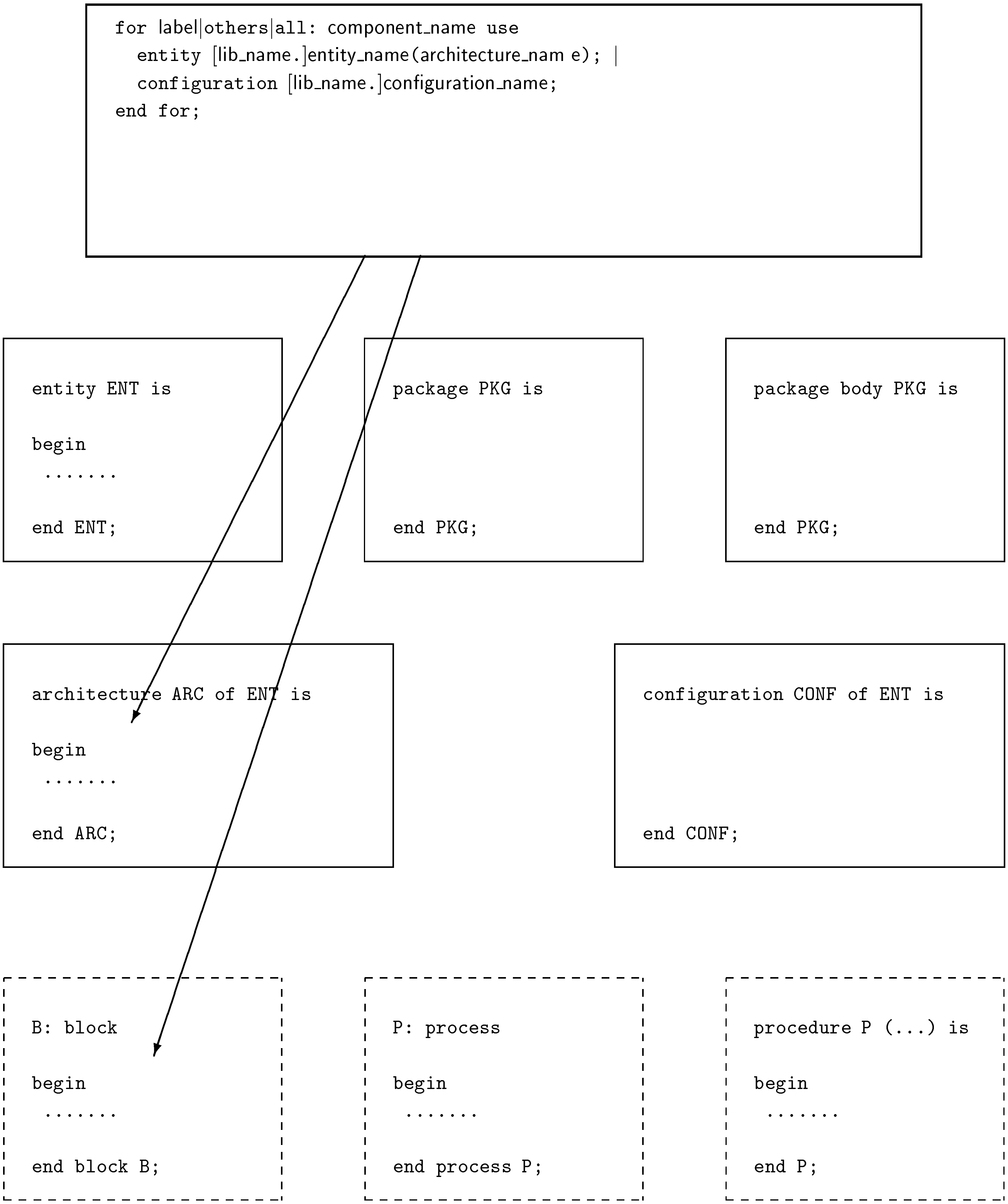


A.4 Spezifikationen

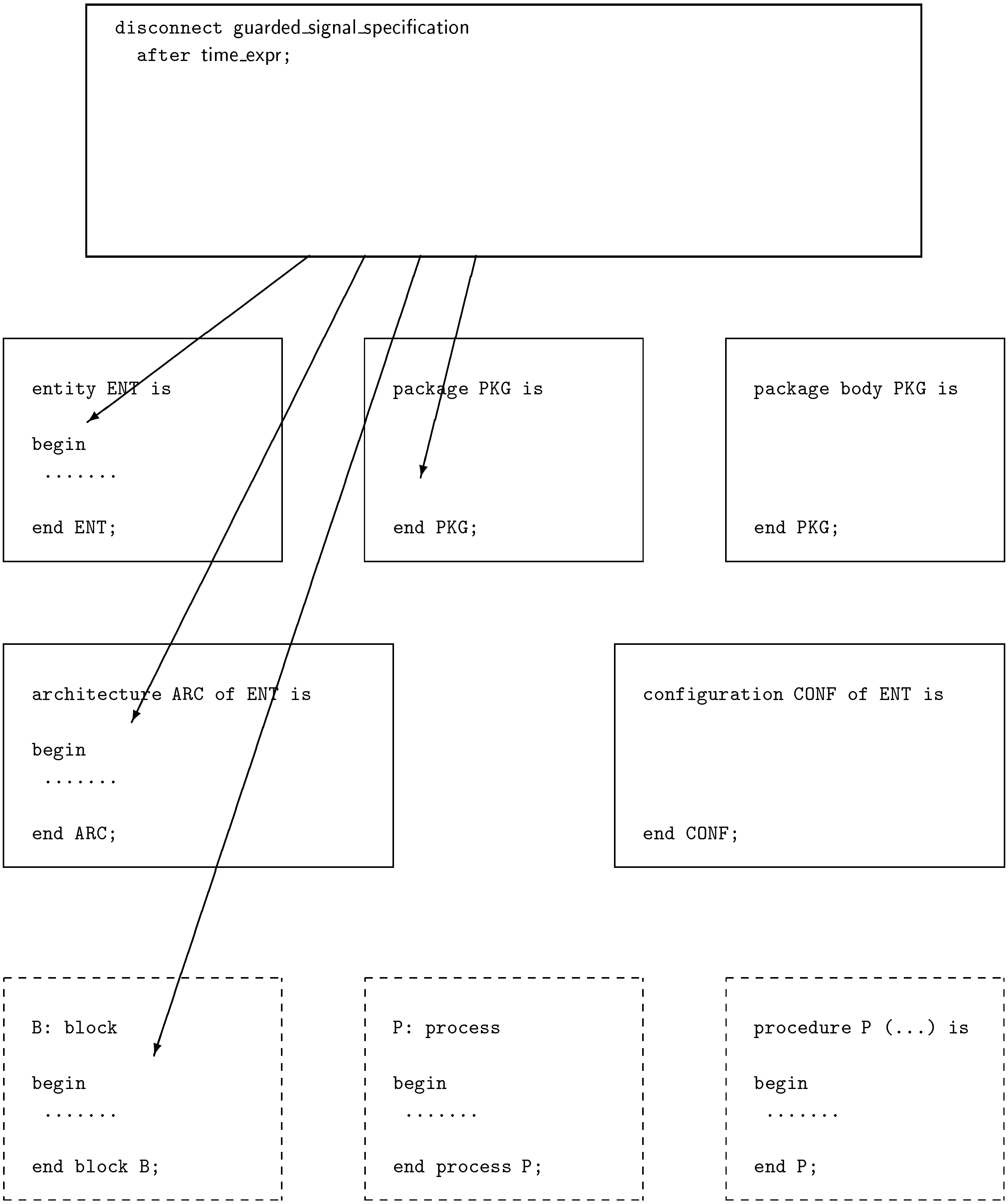
attribute_specification



configuration_specification

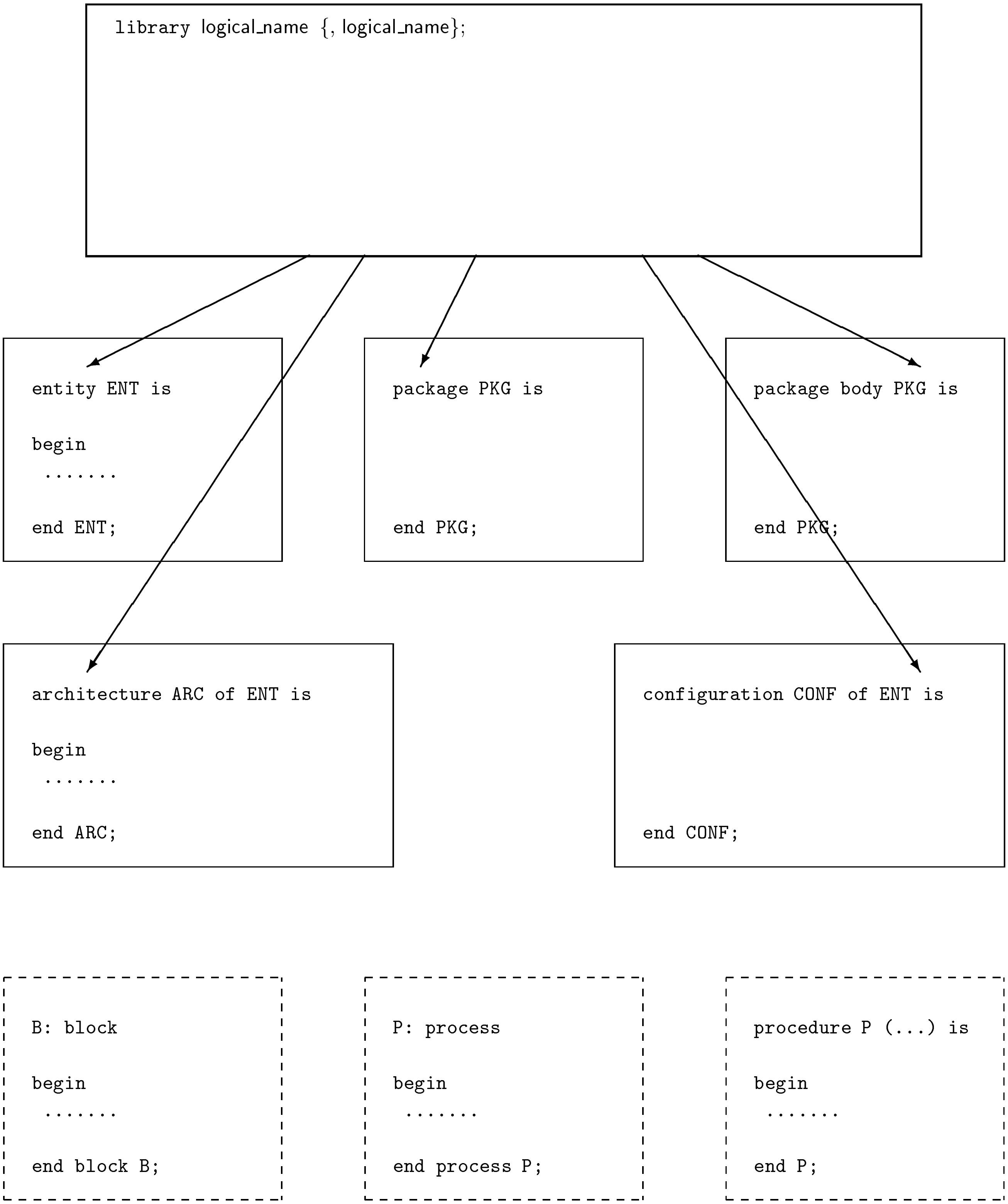


disconnection_specification

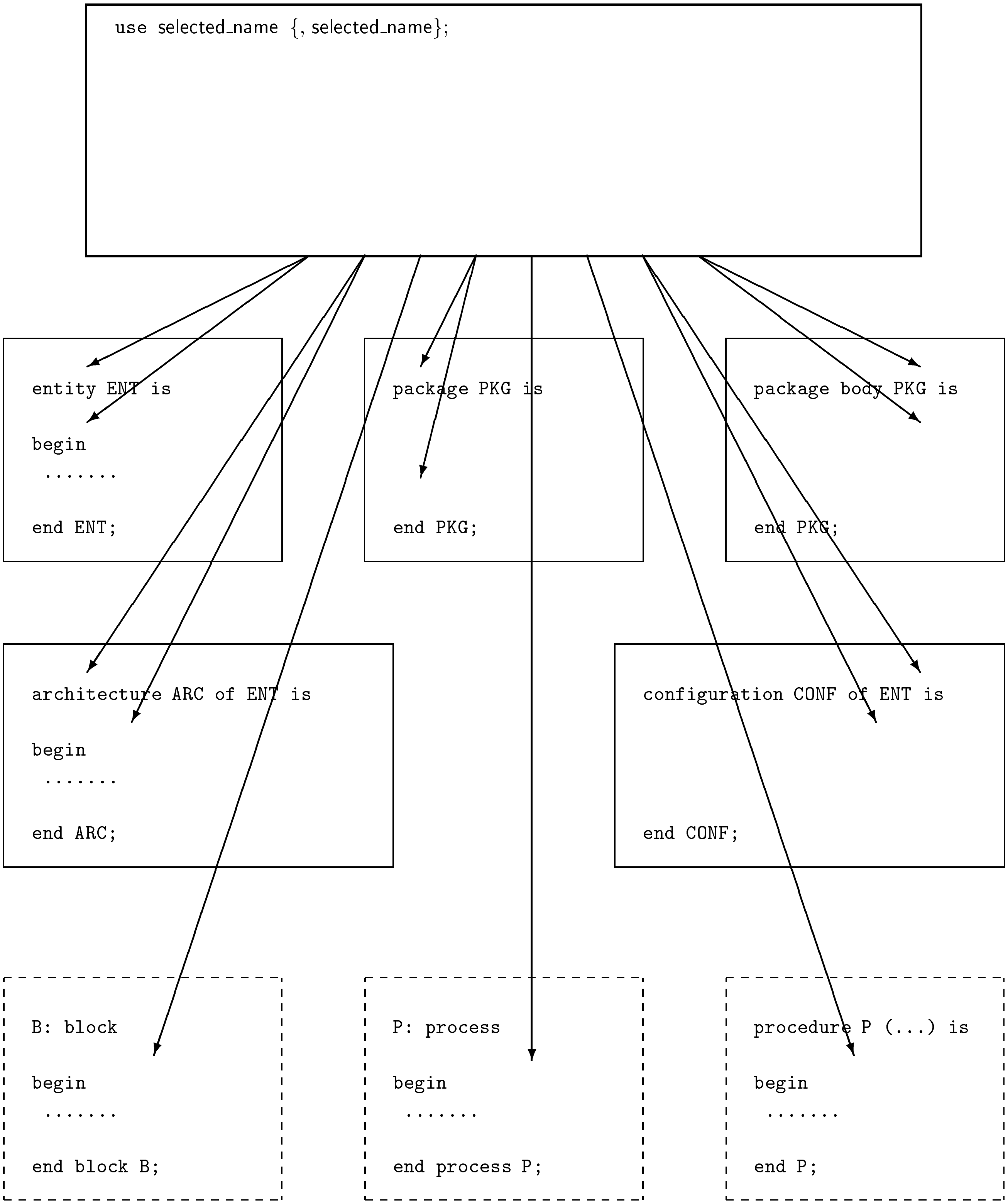


A.5 Bibliotheksbenutzung

library_clause



use_clause



use selected_name {, selected_name};
entity ENT is
begin
....
end ENT;
package PKG is
end PKG;
package body PKG is
end PKG;
architecture ARC of ENT is
begin
....
end ARC;
configuration CONF of ENT is
end CONF;
B: block
begin
....
end block B;
P: process
begin
....
end process P;
procedure P (...) is
begin
....
end P;

A.6 sequentielle Anweisungen

wait_statement

```
wait
  [on signal_name {, signal_name}]
  [until conditional_expr]
  [for time_expr];
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

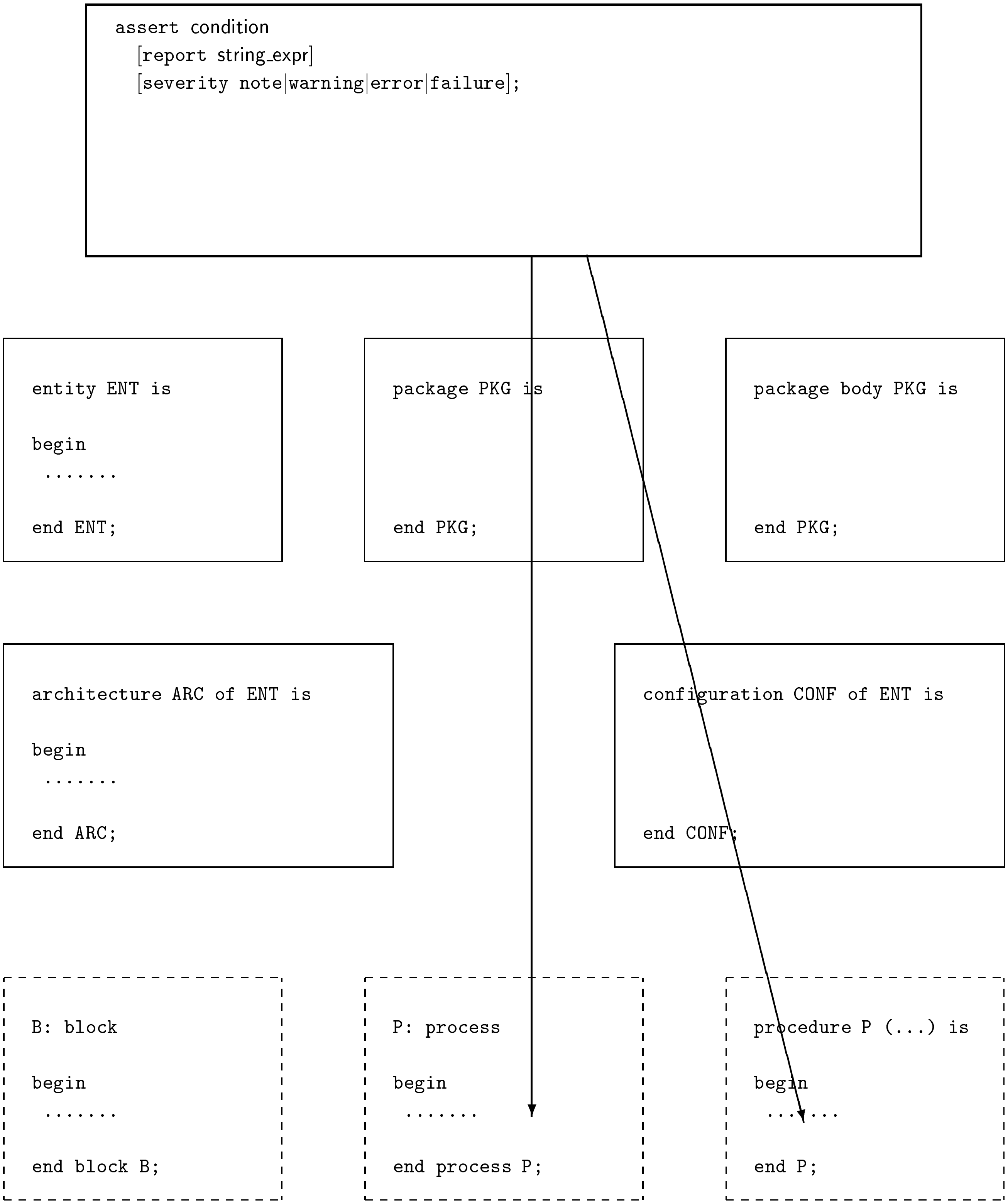
```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

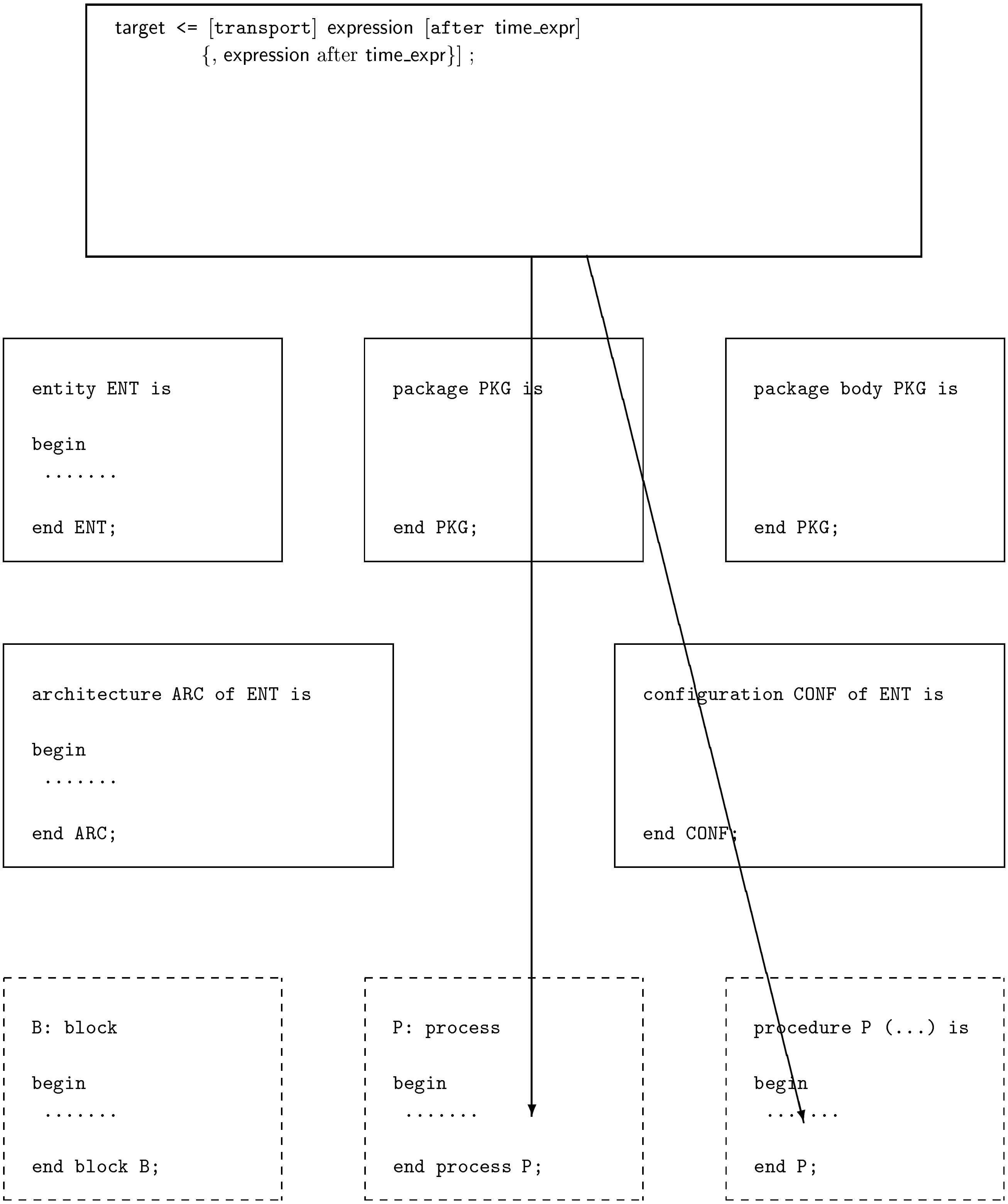
```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

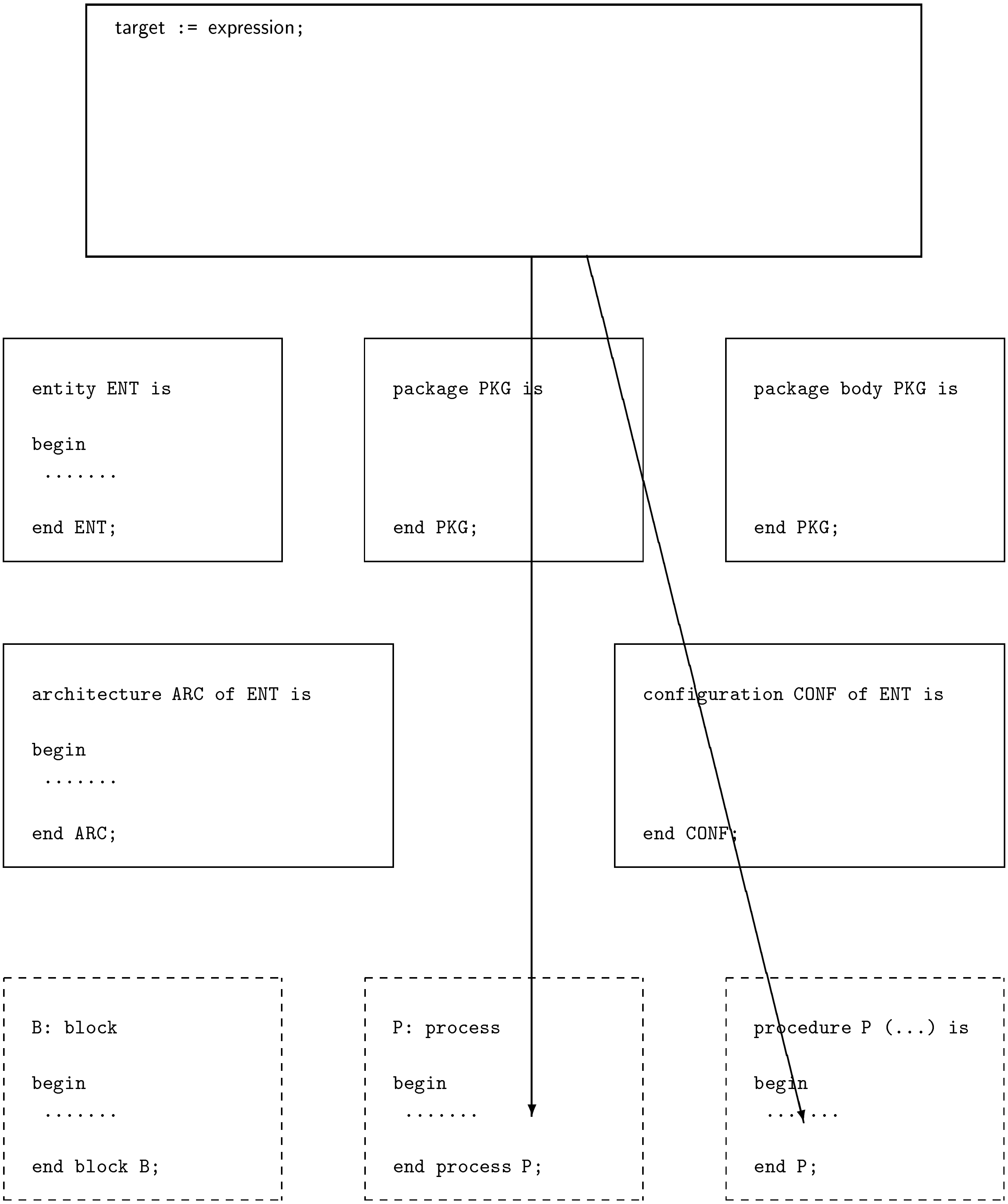
assertion_statement



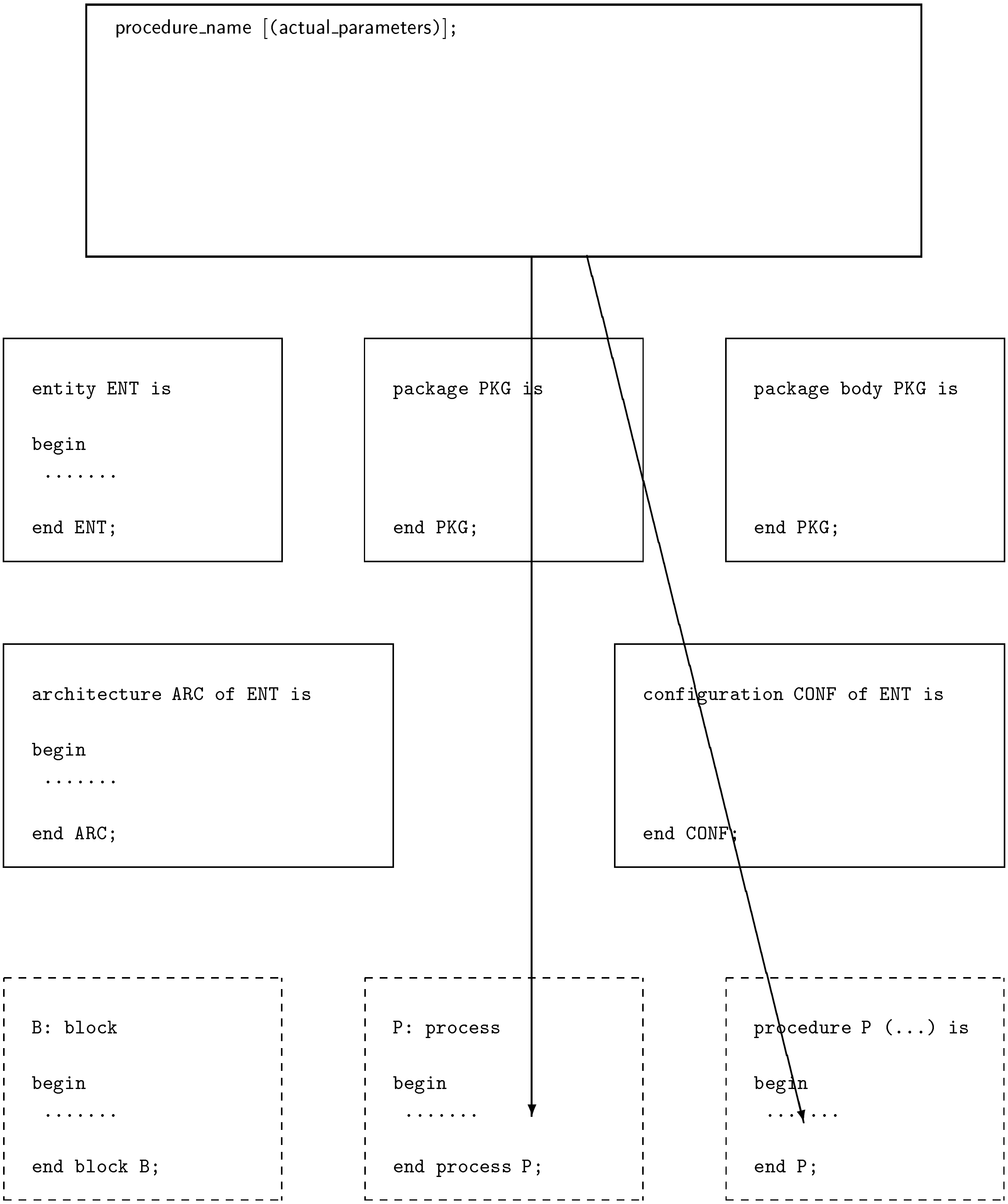
signal_assignment_statement



variable_assignment_statement



procedure_call_statement



if_statement

```
if condition then
  sequential_statements
  {elsif condition then
    sequential_statements}
  [else
    sequential_statements]
end if;
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

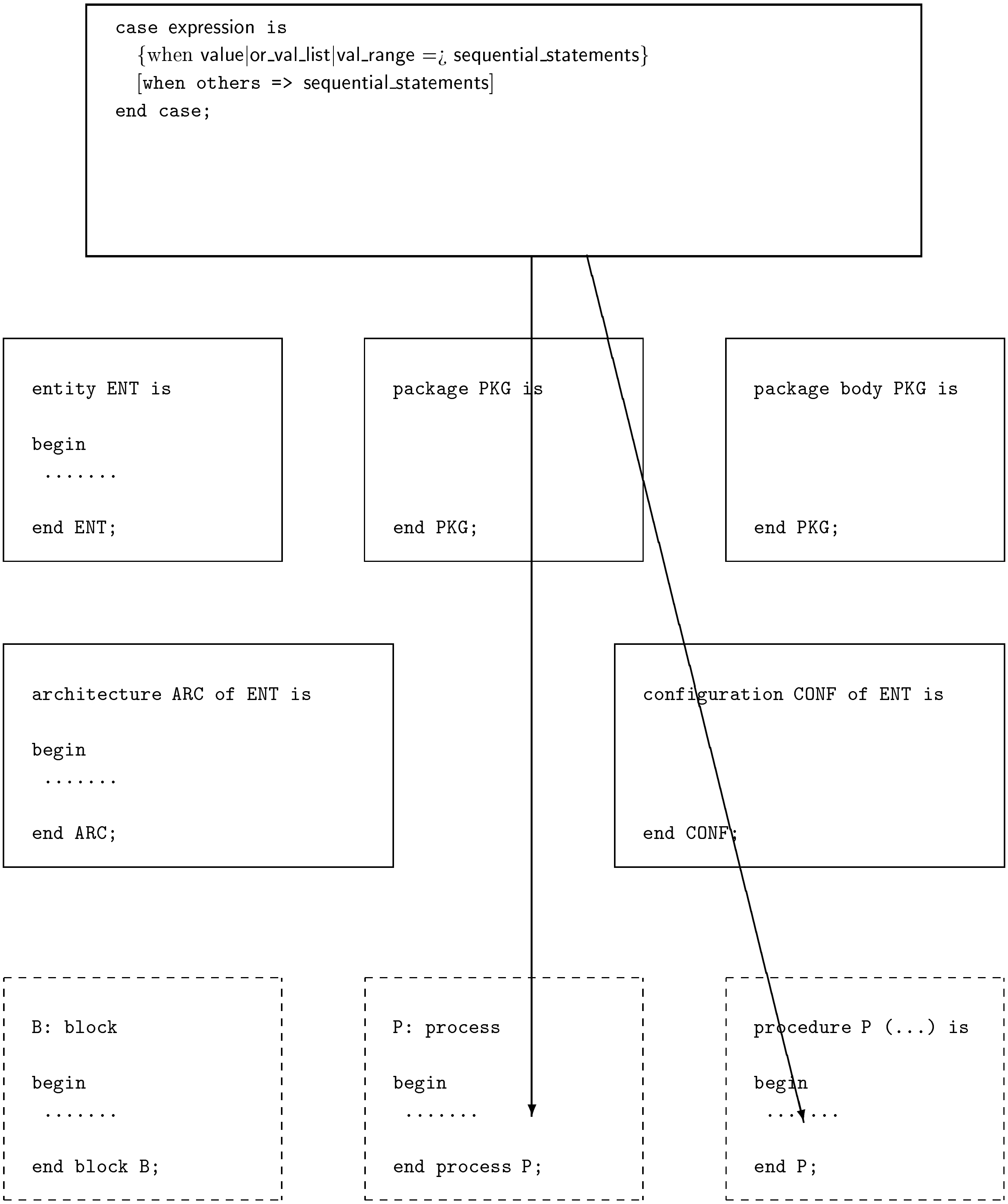
```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

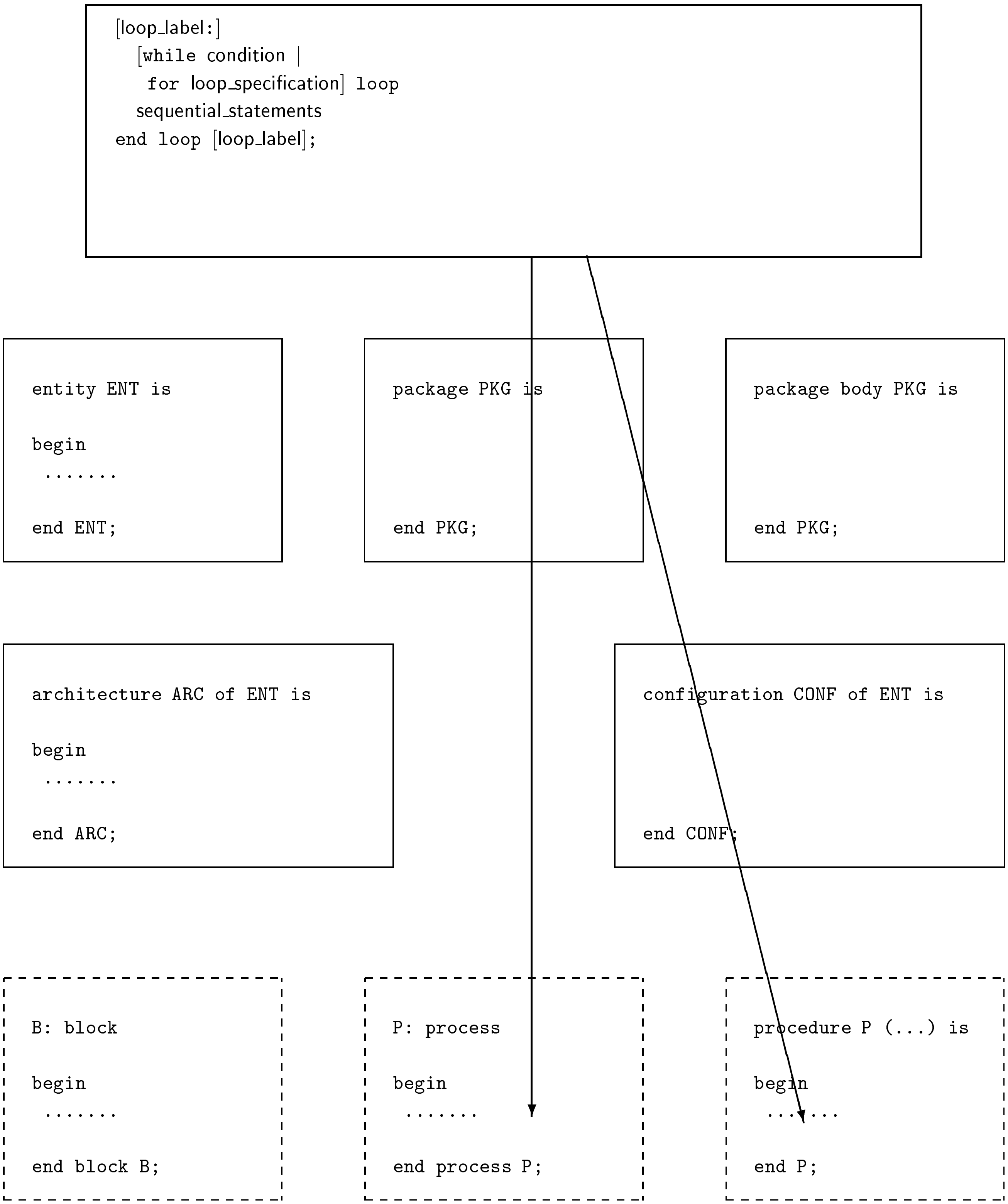
```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

case_statement



loop_statement



next_statement

next [loop_label]
  [when condition];

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

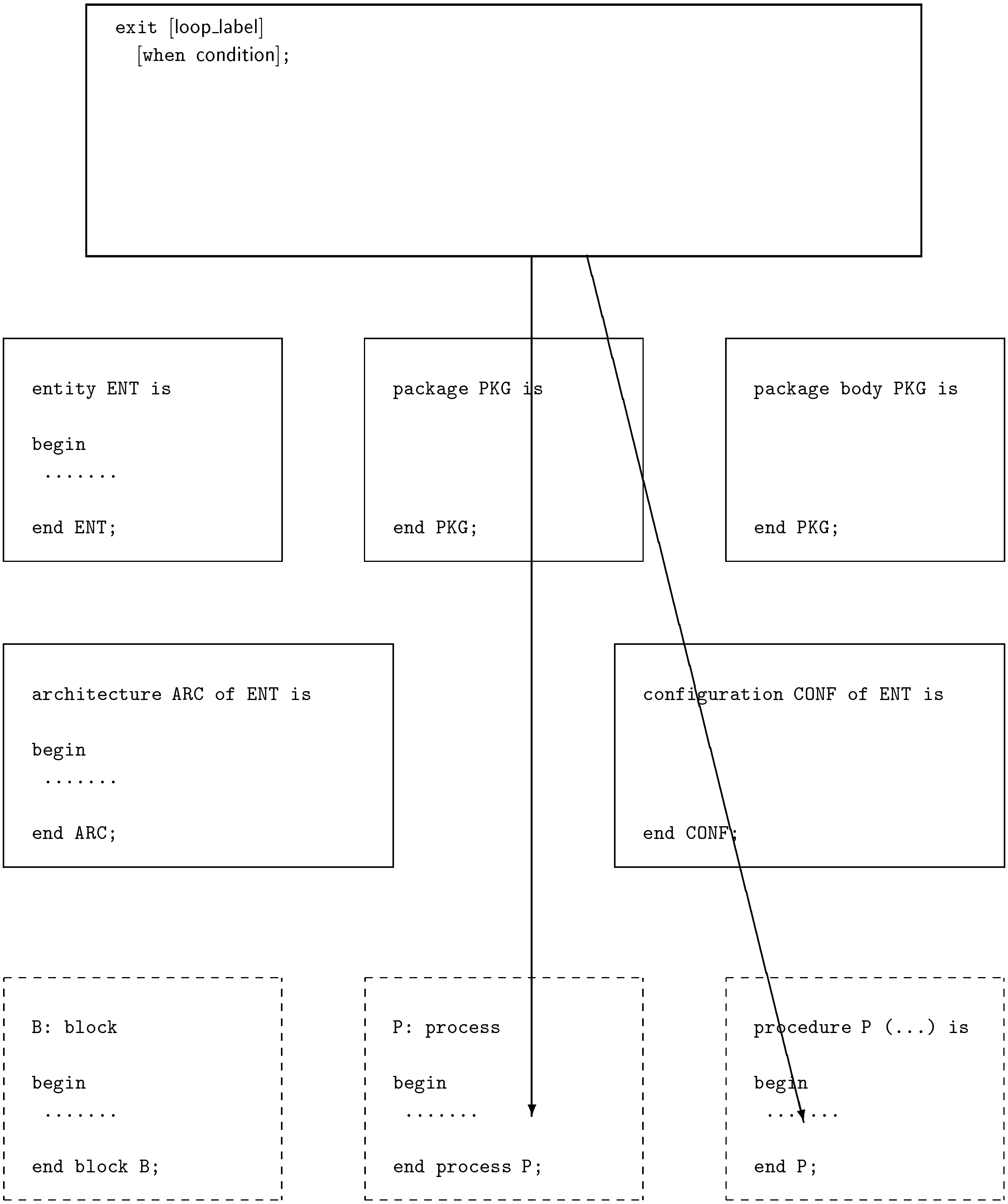
```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

exit_statement



return_statement

```
return [expression];
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

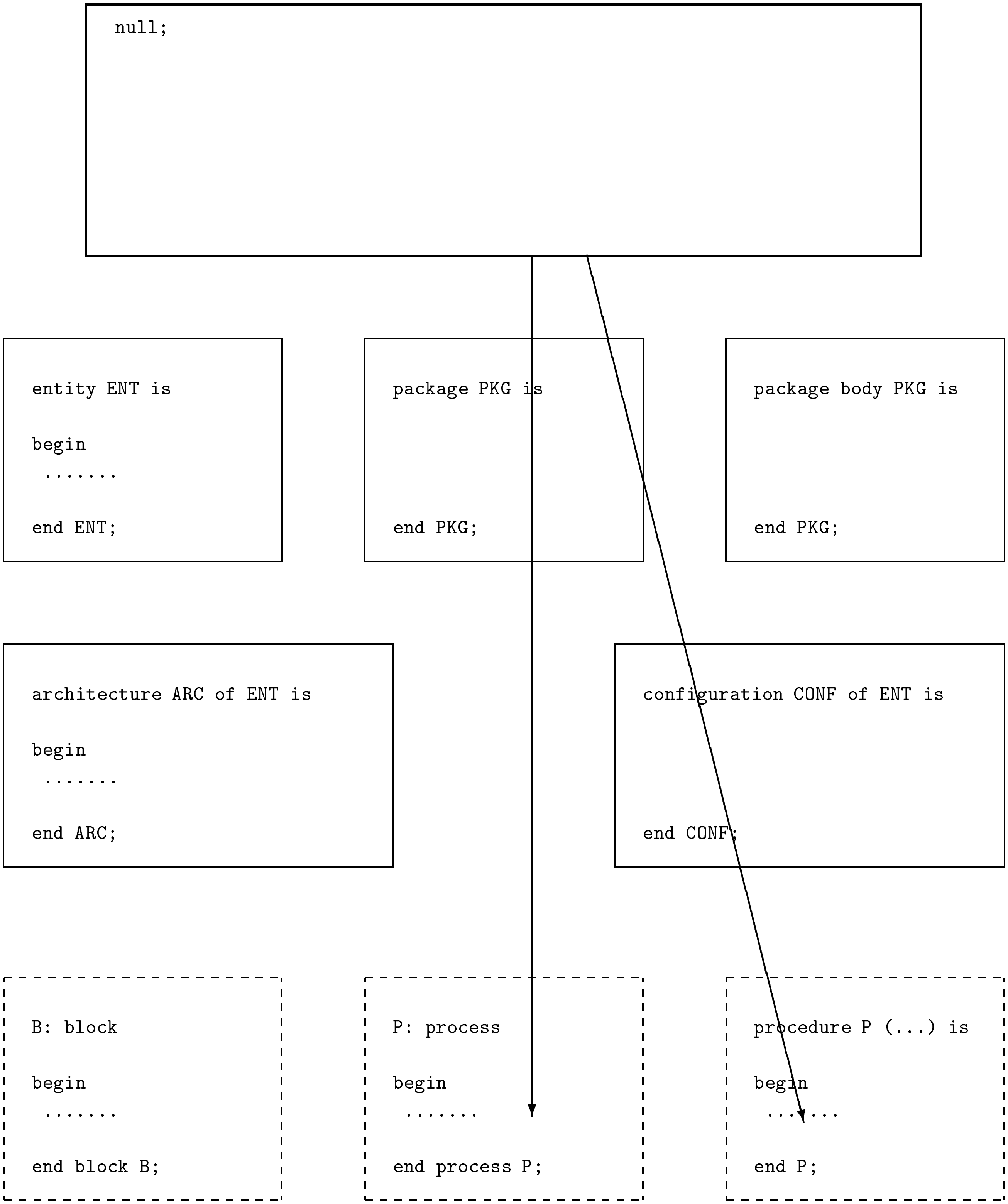
```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

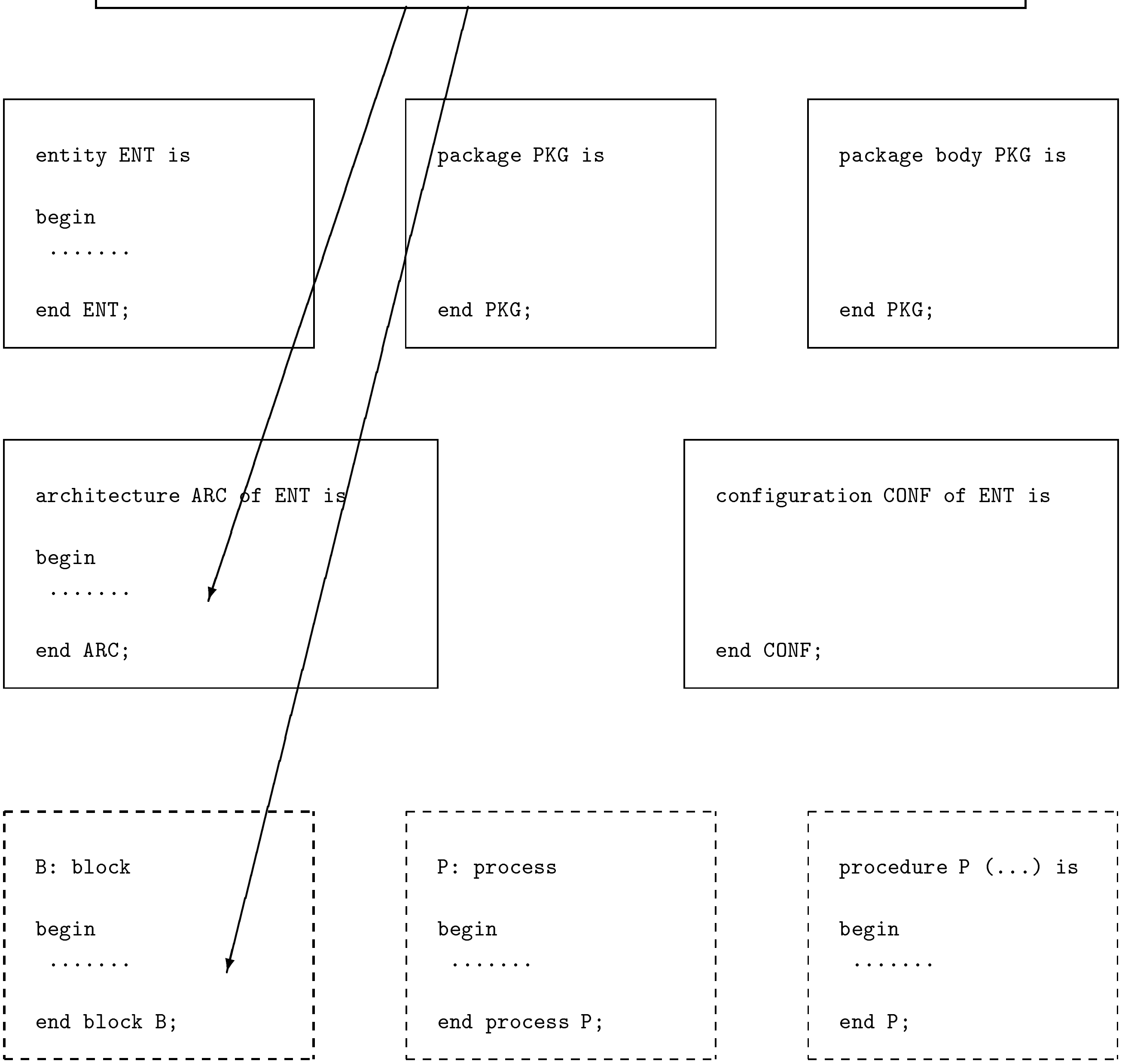
null_statement



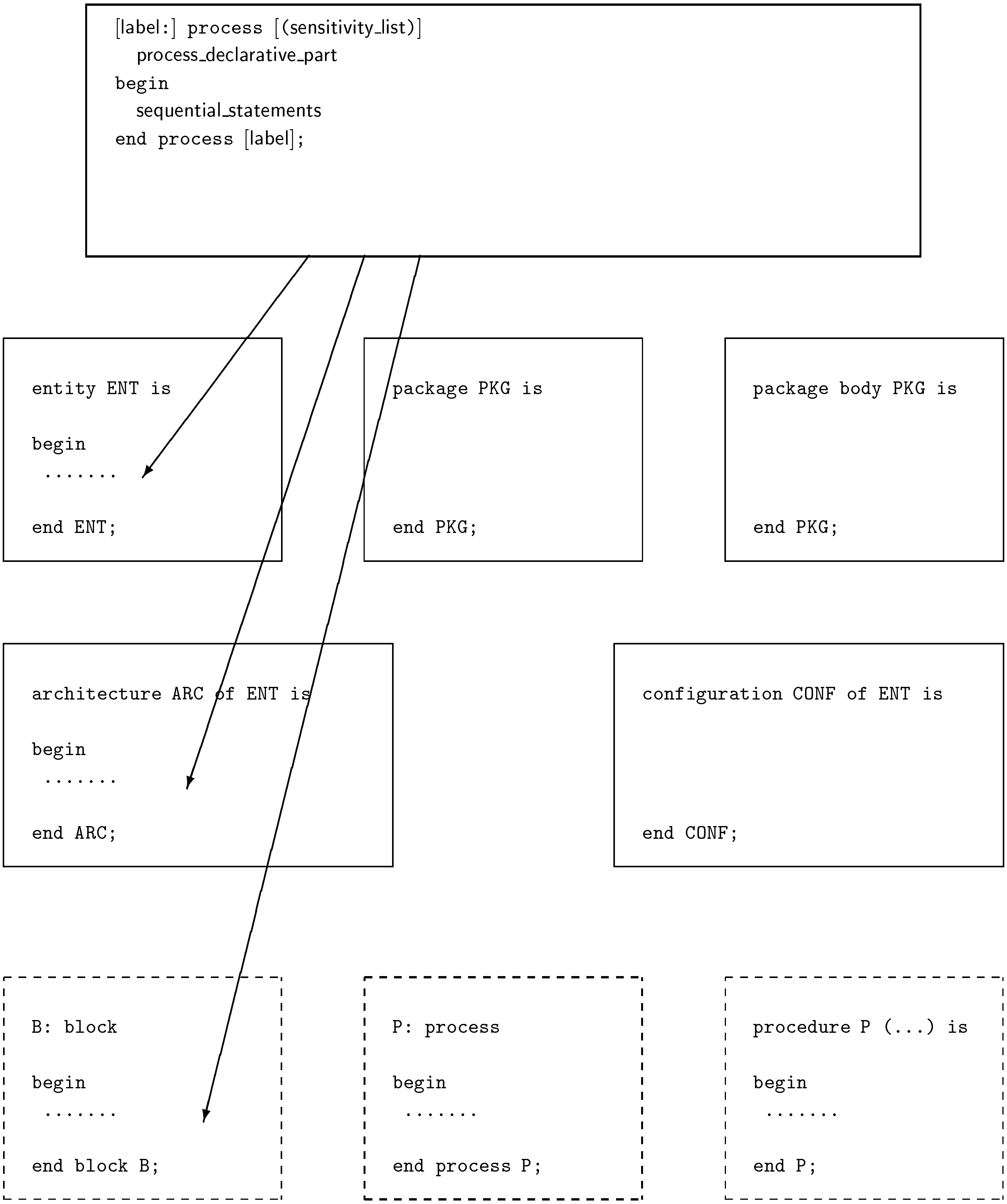
A.7 konkurrente Anweisungen

block_statement

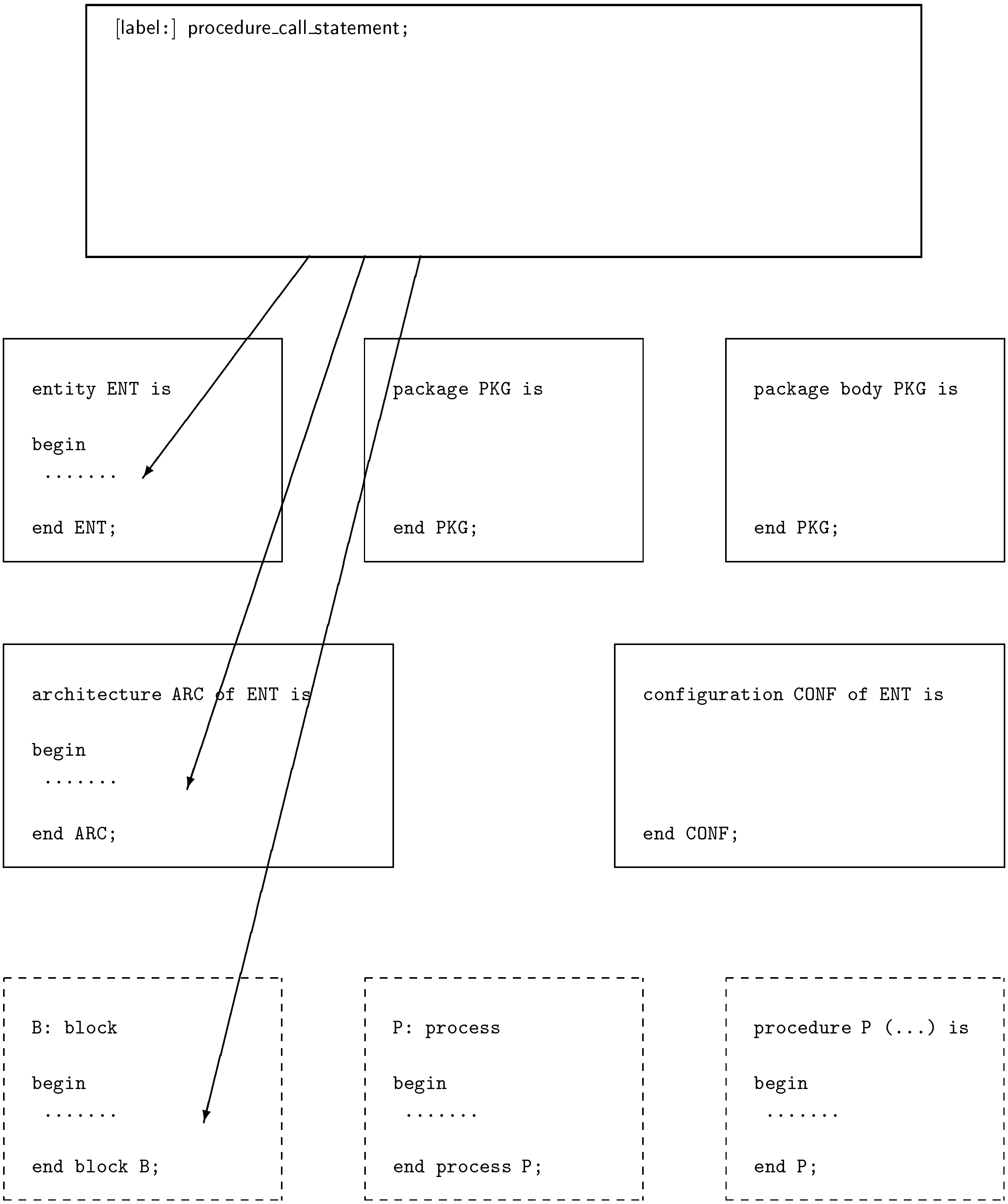
```
label: block [(guard_expression)]
  [generics]
  [ports]
  [declarations]
begin
  concurrent_statements
end block [label];
```



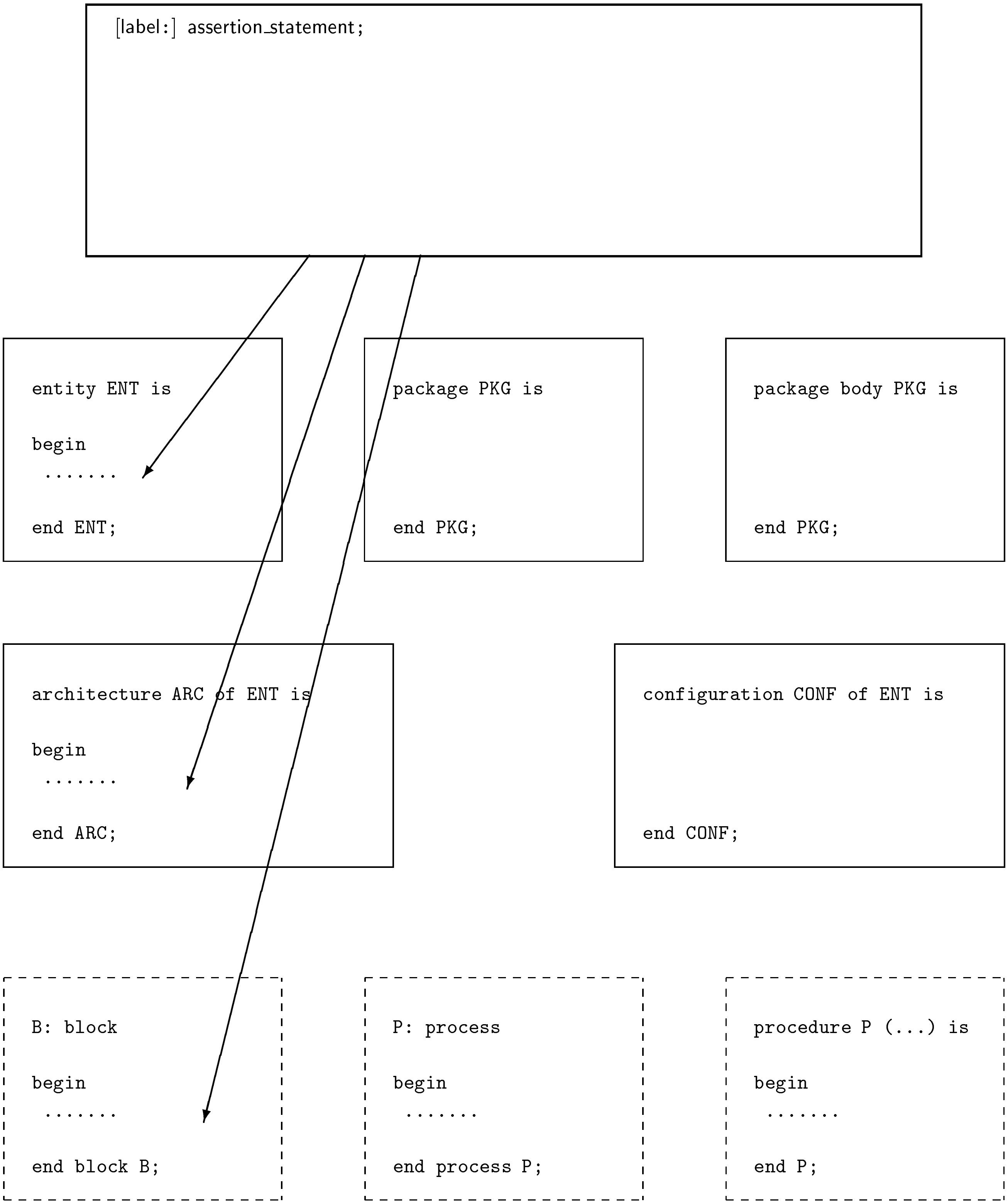
process_statement



concurrent_procedure_call



concurrent_assertion_statement



concurrent_signal_assignment_statement

```
[label:] target <=
   {expression [after time_expr] when condition else}
   [guarded] expression [after time_expr];

– or –

[label:] with expression select
   target <= {expression [after time_expr] when choices[,]};
```

```
entity ENT is

begin
 .......

end ENT;
```

```
package PKG is



end PKG;
```

```
package body PKG is



end PKG;
```

```
architecture ARC of ENT is

begin
 .......

end ARC;
```

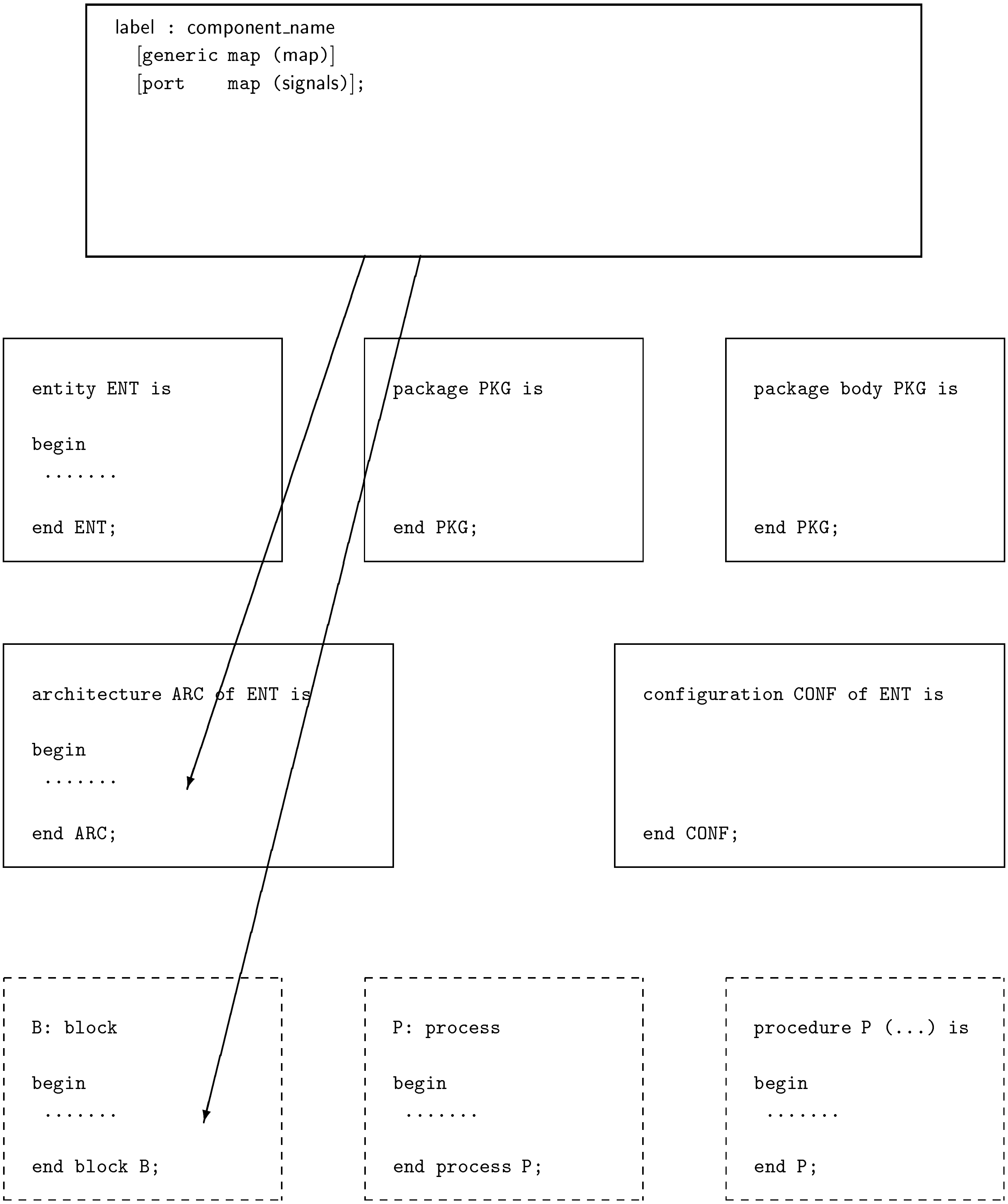
```
configuration CONF of ENT is



end CONF;
```

```
B: block

begin
 .......

end block B;
```

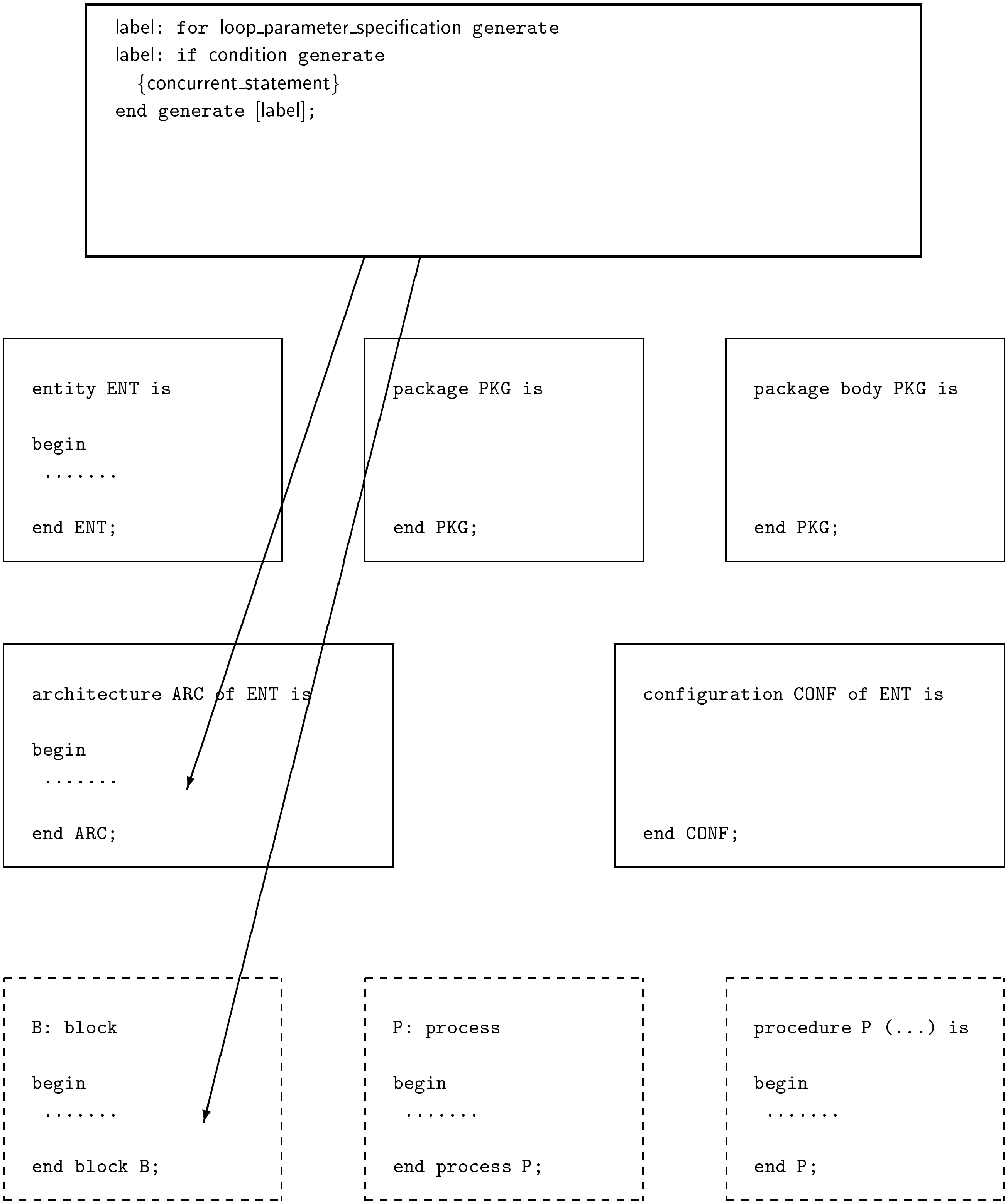
```
P: process

begin
 .......

end process P;
```

```
procedure P (...) is

begin
 .......

end P;
```

component_instantiation_statement



generate_statement



A.8 Vordefinierte Attribute

Attribut	Prefix	Parameter	Ergebnis
auf Typen bezogen:			
`t'base`	Typ/Untertyp t		Basistyp zu t, nur als Präfix anderer Attribute zulässig
`t'left`	skalarer Typ t		linke Grenze von t
`t'right`	skalarer Typ t		rechte Grenze von t
`t'low`	skalarer Typ t		untere Grenze von t
`t'high`	skalarer Typ t		obere Grenze von t
`t'pos(x)`	diskreter oder physikalischer Typ t	Ausdruck: Wert ist in Typ t enthalten	Position von Wert x in t
`t'val(x)`	diskreter oder physikalischer Typ t	Ausdruck: integer	Wert in t: auf Position x
`t'succ(x)`	diskreter oder physikalischer Typ t	Ausdruck: Wert ist in Typ t enthalten	Wert in t: Positionsnr. ist eine größer als bei x
`t'pred(x)`	diskreter oder physikalischer Typ t	Ausdruck: Wert ist in Typ t enthalten	Wert in t: Positionsnr. ist eine kleiner als bei x
`t'leftof(x)`	diskreter oder physikalischer Typ t	Ausdruck: Wert ist in Typ t enthalten	Wert in t: links von x
`t'rightof(x)`	diskreter oder physikalischer Typ t	Ausdruck: Wert ist in Typ t enthalten	Wert in t: rechts von x
auf Arrays bezogen:			
`a'left`[`(n)`]	Array-Objekt, Alias eines Array oder begrenzter A-Untertyp	Ausdruck: integer, innerhalb der Dimensionen von a, Default 1	linke-Grenze des n-ten Index von a
`a'right`[`(n)`]	Array-Objekt, Alias eines Array oder begrenzter A-Untertyp	Ausdruck: integer, innerhalb der Dimensionen von a, Default 1	rechte-Grenze des n-ten Index von a
`a'low`[`(n)`]	Array-Objekt, Alias eines Array oder begrenzter A-Untertyp	Ausdruck: integer, innerhalb der Dimensionen von a, Default 1	untere-Grenze des n-ten Index von a
`a'high`[`(n)`]	Array-Objekt, Alias eines Array oder begrenzter A-Untertyp	Ausdruck: integer, innerhalb der Dimensionen von a, Default 1	obere-Grenze des n-ten Index von a
`a'range`[`(n)`]	Array-Objekt, Alias eines Array oder begrenzter A-Untertyp	Ausdruck: integer, innerhalb der Dimensionen von a, Default 1	Bereich: n-ter Index entsprechend Deklaration
`a'reverse_range`[`(n)`]	Array-Objekt, Alias eines Array oder begrenzter A-Untertyp	Ausdruck: integer, innerhalb der Dimensionen von a, Default 1	umgekehrter Bereich: n-ter Index entspr. Deklaration
`a'length`[`(n)`]	Array-Objekt, Alias eines Array oder begrenzter A-Untertyp	Ausdruck: integer, innerhalb der Dimensionen von a, Default 1	Anzahl der Werte des n-ten Index von a

Attribut	Prefix	Parameter	Ergebnis
auf Signale bezogen:			
`s'delayed(t)`	Signal `s`	Ausdruck: time $\geq$ 0, Default 0 ns	Signal: zu `s` äquivalent, zeitlich um `t` verzögert
`s'quiet(t)`	Signal `s`	Ausdruck: time $\geq$ 0, Default 0 ns	bool-Signal: true –`s` war die letzten `t` Zeiteinheiten ruhig
`s'stable(t)`	Signal `s`	Ausdruck: time $\geq$ 0, Default 0 ns	bool-Signal: true –`s` hatte `t` Zeiteinh. keine Wertewechsel
`s'transaction`	Signal `s`		bit-Signal: Wechsel / toggle –`s` ist im Sim.zyklus aktiv
`s'event`	Signal `s`		bool-Wert: true –`s` hat ein Event
`s'active`	Signal `s`		bool-Wert: true –`s` ist im Sim.zyklus aktiv
`s'last_event`	Signal `s`		Zeit seit dem letzten Event von `s`
`s'last_active`	Signal `s`		Zeit seit letzter Aktivität von `s` in einem Sim.zyklus
`s'last_value`	Signal `s`		Wert von `s` vor dem letzten Wertewechsel
Attribut	Prefix	Parameter	Ergebnis
auf Blöcke bezogen:			
`b'behavior`	Block-Label `b` oder architecture `b`		bool-Wert: true –`b` instanziiert keine Komponenten
`b'structure`	Block-Label `b` oder architecture `b`		bool-Wert: true –`b` enthält keine Prozesse oder vergl. (nicht passive) Anweisungen

A.9 Reserviert Bezeichner

abs
access
after
alias
all
allow
and
architecture
array
assert
attribute

begin
block
body
buffer
bus

case
component
configuration
constant

disconnect
downto

element
else
elsif
end
entity
exit

file
for
function

generate
generic
group
guarded

if
impure
in
initial
inout
is

label
library
linkage
literal
loop

map
mod

nand
new
next
nor
not
null

of
on
open
or
others
out

package
port
postponed
private
procedure
process
pure

range
record
register
reject
rem
report
return
rol
ror

select
severity
signal
shared
sla
sll
sra
srl
subtype

then
to
transport
type

unaffected
units
until
use

variable

wait
when
while
with

xnor
xor

Literatur

[Arm88] J. R. Armstrong: *Chip-Level Modeling with VHDL*, Prentice Hall, Englewood Cliffs, 1988

[Bha92] J. Bhasker: *A VHDL Primer*, Prentice Hall, Englewood Cliffs, 1992

[Coe89] D. Coelho: *The VHDL Handbook*, Kluver Academic Publishers, Norwell MA, 1989

[HaEA91] R. E. Harr, et al: *Applications of VHDL to Circuit Design*, Kluver Academic Publishers, Norwell MA, 1991

[IEEE94] *IEEE Standard VHDL Language Reference Manual – Std 1076-1993*, IEEE, New York, 1994

[LeSh89] S. Leung, M. Shanblatt: *ASIC System Design with VHDL – A Paradigm*, Kluver Academic Publishers, Norwell MA, 1989

[LSU89] R. Lipsett, C. Shaefer, C. Ussery: *VHDL: Hardware Description and Design*, Kluver Academic Publishers, Norwell MA, 1989

[MaLa93] S. Mazor, P. Langstraat: *A Guide to VHDL*, 2nd edition, Kluver Academic Publishers, Norwell MA, 1993

[Per89] D. Perry: *VHDL*, McGraw Hill, New York, 1989

[PeTa96] D. Pellerin, D. Taylor: *VHDL Made Easy!*, Upper Saddle River NJ, Prentice Hall, 1996

Index