Hierarchical Modular Description of VLSI Systems

Franz J. Rammig

University of Dortmund, Abt. Inf. I

Abstract

In this paper it is shown how the design process from functional specification down to the discrete transistor level can be supported by a single CHDL named CAP/DSDL. The language is based upon a single semantical model: Timed Interpreted Petri Nets. Language concepts that are general or dedicated to specific levels of abstraction will be discussed.

1. Introduction

Language design always has been an important aspect of software engineering. After first rather simple approaches of Hardware Description Languages now relatively sophisticated languages have been developed in this field. They are more or less influenced by results from software engineering. As an example of such a language may serve CAP/DSDL which has been designed and implemented by the author and his collegues and has been used very successfully by SIEMENS AG in numerous projects since a couple of years. The language makes use of concepts like Abstract Data Types, Monitors, Petri Nets, Strong Typing, Structured Programming, Assertions, all originating from software engineering.

2. General Remarks

This paper is not intended to be a general introduction to CAP/DSDL but only specific language features are to be discussed. However, as this will be done on the basis of examples some basic principles of the language have to be introduced.

We deffer between constants, types and variables. The basic data object is the bitstring of arbitrary length bit(n). The basic type constructors (array and record) are taken from PASCAL. Records are identified with bitstrings of the length which is computed as the sum of their components' length. Consequently each bitstring can be viewed as a record where components are to be identified by their location within the record, e.g. a.(7 : 0) denotes the leftmost eight bitposition of a bitstring which has to be at least of length 8. As operators we have logical ones (PL/1 notation) &, |, ¬, ¬&, ¬|, ⊕ (exor), ¬⊕ which also may serve as reduction operators, relational ones =, <>, <, >, <=, >=, arithmetic ones +, -, *, /, mod and concatenation ‖ . Relational and arithmetic operators interpret arguments as two's complement integers as long as they are not enclosed in bars; e.g. + means a two's complement addition while |+| means an unsigned integer one. Expression and assignements follow the PASCAL syntax with an "if then else" and a "case of" construct included.

3. Algorithmic constructs

The basic principle for behavioural descriptions in CAP/DSDL is that of Timed Interpreted Petri Nets. These nets may be specified by a designer directly or indirectly via structured concurrent constructs. The latter approach has some limitations but has to be favoured whenever applicable. Experience has shown that nearly all practical control structures can be formulated with structured constructs. For convenience three basic types of transitions are offered by CAP/DSDL:

- AND transition (usual Petri Net transition)
  Notation: on (in_1 & in_2 & ... & in_k) do mark (out_1 & ... & out_n)
- OR transition (place with backward conflict in usual Petri Nets)
  Notation: on (in_1 | in_2 | ... | in_k) do mark (out_1 & ... & out_n)
- DECIDER transition (place with foreward conflict in usual Petri Nets)
  Notation: on (in_1) do if cond then mark (out_1)

The variables involved are special objects of type place. In their declaration their capacity may be bound to a finite value.

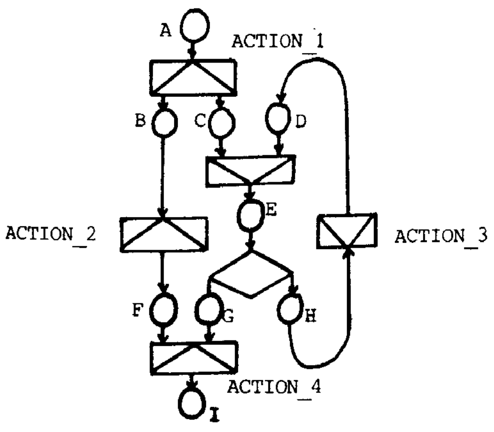
By well known reasons special net templates are offered to the user via own syntactical constructs. If a user restricts himself to use exclusively these constructs it is ensured that the resulting net is 1-safe, deadlock free and reusable.

The constructs offered are the following:

- seqbegin S_1; ...; S_n end
- conbegin S_1; ...; S_n end
- if cond then S_1 else S_2
- case cond of S_1; ...; S_n end
- while cond do S
- repeat S until cond
- for a := i seqto f do S
- for a := i conto f do S

The seqbegin construct corresponds to the begin construct in PASCAL (with same differences that shall not be discussed here) while conbegin means the concurrent execution of the included statements S_1 to S_n. The complete statement will be terminated when all included statements are terminated. In the for statements we allow only constants as bounds. The reason becomes clear immediately if one considers the conto alternative. This means that all statements S_a (S with the proper index value) have to be executed concurrently. It should be noted that we don't have a loop in this case.

The advantage of the structered approach can be seen immediately with the aid of a small example: The control structure to be described may consist of two concurrent branches, where one may be a loop. Such a control structure is reflected by the following CAP net:



This net may be described directly:

```
var A, B, C, D, E, F, G, H, I : place;
net
  on(A) do mark(B & C) ACTION_1;
  on(B) do mark(F)     ACTION_2;
  on(C | D) do mark (E);
  on(E) do if CO then mark(H)
                 else mark(G);
  on(H) do mark(D)     ACTION_3;
  on(F & G) do mark(I) ACTION_4
end
```

Much easier to understand is the following description of the same net:

```
seqbegin
   ACTION_1
   conbegin
      ACTION_2;
      while CO do ACTION_3
   end
   ACTION_4
end
```

It should be noted, that up to now no timing concept has been introduced, neither via delay operators nor via clocking. These possibilities will be discussed later. In an early design phase the CAP/DSDL user can be liberated from these implementation details. He only has to specify a concurrent causality structure as indicated in the above example.

4. Data driven control

In the above section the basic ideas of CAP/DSDL in order to support the algorithmic RT level have been presented. To describe the flow of data through the combinational parts of a hardware system an other concept is offered by CAP/DSDL.

A data variable that is declared with attribute explicit (or without attribute) is interpreted as device with storing capability (e.g. register). It gets a new value if and only if it is ordered to do so explicitely by the control structure. (An assignment statement is executed due to the control structure.)

Variables that are declared with the attribute implicit are interpreted as non storing devices (e.g. wires, outputs of combinational logic). They are allowed to stand on the left side of exactly one assignment statement. It gets a new value implicitly whenever one of the variables within the expression on the right hand side of this assignment statement gets a new value. These statements have to be grouped in a special section of a CAP/DSDL description.

It should be noted that this is a single assignment rule for implicit variables. In fact, CAP/DSDL descriptions may be restricted completely to implicit variables leading to a functional programming style.

The following small example may give an idea about this capability of CAP/DSDL. It describes a gate level solution of a 16 bit wide RS register:

```
var R, S, Q, NQ : implicit bit(16);
impdef
     Q  := R nor NQ;
     NQ := S nor Q
```

Of course the lexical ordering of the assignment statement within the impdef section of a CAP/DSDL description has no influence on the meaning of the description.

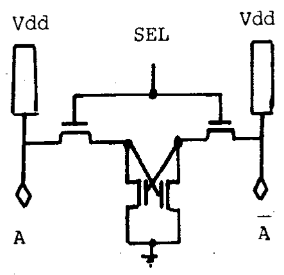
This language feature seems expecially adequate for gate level descriptions of systems or parts of a system within more abstract descriptions. In section 5 it will be shown how the discrete transistor level (switching level) can be covered with the aid of this technique, too. In section 6 we will introduce timing and will show how this concept also offers the framework for the structural (nonprocedural RT level).

5. MOS specific features

MOS introduces a bidirectional point of view into the language. This level of abstraction is covered using the techniques of data driven control. CAP/DSDL offers three builtin procedures for this purpose:

pullup(A) and pulldown(A) with an obvious meaning and transfer (techn, gate, left, right) in order to model transistors. The parameter "techn" controls whether a nMOS or pMOS transistor is described. The parameter "gate" is a unidirectional input while the parameters "left" and "right" are bidirectional ones. Variables that are mentioned as such bidirectional parameters of transistors are interpreted as seven valued ones (per bit) with the meaning: low impedance one and zero, medium impedance one and zero (pulled up or down), high impedance one and zero (charged and then isolated), uncharged. Uncertain values are described by the subset of this set that contains exactly the possible values (so we are working with a 128 valued logic for this purpose). Such variables are restricted on implicit variables. In their declaration a charge decay time can be given. So dynamic storage elements and dynamic logic can be described precisely.

As an example may serve the description of a dynamic nMOS RAM cell:



```
const nMOS = "1";
var A, NOTA, SEL, LEFTMEM, RIGHTMEM: implicit bit
                                      decay(1ØØ);

impdef
   pullup(A);
   pullup(NOTA);
   transfer(nMOS, SEL, A, LEFTMEM);
   transfer(nMOS, SEL, NOTA, RIGHTMEM);
   transfer(nMOS, RIGHTMEM, LEFTMEM, "Ø");
   transfer(nMOS, LEFTMEM, RIGHTMEM, "Ø");
```

6. Timing and clocked systems

Timing in hardware design usually is expressed either by counting clock signals or by the real time behaviour of the envolved actions. In many causes both methods are used and therefore both are supported by CAP/DSDL.

Every assignment statement and every empty statement may include delay specification. The actual delay time may be specified via an arbitrary expression of type integer (bit(n)). So it is very easy to specify delays that are dependent on actual values, states, histories. Intervals of uncertaincy may be specified, too. By delay is meant in CAP/DSDL the period between the evaluation of the arguments of an assignment and the assignment of the resulting value. Within a sequential control structure the statement following the delayed one is initiated after the specified delay time.

Example

```
var R, S : bit; Q, NQ : implicit bit
impdef
   Q  := NQ var R delay (up 5 to 7, down 4 to 6);
   NQ :=  Q var S delay (up 6 to 8, down 3 to 6);
seqbegin
   R || S := "Ø1" delay (if S = "1" then 1Ø else 5Ø);
   R || S := "1Ø" delay (5Ø)
end
```

In order to describe clocked systems in CAP/DSDL first a clock generator has to be defined. This is usually be done via an implicit variable which is defined as its own complement with a proper delay.

Example

```
CLK := not CLK delay (up 5, down 45)
```

Now the edges or the levels of such clock signals may be interrogated by the at resp. when prefix. Any statement may be prefixed with these constructs. If this is done within the explicitely given control part this logical control specification remains unchenged. Only concerning timing it is synchronized with the clock signals.

Example: (see section 3, now synchronized with clock signals)

```
seqbegin
   at SLOWCLOCK do ACTION_1
   conbegin
      at SLOWCLOCK do ACTION_2;
      while CO do at FASTCLOCK do ACTION_3;
   end
   at SLOWCLOCK do ACTION_4
end
```

It should be noted that the at prefix is not restricted to clock like signals.

When used within the impdef part of a CAP/DSDL description the at resp. when statements constitute the main control structure.

The prefix may initiate arbitrarily complex operations in this case. It can be observed that now we have a data driven control of "second order". The isolated control part now may be viewed as a data driven system while the whole system may be viewed as a system triggered by control events. But this is the classical point of view of RT languages. In order to clarify the difference we usually call it structural RT level to distinguish it from the algorithmic RT level described in section 3.

Example: (usual finite state machine, LAMDA and SIGMA are user functions)

```
var CLK: implicit bit; STATE: bit(2); X,Y: bit(16);
impdef
   CLK := not CLK delay(50);
   at STATE = "ØØ" & CLK do conbegin
                               Y  := LAMDA(X,STATE);
                            STATE:= SIGMA(s,STATE)
                            end;
   at STATE = "Ø1" & CLK do ...
   :
   :
```

7. Modularization concepts

CAP/DSDL offers one basic concept for modularization: The procedure. But this single concept has been designed in such a way that the concepts of

- Modules
- Abstract Data Types
- Generic Objects
- Monitors (management of critical sections)

can be subsumed.

First of all a CAP procedure is a procedure in the sense of PASCAL. It constitutes a context with the scope of variables rule from PASCAL. The only difference is that all variables are static ones (own in ALGOL terms). I.e. they maintain their value after the deactivation of a procedure and may be used after a reactivation of it. There may be a formal parameter list. Here in, out and inout parameters are distinguished. Parameter passing is always by reference. However if an actual parameter is a constant or an expression that consists of more symbols than a single variable, the value of the actual parameter is first copied to a dummy variable that is passed by reference (i.e. de facto parameter passing by value).

Like in PACAL also a function procedure is offered using the PASCAL notation.

There is a strong type checking of the formal parameters against the actual ones. This is true also in the case of separately compiled procedures and functions. A separately compiled procedure or function has to be declared like an internal one with the difference that its body is substituted by the keyword external. So the type checking between the formal parameters and the actual ones can be carried out by the compiler while the check whether the declared procedure is compatible with the referenced one is done by the binder.

Example

```
procedure PROCEDURE_DEMO;
   var F,G : bit(16); H : bit(17);
   procedure SWAP (inout A,B : bit(16));
      conbegin
         A || B := B || A
      end;
   function SUM (in A,B : bit(16)): bit(17);
      external
   conbegin
      SWAP(F,G);
      H := SUM(F, F & G)
   end.
```

A procedure or function must contain an explicit control part (i.e. a compound statement). It remains active after a activation until this control part terminates. If a never ending control part is used and implicit variables are used as parameters then procedure are well suited to modularize combinational circuits, too. The "calling" in the main program then has the meaning of a "power on".

Example

```
procedure FULLADD (in A,B,C,CIN : implicit bit;
                   out S, COUT  : implicit bit);
   const NEVER = "Ø";
   procedure MAJORITY (in A,B,C : implicit bit;
                       out D    : implicit bit);
      impdef
         D := A&B | A&C | B&C ;
      seqbegin at NEVER do end;
   procedure PARITY (in A,B,C   : implicit bit;
                     out D      : implicit bit);
      impdef
         D := (exor) (A || B || C);
      seqbegin at NEVER do
   conbegin
      MAJORITY (A,B,CIN,S);
      PARITY   (A,B,CIN,COUT)
   end.
```

Of course in reality nobody would modularize such a small circuit.

Every procedure or function includes a monitor mechanism. I.e. at one point of time a procedure or function can be active only once. All additional activations are delayed until the actually served one has terminated. Concurrent activations are served according to a fixed priority scheme by an arbiter. By this feature a procedure or function in CAP/DSDL models a once existing piece of hardware that may be requested concurrently but is allocated to requests in a time shared manner.

Procedures and functions may not be not declared as single objects but also as types of objects. Instances of such a type may be generated in the usual way using var declarations. It should be noted that a module concept is introduced into CAP/DSDL simply by overcoming a PASCAL restriction (absence of type: procedure). The module concept offers its benefits especially in regular structures. Presently such regular structures became more and more popular in VLSI design. Another important feature of CAP/DSDL modules (i.e. procedure types) is that they may be generic. In the type definition there may be a list of formal attributes standing for constants or types. By these attributes any objects within the procedure type definition may be attributed. In the var declaration these formal attributes have to be substituted by actual attributes.

Example:

```
procedure GENERIC_AND_MODULE_DEMO;
   type ADDER =
      procedure ADDER [WORD: type]
                      (in A,B: WORD, CIN : bit;
                      out SUM: WORD; COUT: bit);
         conbegin
            COUT ||SUM:=if CIN then("Ø"|| A)+("Ø"||B)+1
                                else("Ø"|| A)+("Ø"|| B)
         end
   var   ALU_ADDER : ADDER [bit(16)];
         ADR_ADDER : ADDER [bit(24)]
   .
   .
   .
```

Generic objects are especially suited to be stored in a data base for multiple use various designs. Structures like systolic arrays are supported by arrays of objects of type procedure.

A last (but especially powerful) concept that is included in CAP/DSDL procedures is that of Abstract Data Types (ADT). An abstract data type is defined as a carrier data structure (that may be an ADT as well) and a set of operations on this carrier. A user of an ADT has access to the data structure only via the offered operations. This idea is followed by export procedures in CAP/DSDL. An export procedure has no own control part (so it is the only exception to the rule started above). It consists of a local data structure (carrier data structure) and a set of procedures and functions that manipulate this carrier structure (offered operations on the carrier structure). These procedures and functions have to be listed in an export list in front of the procedure head. So just following the usual scope of variables the internal carrier data structure is hidden from the outside while the operations on it are made available (well controlled encapsulation technique). As an example may serve a very simple ALU that is able to carry out the operations add, sub, and, or, not. The result is stored into an internal register in any case. This register can be read by an additional operation read. This internal register is one bit longer than the used wordlength. In the leftmost bit the carry is stored in. In the example a generic solution is demonstrated.

```
type ALU = export (ADD,SUB,AND,OR,NOT,READ);
                 procedure ALU [WORDLENGTH: const];
   type WORD = bit (WORDLENGTH);
   var  BUFFER : bit (SORDLENGTH + 1)
   procedure ADD (in A,B : WORD; in CI : bit);
      external;
   procedure SUB (in A,B : WORD; in CI : bit);
```

```
      external;
   procedure AND (in A,B : WORD);
      external;
   procedure OR  (in A,B : WORD);
      external;
   procedure NOT (in A : WORD);
      external;
   function  READ  : bit (WORDLENGTH + 1);
      external;
   end;
   :
var ADR_ALU  : ALU[16];
    DATA_ALU : ALU[24];
   :
conbegin
   :
   DATA_ALU.ADD(REG[1], REG[0], REG[2].(0));
   :
   ADR := ADR_ALU.READ.(23 : 0);
   :
end
```

The concept of ADT's is especially valuable at the level of functional specification. Here typically a set of ADT's is specified from which operations are requested. This closes our discourse through the levels of abstraction. It has been shown by which language features different levels of abstraction are supported. These levels are (top down):

- Functional Specification
- Algorithmic Register Transfer Level
- Structural Register Transfer Level
- Gate Level
- Discrete Transistor Level.

Though being so powerful the language is easy to learn and to use as it is based on very few principles. This is reflected by a very good acceptance both in industry and at our university. It turned out that neither design engineers (in most cases with EE background) nor CS students had problems to understand these few principles (i.e. PASCAL + Timed Interpreted Petri Nets + Data Driven Control).

8. Additional language features

Two main features of the language have not yet been discussed: Assertions and Interrupts.

CAP/DSDL allows the user to formulate per procedure or function a set of assertions that must hold throughout the procedure's execution. It should be noted that there is a strict distinction between hardware description and description of certain features of this hardware (requested behaviour). Assertions are used either for formal verification purposes or in the case of simulation as tool that liberates the designer from the neccessity to read same inches of print out. Instead of looking for somewhat in the simulation result he formulates as assertion for what he looks and let the simulator do this job.

Interrupts are very suitable for the specification of highly event oriented systems like telecommunications or industrial control systems. The interrupt concept of CAP/DSDL allows to specify interrupt systems where concurrent algorithms are partially interrupted or that partially wait for interrupts.

While assertions are a levelindependent tool, the interrupt constructs are mainly designed in order to support the functional specification level.

9. References

We avoided references in the text but prefer a referencing annex.

A language reference manual for CAP/DSDL is given by /RA1/. As an introduction into the theory of Petri Nets may serve /PE1/. The idea of classical RT languages is explained very well in /DD1/. For monitors see /HO1/ while /LS1/ may serve as reference to ADT's. The RAM cell of section 5 can be found in /CL1/. Modularization techniques are presented in /WI1/.

/CL1/: W. A. Clarke: "From electron mobility to logical structure: A view of integrated circuits". ACM Comp.Surv.Vol.12# 3 (1980)

/DD1/: J. Duley, D. Dietmeyer: "A digital systems design language (DDL)" IEEE ToC, Vol. C17, # 9 (1968)

/HO1/: C. A. R. Hoare: "Monitors: An operating system structuring concept" CACM, Vol. 17, # 10 (1979)

/LS1/: B.Liskov, A.Snyder. R.Atkinson, C.Schaffert: "Abstraction mechanisms in CLU" CACM, Col. 20, # 8 (1977)

/PE1/: J. L. Peterson: "Petri Nets" ACM Comp. Surveys, Vol. 9 (1977)

/RA1/: F. J. Rammig: "CAP/DSDL, preliminary language reference Manual" Univ. Dortmund, Abt. Inf., Techn. Report # 129 (1982)

/WI1/: N. Wirth: "Modula: A language for modular multiprogramming" Softw. pract. Exper. Vol. 7, # 1 (1977)