ODICE: Object-Oriented Hardware Description In CAD Environment

Wolfgang Müller    Franz Rammig
Cadlab
Bahnhofstrasse 32, 4790 Paderborn, FRG

Since VLSI design can only be controlled by powerful design tools, HDLs must be redesigned to meet the new requirements.
In this paper the hardware description language ODICE which has been integrated into a CAD/CAE framework is presented. ODICE is concerned with structural, behavioral and extended generative description, which corresponds very closely to the graphical structure. It is based on the multilevel HDL DACAPO-III, which is extended by introducing the class of hardware components. This serves as a basic concept for inheritance and additional protocol description.
ODICE is mapped to an internal data model of an object-oriented framework dedicated to CAD/CAE applications. The multifunctional interface of this database allows the code to be integrated into the graphical design environment.

1 Introduction

Object-orientation is one of the most commonly used terms in recent design methodology projects, as it is tightly coupled with handling complex systems. It has been represented in software engineering methods through the use of encapsulation and information hiding, which permit a comfortable manipulation of the large amounts of data objects which often appear in CAD/CAE applications. This has been done in large software projects [1] using Smalltalk-80 [5] and recently Objective-C [2] and C++ [16]. Comparing these languages shows that they all understand abstract data types (ADT) and inheritance. The HDLs CONLAN [13], OODE [17], MoDL [15] and LOGLAN [12] make use of these concepts, too.
ODICE uses these features by mapping them to a description level which maps nicely to graphical structures. Physical components are represented by the ADT class, which contains methods describing the behavior of the class. This leads to a HDL with separate structural and behavioral descriptions, which in this aspect is similar to VHDL [19] and ZEUS [11].
By separating structural and behavioral hardware description, the designer can directly control the physical organization and therefore the graphical representation. Furthermore this kind of description supports silicon compilation in a very effective way.
ODICE is based on the multilevel HDL DACAPO-III (CAP/DSDL) [4], which is oriented on PASCAL/MODULA-2. ODICE is a strongly typed procedural language. The basic syntax is PASCAL-like and the modularization is inherited from MODULA-2. The compound statements are extended by parallel synchronous and asynchronous control structures. The elementary data structure is a bitstring of arbitrary length. All complex data structures are based on this structure. Furthermore like DACAPO-III it distinguishes between storing (explicit) and nonstoring (implicit) objects. Implicit objects are assigned in a special implicit definition part (IMPDEF) and are evaluated continuously. ODICE is designed to be upward compatible with DACAPO-III.
The first part of this paper is an overview of the syntactical and conceptual constructs of ODICE.
The second part introduces the object-oriented Internal Data Model (IDM) of the CWS (CAD-LAB Workstation) \(^1\). The CWS is designed as an open, generic object-oriented machine for tool integration.
Finally the basic concept of the ODICE preprocessor is described, which uses the object graph of the database for code generation.

## 2 Object-Oriented Hardware Design

### 2.1 Classification of Hardware

In this chapter a possible classification of hardware components using classes is proposed. Classes correspond to structural components. A class dictionary is built up by grouping components by their functionality. The relation among classes is a static hierarchy. The design granularity increases from the root down to the leaves. The leaves represent the most detailed level and the inner nodes represent more general parts. This implies a top down refinement of classes and methods. The root is described by the metaclass, where frequently used standard methods are predefined.

As shown by the example in figure 1, ripple adder and carry lookahead adder can be grouped under the class parallel adder, where in object-oriented terms the parallel adder is the superclass.

Thus each hardware component can be classified, i.e. gates, adders, CPUs etc. Subclasses can also be built by different level descriptions of one component, i.e. switch level, algorithmic level.

This class tree thus forms a clear and flexible base for an advanced design and simulation environment. This means:

- more flexible simulation
- base for inheritance of behavior
- view similar to structured component libraries
- more structured and flexible hardware design
- support of top down and bottom up design

\(^1\) CADLAB is a research institute of NIXDORF COMPUTER AG and PADERBORN UNIVERSITY
2.2 Inheritance

HDLs like LOGLAN und CONLAN understand inheritance in the sense of Smalltalk-80 and SIMULA-67. ODICE introduces inheritance based on first order behavioral descriptions, which are available on the preliminary defined class dictionary as discussed above. In its present state ODICE does not consider multiple inheritance as each class has a unique father (see figure 1). A single class inherits methods by its superclass, its super superclass, etc. This shall be illustrated in an example referring to the class tree of figure 1. Assume the method ‘Add()’ to be defined in the class ‘adder’. ‘Add()’ could be implemented by adding two bitstrings in one statement. The method ‘Add()’ is inherited to all sons of the class ‘adder’. It is now up to the designer to decide whether he should declare the method in the class ‘ripple carry’, by refining the method using a bitwise addition or if he should simply access the method of the class ‘adder’ offered by inheritance.

To make the inheritance more flexible, friend relations are introduced. Since they make the inheritance harder to maintain they should be used less frequently. If a method of another class is designated as its friend method, only this method is inherited from the named class.

This concept based on the statical class tree leads to a flexible design environment for design and simulation of highly complex digital circuits. Stepwise refinement of the behavioral description supports top down as well as bottom up design.

2.3 Generative Design

ODICE allows generative hardware design by describing hierarchical structures in other HDLs closely attached to generic design [11, 15, 19]. Generative descriptions in ODICE are attached to structures featuring classes and linked to graphical structures. In this way it is possible to split the description into a subcomponent definition part and a subconnect declaration part, where the connections to the subcomponents and the interconnections between the subcomponents are described. Generative relations are defined in a graph in which structural dependencies are assigned according to the structure objects are stored in CAD databases. This graph is converted to generic procedures at instantiation time. The structure-graph of the ripple carry adder illustrated in figure 7 [8] is shown in figure 2. This example demonstrates the structure dependencies of an n-input adder which contains n full adders each consisting of two half adders and one or gate.
Recursive hardware descriptions, which are known from HDLs like HILL [10], construct cyclic dependency graphs. Figure 2 presents a cyclic graph referring to the graphical structure of figure 8. For motivation some recursive components are given below.

- carry lookahead adder of size $n$, consisting of one combine component and two adders of size $n/2$ [10]. (see also figure 2)
- butterfly network of size $n$, consisting of $n/2$ comparators and two butterfly networks of size $n/2$ [18].
- orthogonal tree network for sorting of size $n$, consisting of four tree networks of size $n/4$ [14].

2.4 Class Declarations

As mentioned above hardware components are established as classes and their behavior as methods. This has the following requirements:

- interface definitions (Ports)
- generative descriptions defining subcomponents and subconnections
- specification of public behavior
- specification of local behavior (explicit and implicit)
- local objects (constants, types, variables)
- local assumptions

Mapping to ODICE leads to the following syntax:

```
CLASS <class_name> <class_parameters>
SUPER <superclass_name>
FRIEND <friend_specification>

PORT <port_specification>

SUBCOMPONENT <subcomponent_list>
SUBCONNECT <subconnect_specifications>

CONST <<const_declarations>
TYPE <type_declarations>
VAR <var_declarations>
ASSERTIONS <assertion_part>
IMPDEF <implicit_part>

/private_method_declarations>
/public_method_declarations>

END_CLASS
```

SUPER and FRIEND specification-parts describe the class and inheritance relations, and the SUBCOMPONENT and SUBCONNECT parts define the generative structure. The class can be parametrized by constants and types referring to generic types of DACAPO-III (generic class). Arithmetic operations can be assigned to constants especially useful in recursive definitions like [CONST n:= n/2]. The expression is first evaluated and then assigned to the parameter as a constant. Classes need not contain any behavioral descriptions. They can also be used for building up parts of the class dictionary.
2.5 Generative Declaration

2.5.1 Simple Structures

A generative declaration is initiated by the keyword `SUBCOMPONENT` followed by a list of subcomponents each specified by name and number.

```
SUBCOMPONENT  halfadder #[2], or_gate #[i];
```

The example defines three subcomponents: 2 halfadders and 1 or-gate.

2.5.2 Recursive Structures

To describe recursive structures special syntactical constructs are provided. When defining structures the most common form of recursion may appear, such as non-linear and non-iterative. The ODICE preprocessor needs to create simple structures, hence some special syntactical and parameter restrictions are introduced with respect to transformation.

In recursive component descriptions generally constructs such as the following function definition appear:

```
h3(x) ::= if p(x)
    then c1(h1(x))
    else c2(h2(x), h3(g(x)), h3(g(x)))
```

The ‘h’ represents components, and the ‘c’ describes interconnections. This structure belongs to the graph of the carry lookahead adder of figure 2. To avoid nested recursions by syntax and to distinguish between desired and undesired cycles, the following construct is used:

```
RECEASE ( p(x) ) h1(x)
RECBODY    h2(x) #[1], h3(g(x)) #[2]
```

Problems in transforming the structure are reduced by restricting parameters to generic types inherited from DACAP0-III. Only structures are described recursively. Therefore only simple relations appear, usually expressed by strongly monotonous rising or falling functions like n*4, n/2. This makes it possible to check if the recursion might converge or not.

2.5.3 Connections

To describe interconnections in a more powerful way the SUBCONNECT part has its own metasyntax using `while`, `repeat`, `for` loops in conjunction with `if . . . then . . . else` conditions.

The connections are assigned with ‘==’ possibly affected by a special delay. Assignment to interface variables already declared in the port specifications is allowed. These variables must be prefixed by the corresponding classname. Names of subcomponents must be extended by a number uniquely identifying the instance and referring to the subcomponent definition. For instance, if two halfadders are defined, the ports of each halfadder can be referred to by using `halfadder[1]`, `halfadder[2]`, or `halfadder[i]` i ∈ {1, 2} as prefix. The interconnections are assigned within a PASCAL-like compound statement, where some variables as control variables may be declared. Following the keyword `SUBCOMPONENT` a procedure method can be declared. According to this declaration the substructure can be accessed as a public method.

Introducing this PASCAL-like metalanguage, interconnections can be described in a very efficient way with respect to generic parameters and constants. Hierarchical structures may be constructed depending on parameters like bitsize, technology and abstraction level.
begin  
  var i : integer;  
  for i:= 1 to n do  
    begin  
      if (i = 1)  
        then full_adder[i].Cin = 0;  
      if (i < n)  
        then full_adder[i+1].Cin = full_adder[i].Cout;  
      full_adder[i].x = ripple_adder.x(i-1);  
      full_adder[i].y = ripple_adder.y(i-1);  
      ripple_adder.z(i-i) = full_adder[i].z  
      if (i = n)  
        then ripple_adder.z(i) = full_adder[i].Cout  
    end  
end

2.6 Incarnations and Methods

2.6.1 Incarnations

Incarnations are handled in the variable declaration part, describing an instance as a variable of a certain class.

  <list_of_instances> : CLASS <classname> [parameter_list]

This description corresponds to the incarnation of procedures in DACAPO-III.

2.6.2 Method Declarations

Classes can contain public method declaration-parts and private ones. Private methods are declared in the same way as DACAPO-III procedures and functions, however prefixed by the keyword PRIVATE. PUBLIC declarations are similar to DACAPO-III except for the parameterlist, where parameters are simply listed. They must be declared previously as an interface variable in the port declaration part. Functions are defined without any return type. Instead of this the function name must appear in the port declarations, where the return type is assigned as an OUT port. This modification of the syntax is necessary as communication is allowed only through the port interfaces corresponding to the physical and graphical representation of a circuit.

2.6.3 Method Calls

To call a method the method-name, including parameters and the name of the instance, must be specified. Multicasting is supported by enclosing a list of instances in parenthesis. Then the given method calls are evaluated concurrently.

  <instance_name> <- <method_call>;  
  ( <instance_list> ) <- <method_call>;

3 Protocols

3.1 Protocol Definitions

With increasing complexity in PCB and IC design the importance of protocols increases in the direct proportion to the number of communicating parts.
The separation of the protocols from the other behavioral descriptions supports a more flexible simulation and design, and a correspondence to the physical realization is achieved where protocols are controlled by separated hardware. Thus ODICE allows the user to design protocols in a special protocol definition part. Protocols are bound to methods extending the method call. Multilevel protocols according to the ISO reference model are also supported. A protocol layer always represents an encapsulation of the communicating partners, eventually describing an interface to the next higher level.

ODICE uses protocols as part of a description environment. In particular, descriptions in mixed level simulation are supported adapting different protocol mechanisms of different levels. Figure 3 illustrates a simple handshake synchronization containing two partners, which are connected by the signal lines ‘request’ and ‘acknowledge’.

ODICE uses procedure-like constructs to specify a protocol. The interface is represented by the parameters. Since a complete protocol description may range from the physical layer up to the application layer, the specification must be kept general. Therefore, the ODICE protocol constructs act simply as a frame for the DACAP-III syntax. Protocols can only be bound to methods calling an instance.

All protocols between two or more partners are activated by altering states affected by control lines. After receiving a signal a partner carries out some action and responds to the sender and/or sends signals to a third partner again altering signal lines. Furthermore a protocol has to describe different roles, i.e. role of the master and slaves. Based on these considerations the following specification scheme has been defined:

```
PROTOCOL <protocol_name> ("<protocol_parameter_list>")

CONST <constant_definition_part>
TYPE <type_definition_part>
VAR <var_declaration_part>
  <routine_declaration_part>
ASSERTIONS <assertion_part>
IMPDEF <implicit_part>

ACTION <action_declaration_part>
  <partner_specification_part>

END_PROTOCOL
```

The `<action_declaration_part>` consists only of an enumeration of identifiers representing placeholders for special actions used in the `<partner_specification_part>`. The `<partner_specification_part>` consists of one or more `partner_specifications` described below.

```
PARTNER <partner_name> :  "<compound_statement>"
```
Figure 4: handshake overlaid by hdlc

The `<compound_statement>` is inherited from DACAPO-III syntax, extented by the previously mentioned actions.

3.2 Binding Protocols

Protocols are dynamically bound to method calls, encapsulating the method by the specified partner protocol. The predefined action in the protocol specification part is replaced by the method. When binding a protocol, both the protocol and the partner whose part is overtaken have to be identified. All communicating partners must be specified together:

```
USE <protocol_name>
  <object_name> <- <method_name> AS <partner_name>
  ...
END_USE
```

Multiple binding of several protocols is specified simply by naming protocols and for each method call the corresponding partners beginning from outer levels to inner ones as shown in figure 4, using ‘handshake’ overlaid by ‘hdlc’ featuring two different abstraction levels. In this case ‘sender’ and ‘receiver’ represent partners of ‘hdlc’, while ‘c1’ and ‘c2’ belong to the ‘handshake’ protocol.

```
USE hdlc USE handshake
  classA <- send(data) AS sender AS c1;
  classB <- receive(data) AS receiver AS c2
END_USE
```

4 Implementation Using the CADLAB Framework

4.1 The Object-Oriented Interface of the CWS

The CADLAB Workstation (CWS) is an open, generic system for tool integration. Within the CADLAB project a universal database interface to the CADLAB framework has been developed, called the Internal Data Model (IDM) [7, 9].

The IDM is based on its object-oriented kernel system, which serves as an efficient data handling system in main memory. The basic IDM object-types are the classes primitive and complex objects. Primitive objects serve as entities encapsulating information at the lowest level of design granularity. Within the kernel system primitive as well as complex objects come into existence as nodes of a graph whose bidirectional edges form ‘father-son’ relationships. Subgraphs of primitive objects make up a complex object. The primitive object of the standard predefined type ‘entry’ marks a unique location for entering the subgraph. The ‘instance’ object allows the creation of references
from any primitive object to other complex objects, and thus offers a general mechanism for building design hierarchies. The IDM offers a set of about eighty functions, which represent the generic functionality of the kernel system. The most important types are:

- dynamic configuration of object types
- manipulation of selected objects, including create, delete, copy
- simple navigation routines to fathers, sons or brothers
- iterative graph traversal and manipulation as specified by an imported user-function

The IDM is an universal interface which serves for integration in the entire framework.

4.2 Mapping ODICE Syntax

Since the graphics module of the CWS for rapid selection and manipulation of graphical data, the Graphical Data Structure (GDS)[6], is designed with respect to the IDM interface, a major effort was made to provide the same interface to the ODICE preprocessor as that for the graphical environment. One complex object is made for all class- and protocol-descriptions of one module. First the class tree is mapped to primitive objects representing nodes by objects typed ‘class’ and edges typed ‘tree_edges’. The generative relationships are identified as ‘gen_edges’. An incarnation creates a relation between the ‘gen_entry’ (a unique son of the predefined ‘entry’) and the specified ‘class’ object (see also figure 5).

The code is stored line by line in objects of type ‘line_of_code’, which are attached to the class object structured by constants, types, variables, assertions, private and public methods.

4.3 The Interface Between ODICE and the Graphical Environment

In the database of the CWS framework, graphical data is stored in primitive and complex objects with respect to the EDIF standard [3]. Graphical information about the components is attached to primitive objects of type ‘figuregroup’. For each figuregroup an additional son of type ‘code_instance’ is created. It refers to the referenced class object in the complex object that contains the class definitions. Through this reference the code belonging to a graphical symbol can be accessed and edited. Thus in component design the graphical and behavioral design can be tightly coupled. Furthermore, parts of the structural description can be generated automatically and interactively within the graphical environment.
4.4 Code Generation

A prototype of an ODICE preprocessor has been implemented mapping the source code to the IDM interface of the CWS framework. It uses the object graph for evaluation and creates DACAPO-III code. The IDM serves for mapping the structured code to the graphical interface and to realize dynamical evaluation such as inheritance and protocol binding. During code generation classes are transferred to DACAPO-procedures. Method calls are transformed to procedure calls and incarnations to variable declarations of the corresponding procedure type. The basic constructs of DACAPO-III are syntax checked and then handed over without modification. Figure 6 shows the functionality of the ODICE preprocessor.

5 Conclusion

This paper introduced the HDL ODICE, which may be characterized as an extension of DACAPO-III to classes, inheritance and protocol specifications. With the generative concept it supports object-oriented hardware design with the aid of component libraries. Mapping the code to the IDM interface of the CWS, the description of structures can be coupled tightly with graphical design, and behavioral descriptions may be integrated smoothly into the CAD/CAE design process. The integration of a HDL into a CAD environment using an efficient database eases the design of highly complex integrated circuits.

Acknowledgements

The authors would like to thank Klaus Gottheil, Bernd Kleinjohann and Georg Lehrenfeld for valuable discussions. Wilfried Fox, Klaus Gröning and Jürgen Strauß were of great help in the implementation of the prototype. The authors also would like to thank Charles White, who invested time in reading this article.

References


A Figures

Figure 7: ripple adder of bitsize 4
B Example of an ODICE Description

Description of a ripple carry adder of bit size 2 corresponding to figure 7.

MODULE circuit;

{------------------------------ CLASS adder -----------------------------}
CLASS adder;
SUPER class;
END_CLASS;

{------------------------------ CLASS ripple_adder ----------------------}
CLASS ripple_adder [CONST n=1];
SUPER adder;
PORT IN x, y : bit(n);
   OUT z : bit(n+1);
SUBCOMPONENT full_adder #[n];
SUBCONNECT PROCEDURE add(x, y, z);
   begin
   Var i : integer;
   for i:= 1 to n do
   begin
      if (i=1)
      then full_adder[i].Cin == "0";
      if (i < n)
      then full_adder[i+1].Cin == full_adder[i].Cout;
      full_adder[i].x == ripple_adder.x(i-1);
      full_adder[i].y == ripple_adder.y(i-1);
      ripple_adder.z(i-1) == full_adder[i].z
      if (i = n)
      then ripple_adder.z(i) == full_adder[i].Cout
      end
   end
END_CLASS;

{------------------------------ CLASS comb_components ------------------}
CLASS comb_components;
SUPER CLASS;
END_CLASS;

{------------------------------ CLASS full_adder ------------------------}
CLASS full_adder;
SUPER comb_components;
PORT  IN  x, y, Cin : bit;
  OUT z, Cout : bit;
SUBLAYER  half_adder #[2], or_gate #[1];
SUBCONNECT  begin
  half_adder[i].a == full_adder.x;
  half_adder[i].b == full_adder.y;
  half_adder[2].a == full_adder.Cin;
  half_adder[2].b == half_adder[i].sum;
  or_gate[i].i1 == half_adder[i].carry;
  or_gate[i].i2 == half_adder[2].carry;
  full_adder.Cout == or_gate[i].output;
  full_adder.z == half_adder[2].sum;
end
ENDCLASS;

{--------------------------- CLASS half_adder ---------------------------}
CLASS  half_adder;
SUPER  comb_components;
PORT  IN  a, b : bit;
  OUT sum, carry : bit;
SUBLAYER  xor_gate #[1], and_gate #[1];
SUBCONNECT  begin
  xor_gate[i].i1 == half_adder.a;
  and_gate[i].i1 == half_adder.a;
  xor_gate[i].i2 == half_adder.b;
  and_gate[i].i2 == half_adder.b;
  half_adder.sum == xor_gate[i].output;
  half_adder.carry == and_gate[i].output;
end
ENDCLASS;

{--------------------------- CLASS gates -------------------------------}
CLASS  gates;
SUPER  CLASS;
ENDCLASS;

{--------------------------- CLASS or_gate -----------------------------}
CLASS  or_gate;
SUPER  gates;
PORT  IN  i1, i2 : bit;
  OUT output : bit;
IMPEXP  output := i1 | i2;
ENDCLASS;

{--------------------------- CLASS xor_gate -----------------------------}
CLASS  xor_gate;
SUPER  gates;
PORT  IN  i1, i2 : bit;
  OUT output : bit;
IMPEXP  output := i1 @ i2;
ENDCLASS;

{--------------------------- CLASS and_gate -----------------------------}
CLASS  and_gate;
SUPER  gates;
PORT  IN  i1, i2 : bit;
  OUT output : bit;
IMPEXP  output := i1 & i2;
ENDCLASS
{----------------------------------- Actions -----------------------------------}

VAR x, y : implicit bit(2) := "00";
  z : implicit bit(3);
  adder_1 : ripple_adder [CONST n=2];

SEQBEGIN
  x := "01";
  y := "10";
  adder_1 <- add(x, y, z);

END.
C DACAPSO-III Code Generated from the above Description

MODULE circuit;

CONST ripple_adderadd_ = i;

{------------------------------ CLASS ripple_adder -----------------------------}
TYPE ripple_adder0_ = WITH GENERIC [CONST n = i] PROCEDURE ripple_adder0_
  (IN method_select_ : bit(8);
   IN x_class_, y_class_ : implicit bit(n);
   OUT z_class_ : implicit bit(n+1));

TYPE full_adder_ = PROCEDURE full_adder_
  (IN x, y, Cin : implicit bit;
   OUT z, Cout : implicit bit);

{------------------------------ CLASS or_gate -----------------------------}
TYPE or_gate_ = PERMANENT PROCEDURE or_gate_
  (IN i1, i2 : implicit bit;
   OUT output : implicit bit);
IMPDEF output := i1 | i2;
CONBEGIN END;

{------------------------------ CLASS half_adder -----------------------------}
half_adder_ = PROCEDURE half_adder_
  (IN a, b : implicit bit;
   OUT sum, carry : implicit bit);

{------------------------------ CLASS xor_gate -----------------------------}
TYPE xor_gate_ = PERMANENT PROCEDURE xor_gate_
  (IN i1, i2 : implicit bit;
   OUT output : implicit bit);
IMPDEF output := i1 @ i2;
CONBEGIN END;

{------------------------------ CLASS and_gate -----------------------------}
and_gate_ = PERMANENT PROCEDURE and_gate_
  (IN i1, i2 : implicit bit;
   OUT output : implicit bit);
IMPDEF output := i1 & i2;
CONBEGIN END;

VAR and_gate_ : array[1:1] of and_gate_;
xor_gate_ : array[1:1] of xor_gate_;
i1xor_gate, i2xor_gate, outputxor_gate, i1and_gate, i2and_gate, outputand_gate : implicit array[1:1] of bit;
IMPDEF i1xor_gate[1] := a;
i1and_gate[1] := a;
i2xor_gate[1] := b;
i2and_gate[1] := b;
sum := outputxor_gate[1];
carry := outputand_gate[1];
CONBEGIN
  xor_gate_[1](i1xor_gate[1], i2xor_gate[1], outputxor_gate[1]);
  and_gate_[1](i1and_gate[1], i2and_gate[1], outputand_gate[1])
END;
VAR or_gate_ : array[1:1] of or_gate_;  
   half_adder_ : array[1:2] of half_adder_;  
   iior_gate, i2or_gate, outputor_gate : implicit array[1:1] of bit;  
   ahalf_adder, bhalf_adder,  
   carryhalf_adder, sumhalf_adder : implicit array[1:2] of bit;  
IMPDEF ahalf_adder[1] := x;  
   bhalf_adder[1] := y;  
   ahalf_adder[2] := Cin;  
   bhalf_adder[2] := sumhalf_adder[1];  
   iior_gate[1] := carryhalf_adder[1];  
   i2or_gate[1] := carryhalf_adder[2];  
   Cout := outputor_gate[1];  
   z := sumhalf_adder[2];  
CONBEGIN  
   half_adder_[1](ahalf_adder[1], bhalf_adder[1], sumhalf_adder[1],  
                   carryhalf_adder[1]);  
   half_adder_[2](ahalf_adder[2], bhalf_adder[2], sumhalf_adder[2],  
                   carryhalf_adder[2]);  
   or_gate_[1](iior_gate[1], i2or_gate[1], outputor_gate[1]);  
END;  

VAR full_adder_ : array [1:n] of full_adder_;  

PROCEDURE add (IN x, y : implicit bit(n);  
              OUT z : implicit bit(n+1));  
VAR Cinfull_adder, xfull_adder,  
    yfull_adder, zfull_adder,  
    Coutfull_adder : implicit array [1:n] of bit;  
IMPDEF Cinfull_adder[1] := "0";  
   xfull_adder[1] := x.(0);  
   yfull_adder[1] := y.(0);  
   z.(0) := zfull_adder[1];  
   Cinfull_adder[2] := Coutfull_adder[1];  
   xfull_adder[2] := x.(1);  
   yfull_adder[2] := y.(1);  
   z.(1) := zfull_adder[2];  
   z.(2) := Coutfull_adder[2];  
CONBEGIN  
   full_adder_[1](xfull_adder[1], yfull_adder[1], Cinfull_adder[1],  
                zfull_adder[1], Coutfull_adder[1]);  
   full_adder_[2](xfull_adder[2], yfull_adder[2], Cinfull_adder[2],  
                zfull_adder[2], Coutfull_adder[2]);  
END;  

CONBEGIN  
   if (method_select_ = ripple_adderadd_)  
      then add(x_class_, y_class_, z_class_)  
   else;  
END  

{------------------------ Actions -------------------------------}
VAR x, y : implicit bit(2) := "00";  
   z : implicit bit(3);  
   adder_1 : ripple_adder0_ WITH GENERIC [CONST n=2];  
SEQBEGIN  
   x := "01";  
   y := "10";  
   adder_1(ripple_adderadd_, x, y, z);  
END.