Object-Oriented Design of an Expandable Hardware Description Language Analyzer for a High-Level Synthesis System

Lian Yang,
ImageBuilder Software,
7300 SW Hunziker,
Portland, OR 97223.

Marek A. Perkowski, David Smith
Department of Electrical Engineering
Portland State University,
Portland, Oregon, 97207.

Abstract

The paper presents a new approach to high-level synthesis system design in which object-oriented programming techniques are used to construct an expansible Hardware Description Language(HDL) analyzer. There are several major advantages of this new methodology over the traditional top-down approaches. The object-oriented data model for high-level synthesis systems is shown to be a better way to model the high-level synthesis design entities. A formal Object Oriented Programming(OOP) model of high-level synthesis and systematic ways of expanding the system are also described. This design style influences the construction of the entire high-level synthesis system. The system has been successfully implemented in C++ and has proven to be reliable, expansible, and sufficiently fast.

1 Introduction

As the VLSI design process becomes increasingly dependent on automated tools, various high level synthesis systems, such as ADAM [16, 17], MIMOLA [10, 28], MacPitts [23], SYCO, and CMUDA have emerged in academic environments. On the other hand, Computer Aided Engineering companies introduce and constantly improve design environments that start from register-transfer, state machine or logic level and allow simulation and automatic design down to the layout level. One of the most complete and advanced of these is the design environment of Mentor Graphics [13], which currently starts from the behavioral/structural specification of a digital system under design in language M. To make such tools available to the user whose area of expertise is algorithm design rather than the hardware design, we are working on system DIADES, a high-level synthesis tool, created on top of the Mentor's design environment. This will allow for parallel programming at the micro-level, direct mapping of algorithms to hardware, usage of pre-specified user-defined modules and optimization of complex Microprogrammed Controllers (MCs) [9, 18, 19, 20, 21, 22, 27]. *ADL (Automated Design Language)* [21, 22] has been created as the behavioral/functional/structural hardware description language (HDL) of DIADES. In other words, our entire design automation system can be treated as an ADL support environment. Since the technology is constantly changing, and new design tools and fast prototyping methods are added to all levels of our design environment, there arises a need to develop an expansible high-level synthesis language that would have two levels of expansibility, one oriented towards the user, and one oriented towards the system developer/integrator. We decided that the best way to develop such a language and system is to use the object oriented methodology. Ultimately, this decision has had a profound influence on the entire DIADES system design methodology. In this paper we present the ADL Analyzer called TAG90, which compiles to an internal description form called Program Graph (P-graph) [18, 22, 27].

Although we present only a single realized implementation of an analyzer, it is our opinion that the developed by us object-oriented methodology is very general. In particular, it can be used for fast creation of expansible compilers for not only high-level synthesis languages but also other design-related languages.

Generally, each high-level synthesis system has a software module that translates its source HDL into an intermediate representation (IR). In this paper this software module is called a **Source Language Analyzer (SLA).**

Our SLA has been been written in C++, using tools Lex and Yacc. It is fully operational and has been successfully used since the Fall of 1990. It is the goal of this paper to present the advantages of our **object-oriented expansible approach to the design** of HDLs', based on the SLA concept. We will discuss a very general object-oriented formalism consisting of a notation that can be used to create expansible programs for analyzers and other high-level synthesis tools.

Most high-level synthesis systems use purely top-down design methodologies [11]. They begin by translating the behavior description into a data flow graph representation. At this point a series of transformations is applied to make the design more efficient [25]. What this approach lacks is the ability to use detailed low-level physical information when choosing a *register transfer* (RT) structure [11]. The approach also lacks expansibility which is an important feature in modern HDLs [3]. The advantages of object-oriented design techniques within the VLSI design automation area have been discussed in many

0073-1129-1/92 $3.00

papers, including [1, 2, 5, 26]. There even exists a hardware description language that is object-oriented [24]. However, all HDLs use the object-oriented data model (data base/library) at the RT level. In [7] a CAD design version control system is proposed and an object-oriented design method is used in which objects describe *designs*. The objects from [7], however, correspond to the management versions of the design results and not to the design process itself.

In this paper, the design process of the ADL Analyzer employs object-oriented (OO) techniques. The OO methodology used by us has obvious advantages over the traditional top-down design methods used in most high-level synthesis systems. They are: (1) *Better data modeling.* The *object* data model employed in our approach can simulate the hardware design entities at various design levels. In contrast, in the traditional $top-down$ approaches, the data models are black boxes, which are passive, abstract, and suffer from a lack of communication between the modules. (2) *More systematic design process.* Using this methodology, the design process and its coding process are organized in the way required by modern software engineering theory. The gap between the conceptual design and the coding is drastically reduced. (3) *Greater expansibility.* The ADL language becomes expansible and its extensions are partially under the control of the user. (4) *Better data management.* The *class lattice* (see section 4) offers better data management and information sharing. Both the low level information and the higher level constraints become accessible to all design entities.

Language extensibility of a HDL is an important feature in designing a high-level synthesis system. In [3], language extensibility is regarded as one of the most important features for a *hardware description language,* and one of the goals of the **VHDL** (Very High Speed Integrated Circuit Hardware Description Language) language design.

This paper has the following organization. In section 2 we discuss the salient features of the ADL language and the general architecture underlying ADL program compilation. The architectural description is given, which separates the control flow from the data flow. In section 3 the process of analyzing the ADL language using object-oriented methodology and the formal models of *object* and *object transformation* are presented. In section 4 the language extensibility of ADL and TAG90 is discussed in detail. Two kinds of extension processes are investigated, and a user-controlled extension of ADL is proposed as a new feature in high-level synthesis. Section 5 presents conclusions and future work.

2 The Target Architecture of Our System

In this section the underlying architecture of the ADL is analyzed. The basic concept of DIADES is that of a **A DIADES Digital System (DDS)** which is based on a *Glushkov model* of a digital system. The task of a DDS is carried out by two main units: the data path (DP) and the control unit (CU). The design is specified in terms of a hierarchy of DDS units, described in respective ADL programs, and next automatically synthesized and optimized independently by DIADES. Most current high-level synthesis systems use similar models [4, 6, 12]. ADL is a C-like HDL. Its program consists of statements, which are behavioral, functional or structural. Each statement contains two facets: a node of the *control flow* and a node of the *data flow.* In this paper, the control flow node and its arc are represented by the *micro instruction (MI);* the data flow node and its arc are represented by the *micro operation (MO).* The list of the micro instructions represents the control flow and the list of the micro operations represents the data flow.

An **MI** is defined as a 6-tuple $< ID, TY, NE, BR, S, TI >$, where **ID** is the label of the MI; **TY** is the addressing type of the MI; **NE** is the label of the successive MI ; **BR** is used only for conditional type MI, representing the label of the branch MI; **S** is the set of the MOs signaled by this MI; and **TI** represents the execution time of the MI. In the above definition, TY can be one of: *seq* - sequential type; *con* - conditional type; or *jum* - jump type. S is a set that could be NIL, or of one or more elements; TI is an integer that represents the time unit. An **MO** is defined as a 5-tuple $< ID, OP, S, \{I\}, \{O\} >$, where **ID** is the label of the MO; **OP** is the operator mnemonic; **S** is the label of the MI that controls this MO; **I** is the set of source elements; **O** is the set of destination elements. In ADL there are four kinds of operators for OP: Arithmetic Operators: +, -, *, / ; Logical Operators: AND, EXOR, NAND, OR, NOR, XOR, XNOR, and NOT; Relational Operators: $>, <, =, \neq, \geq, \leq$; and Built-in Blocks of the **ASL** (ADL System Library): mathematical functions, parallel programming primitives of message-passing semantics, and some other complex operation blocks [22]. In the above MO definition, *the source elements* and *the destination elements* are mentioned. An element is a storage unit. A storage element in TAG90 is represented by **VAR** (variable), which is a 4-tuple: $< MN, TY, SC, BL >$, where **NM** is the mnemonic of a storage element; **TY** is the type of the storage element, including *int* (integer) and *logic* (logical); **SC** is the scope of the storage element, inlcuding *input*, *intern*, and *output*; and **BL** is the bit length of the storage element. Each ADL statement actually contains an MI and an MO. Both MI and MO are machine (target code) independent, thus are suitable for conceptual intermediate representation of CU and DP.

The P-graph based on the MI-MO-VAR model is a result of compilation. Next it is subjected to several transformations: First, finite state machines (FSMs), microprogrammed controllers (MCs), and parallel controllers are created from the P-graph. Second, they are optimized and partially verified. Third, a data path built from standard modules is created, its M-language structural specification is generated, as well as the description of logic functions and finite state machines for further design [18, 19, 20, 21].

3 Applying OOP to High-Level Synthesis

3.1 Object and Its Formal Models

In object-oriented systems and languages, any real-world entity is uniformly modeled as an **object.** Furthermore, an unique identifier is associated with each object. The object identifier is used to pinpoint an object to retrieve. Every object has a state and a behavior. The *state of an object* is the set of values for the attributes of the object, and the *behavior of an object* is the set of methods (program code) which operate on the state of the object. A **class** is specified as a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class. An instance of class C can be called a *C-object.* The notation will be used through the rest of the paper. For an object o, the notation $Class(o)$ denotes the corresponding class of o.

Methods are implementations of operations on the instances of a class. There are two kinds of methods: (1) A **procedure** performs an action which may change the state of an object. (2) A **function** computes some value deduced from the state of the object.

A special kind of a procedure, called a **constructor,** constructs a class object from the data it needs.

3.2 ADL Object and Its Formal Models

In high-level synthesis, most of the object-oriented data models are at the RT level. Lipsett [8] gives the description of the *design entity,* the principal hardware abstraction in VHDL. Briefly, a design entity concept provides for an effective separation of interface and function, thus allowing hierarchical design decomposition. It is our claim that the object-oriented data model is the best candidate to implement the design entity model.

An ADL Object (AO) is a design entity that a DDS is composed of. The AO is an abstraction of a collection of logically related aggregates of data, without regard to their internal structure. There exists several relationships between AOs. Some operations are also defined to transform the states of AOs. An *object* will refer to the *ADL Object,* for the sake of convenience. The object is a three-tuple $< I, C, S >$, where I denotes the *identifier* of the object, which is a unique symbol for all objects; C denotes the class type to which the object belongs; and S denotes the state of the object at any moment. The set of possible states of an object o (an object whose *identifier* is o) is denoted as $S(o)$, where

$$S(o) = \{si \mid si \text{ is a state of } o\}.$$

The set of states of object o will be denoted as $S(A)$, if o is an instance of the class A. Actually the notion $S(A)$ is more general.

3.3 The Relationship Between Object Classes

We will introduce four kinds of relationships: *inherit-from, composed-of, supervised-by,* and *peer-with.*

Inherit-from. The relation *inherit-from* is denoted as $A \subset B$, where B is a super class of A or A is derived from B. In the inheritance relation, a class that inherits (directly or indirectly) from class C is said to be a *descendant* of C. C is said to be the *ancestor* of its *descendants.* A class is considered to be one of its own descendants. The notation *Family(C)* represents the *family of class C,*

where Family(C) = $\{A \mid A \subset C\}$.

Composed-of. An object is either **primitive** or **composite.** Primitive objects cannot be further decomposed into other objects. On the other hand, composite objects are formed from primitive and/or other composite objects. The relation *composed-of* is denoted by $A << B$, where A is an attribute of B or B contains A.

Supervised-by. The relation *supervised-by* is denoted by $B < A$, where class A is called the *supervisor* of class B and B is said to be supervised by A. In this relationship, the objects of the supervisor of a class are able to access the attribute data of the instances of the class.

Peer-with. The relation *peer-with* is denoted by $A \leftrightarrow B$, where there exists a class C so that $A \subset C$, $B \subset C$, and B and A do not have a *supervised-by* or *composed-of* relationship.

Let us now present a few examples of the above relationships. The *inherit-from* relation captures the generalization relationship between a class and its direct and indirect subclasses. For example, an FSM and an MC are two subclasses of class Control Unit (CU). The CU is the generalization of the FSM and the MC. The FSM and the MC are the specification of the CU. More importantly, the subclasses can inherit attributes and methods from their superclasses. The *composed-of* relation captures the composition relationship between a class and its attributes. For example, a digital circuit is composed of a *data path* and a *control unit.* Therefore, it is said that the *digital system* has two attributes: a *data path* and a *control unit.* The *supervised-by* relation is not mentioned by most current literature in OOP. It is, however, an important relationship between ADL classes. This relation is particularly useful in *SLA* design. For example, in TAG90, class DP and class CU are supervised by class DDS. Thus class DDS has a method to access the attributes in both classes DP and CU, which allows for the interconnection between the CU and the DP and the optimization that is related to both DP and CU. This relation is similar to the *composed-of* relation. However, there are some aspects that differentiate between these two relationships: (1) The supervisor class A of class C does not contain a *C-object* as its attribute; however, an *A-object* can access the attributes of a *C-object,* which also can be realized by the relationship $C << A$. It is obvious that the *supervised-by* relation uses less storage space than the *composed-of* relationship. (2) The *supervised-by* relationship also can be inherited through the class hierarchy as in a *composed-of* relationship, but in a much simpler way. For example, if $B < A$ and $C \subset A$ then $B < C$. This *inheritance* of *supervised-by* re-

lationship can be realized by C inheriting the methods that access the attributes of B from A. (3) The *supervised-by* relationship allows expansion of the horizontal communication among classes. This communication is very important in modeling hardware design entities. This is because the objects in hardware design, namely the design entities, are always mutually interconnected. In software design, this interconnection can be realized by defining a *port* object that supervises the interconnected objects (see Fig.1).

3.4 The Transformations of Objects

Three types of object transformations will be introduced. Two ways will be used to denote a transformation. One is the *code notation*, the other is the *semantic notation*. The code notation reflects the programming aspects of a transformation, while the semantic notation interprets the *code notation* by giving the semantic interpretation to the transformation. The three types of transformations on the object are: a *change* of a state, a *computation* of a state, and a *synthesis* of an object.

3.4.1. A Change of a State

As mentioned above, a *procedure* is an action that changes the state of an object. The *prototype of procedure* consists of three components: the *name* of the procedure, the *parameter* of the procedure and the *class* to which the procedure belongs. A procedure f of class C is denoted as $\mathbf{P}^f_C$. We assume that different procedures in the same class cannot use the same name. Therefore, the above notation represents a distinct procedure although the features of its parameter type are not specified.

The definition of procedure $\mathbf{P}^f_C$ is as follows:

$\mathbf{P}^f_C : PL \times \mathrm{S(C)} \rightarrow S(C)$;

where $\mathbf{P}$ means *procedure*, C is the class type of the procedure, f is the name of the procedure, PL is the set of parameter lists of f, and $S(C)$ is the set of all states of a *C-object*. The *code notation* of applying a procedure to a *C-object* is: $f(p)\_o$ where f is a procedure, p is a parameter array, and o is the object f is applied to. The semantic notation of the above code notation replaces f by $\mathbf{P}^f_C$, and o by $< o\ C\ n >$. Therefore, the semantic notation of the code notation $f(p)\_o$ is:

$$\mathbf{P}^f_C\ (p)\_< o\ C\ n > \qquad (1)$$

If the result of (1) needs to be illustrated, (1) can be expanded as:

$$\mathbf{P}^f_C\ (p)\_< o\ C\ n > \ = \ < o\ C\ n' > \qquad (2)$$

where '=' means 'the result is'. In (2), the result is $< o\ C\ n' >$, where n' is a state of the object o, and $n' = f(p, n)$. Formula (2) illustrates a state change $< n,\ n' >$ within *C-object* o.

3.4.2. A Computation of a State

As mentioned above, a *function* computes some value deduced from the state of the object. The *prototype of function* also consists of three factors: the *name* of the function, the *parameter* of the function and the *class* to which the function belongs. A function g of class C is denoted as $\mathbf{F}^g_C$. The definition of function $\mathbf{F}^g_C$ is as follows:

$\mathbf{F}^g_C : PL \times \mathrm{S(C)} \rightarrow V$,

where $\mathbf{F}$ denotes *function*, C is the class type of the function, g is the name of the function, PL is the set of parameter lists of g, $S(C)$ is the set of all states of a *C-object*, and

$$V = \{v_i \mid v_i = g(p,\ s_i),\ si \in S(C),\ p \in PL\}.$$

The code notation of applying a function f to a *C-object* o is denoted $g(p)\_o$, where g is a function, p is a parameter array, and o is the *identifier* of the object that g is applied to. The semantic notation of the above code notation replaces g by $\mathbf{F}^g_C$, and o by $< o\ C\ n >$. Therefore, the semantic notation of the code notation $f(p)\_o$ is:

$$\mathbf{F}^g_C\ (p)\ \_< o\ C\ n > \qquad (3)$$

If the result of (3) needs to be illustrated, (3) can be expanded as:

$$\mathbf{F}^g_C\ (p)\ \_< o\ C\ n > \ = \ v \qquad (4)$$

where '=' means 'the result is'. In (4), the result is v, where $v = g(n)$, and n is a state of the object o. Therefore, (4) shows a computation $< n,\ v >$ from a *C-object* at state n.

3.4.3. A Synthesis of An Object

A synthesis of an object is done by an object constructor. This transformation shows how an object is constructed from other objects. A transformation $\mathbf{S}^+_C$ is defined to denote the operation of *synthesis*, which constructs a *C-object*. The code notation for the synthesis transformation is

$\mathbf{S}^+_C\ (O_1,\ O_2,\ ...,\ O_i)$,

where $Class(O_n) << Class(O)$ and $1 \leq n \leq i$.

The semantic notation for the above code notation is

$\mathbf{S}^+_C\ (< o_1\ C_1\ n_1 >, < o_2\ C_2\ n_2 >, ..., < o_i\ C_i\ n_i >)$
$= < O\ C\ n >$,

where $C_n << C$ and $1 \leq n \leq i$.

This operation allows the synthesis of new objects from the existing objects. The synthesis transformation is the construction of an object; therefore, it is called the *construction transformation*. To distinguish the *construction transformation*, a function and a procedure will be called *pure transformations*.

3.4.4. Two Rules for Pure Transformations

A *pure transformation* is denoted as $\mathbf{T}$. $\mathbf{T}$ can be either a *procedure* or a *function*. $\mathbf{T}^f_C$ denotes either a procedure $\mathbf{P}^f_C$ or a function $\mathbf{F}^f_C$.

A transformation's *code notation* and its *semantic notation* can be connected by $\equiv$ in a format $cn \equiv sn$, which means the semantic notation sn interprets the meaning of the code notation cn. For example, $f(\ )\_o \ \equiv \ \mathbf{T}^f_C\ (\ )\_< o\ C\ n >$ means that

the semantic notation for transformation $f(\ )\_o$ is $\mathbf{T}^f_C\ (\ )\_< o\ C\ n >$.

There are two rules of **T**: *Polymorphism Rule* and *Supervision Rule.*

3.4.4.1. Polymorphism Rule.

Polymorphism means that a single format has different meanings. Meyer [14] gives the following definition: " "Polymorphism" means the ability to take several forms. In object-oriented programming, this refers to the ability to refer at run-time to instances of various classes".

An object variable (a variable in short) is an alias of an object. A variable can be *bound to* an object, in which case the variable is the alias of the bound object and the object is said to be the value of the variable.

There are three factors in a variable: the name, the class type, and the value. *To declare a variable* is to assign a class type to the variable. For example, a variable x declared to be of class C type is denoted as $C\ x$. Function *D_class(x)* returns the declared class type of variable x. To assign a value to a variable means to bind this variable to an object. The assignment is denoted by operator ":=". For example, to assign an object $< i\ C\ n >$ to variable x is denoted as $x := i$ or $x := < i\ C\ n >$.

(1) Polymorphic assignment rule. If $D\_class(x) = C$, then $x := < i\ B\ n >$ is a legal assignment iff $B \subset C$. This rule allows a variable to be assigned any object that is an instance of a class family. This class family is determined by the previously declared class type of this variable. When a variable is assigned an object, the class type of this object becomes the *actual type* of the variable. Therefore, the class type of variable x is not necessarily of one type but can be a set of types. The set of the class type of variable x is denoted by TYPE: $TYPE = Family(C)$ and $C = D\_class(x)$.

(2) Polymorphic transformation rule. If x is a variable, $D\_class(x) = C$, and f is either a procedure or a function, then

$f(p)\_x \equiv \mathbf{T}^f_B\ (p)\_< i\ B\ n >$,

iff:

(1) f is a *virtual* method declared in C,

(2) B $\in Family(C)$, *and*

(3) f is also declared in class B.

In the above formula, the transformation $f(p)\_x$ is dynamically interpreted according to the different binding of x. The formula shows a powerful mechanism of program expansibility.

3.4.4.2. Supervision Rule. The supervision rule for object transformations is as follows:

If $C < A$ and a *C-object* $< o\ C\ n >$ is a parameter of transformation $\mathbf{T}^f_A$ of class A, which has the semantic notation

$$\mathbf{T}^f_A\ (< o\ C\ n >)\_< o'\ A\ n' >, \tag{5}$$

then there exists a method of C, $\mathbf{T}^g_C$, which when applied to object $o\ C\ n$ produces the same result as (5). This will be denoted as:

$$\mathbf{T}^g_C\ \_< o\ C\ n > = \mathbf{T}^f_A\ (< o\ C\ n >)\_< o'\ A\ n' > \tag{6}$$

The meaning of (5) is that the supervisor class A of C has a method f to access the attributes of class C. Formula (6) means that methods f and g of class C are equivalent.

3.5 ADL Class Lattice

The design entity within the DDS is modeled as an ADL Object. In this section, the structure and organization of the classes of the ADL Objects will be presented.

3.5.1. The Root of the System

The root of the class structure is the origin of every ADL class. From the point of view of OOP, it is the ancestor class of all classes in the class structure. The root of ADL classes is denoted by **ROOT.** The relation between *ROOT* and other classes is: $C \subset ROOT$, where C is an ADL class. Therefore, any class in this structure can be viewed as the member of $Family(ROOT)$. See Fig.2.

3.5.2. The Architecture Aspect: Class System and Its Descendants

The global architecture of the DDS is predefined. This global architecture needs thus to be represented in the class structure. Class **SYSTEM** is a general class of systems incorporating the architecture of the target digital system. Class SYSTEM includes the abstract class DDS and two *main design entity classes* CU and DP. **DDS** is an abstract class reflecting the architecture of the target digital system in our system. Classes **DP** and **CU** are supervised by class DDS. The relationship between these three classes under class SYSTEM is:

$CU \leftrightarrow$ DP **AND** CU < DDS **AND** DP < DDS

Classes DP and CU contain *composite* and *primitive* design entities. The primitive design entities belong to classes under the class PRIMITIVE. Class DDS contains the global information of the DDS and has access to the data and methods in both CU and DP.

3.5.3. The Functional Aspect: Class Primitive And Its Descendants

Various design styles and design methodologies are currently used in high-level synthesis systems for the design of functional elements. Formula MI-MO-VAR introduced in this research will represent the design entities of the functional level architecture. Class **PRIMITIVE** is an abstract class dealing with design entities at a lower level compared with the classes of the SYSTEM family. Formula MI-MO-VAR corresponds to three main design entities at this level. Class VAR, MI and MO objects are able to supervise lower level information stored in the ASL through the *supervised-by* relationship between classes LIB_ELEM and PRIMITIVE. This allows the transformations and optimizations at the functional level to take into account the bottom information at an early stage. For the same reason class LIB_ELEM is also *supervised-by* class SYSTEM.

3.5.4. The Lattice Structure of Classes

Fig.2 shows the lattice diagram of ADL classes. All classes described in the above section are shown, together with the relationships between them. Usually, the *inherit-from* relation is the only relation between classes in an object-oriented programming language [15]. As a new contribution to the SLA design the relation *supervised-by* is introduced in the lattice of ADL classes. This class structure model essentially improves the expressionability of the object-oriented system that realizes the high-level synthesis system.

3.6 ADL Classes Close-Up

ADL classes are the abstract types of the ADL objects. An ADL class will be simply called a *class*. The diagram of a class is :

{class:
class-name
inherit-from:
parent-name(s)
supervised-by:
supervising-class
attributes:
(a1, a2, ..., an)
methods:
(p1, p2, ..., pi)
(f1, f2, ..., fj) }

In the above diagram, *pi* denotes a procedure and *fi* denotes a function. The following classes will be the recognized types of the *design entity* of our system (the remarks following ";" are the comments).

CLASS MI represents the *node of the control flow* of a DDS. The diagram of MI class definition is:

{ class:
MI
inherit-from:
PRIMITIVE
; inherits features from PRIMITIVE
supervised-by:
SYSTEM ; supervised by its superclass
attributes:
(id, ; the label of the MI
ty, ; the addressing type of the MI
ne, ; the label of the successive MI
br, ; the label of the branch MI
(used in conditional MI)
mo-list, ; the set of MOs signaled by this MI
ti ; the execution time of the MI)
methods:
(; list of procedures
MI-constructor, ; the constructor of MI-object
set-attributes; set the attributes of MI-object)
(; list of functions
print() ; the method that prints a MI-object in a given format) }

The diagrams of the rest of the classes will not be presented here. However, their purposes and positions in the ADL class lattice are outlined in the following.

CLASS LIB-ELEM is a pseudo class of the ASL elements. **CLASS DP** is a composite class. A DP is composed of a list of *MO*s and a hash table of *VAR*s. The procedure *compress* of DP is the major optimization transformation in the class lattice. It compresses (optimizes) the MO-list according to the dependency of data in it in order to exploit the maximal parallelism of the data flow. A DP is supervised by DDS, which means that a DDS-object has accessibility to the attributes of a DP-object. For instance, function *print-nalisset* uses attributes *mo-list* as input and gets a list in a **nalisset** format. Function *print-lzmset* uses attributes *var-table* as input and gets a list in **lzmset** format. **nalisset** and **lzmset** are P-graph lists.

CLASS CU represents the control unit of a DDS. The ADL_CU class is composed of a list of MIs. The description is similar to that of a microprogrammed control unit, which is composed of *micro-instructions*. The procedure *compress* is an optimization transformation on CU-objects. The algorithm for *compress* used here is similar to the scheduling algorithms used for the compaction of microprograms. CU is supervised by DDS, which means a DDS-object has accessibility to the attributes of CU. Function *print-coplisset* uses *mi-list* as input and gets a list in **coplisset** format. List **coplisset* is the control flow list for the P-graph.

CLASS DDS is the supervisor class of class DP and CU. It does not contain a DP-object or a CU-object. However, a DDS-object has accessibility to the attributes of both DP and CU objects. DDS is the supervisor for classes DP and CU. It is actually an abstract class for a Diades Digital System. Since a DDS-object has the right to access all information stored in the DP-object and CU-object, it is able to generate an IR according to the predefined format. Currently, the IR for our system is the P-graph. Therefore, function *gen_p_graph* is defined to generate the P-graph. By accessing the methods of DP and CU, the lists **nalisset**, **lzmset**, **coplisset**, and **nolisset** of the P-graph are generated. The P-graph list of predicates, **plisset**, needs information in both DP and CU. Therefore, in function *gen_p_graph*, code for generating **plisset** is specified.

The **CLASS SYSTEM** is the super class of several composite classes. It supervises the classes DP, CU, and DDS. There are two purposes for this class: to perform some global operations and optimizations and to make possible the incorporation of new technologies into DIADES. SYSTEM is the abstract class representing the architectural features of the *target digital system* to be designed. A SYSTEM-object is constructed before the construction of any other objects. The attribute *constraint-list* of the SYSTEM is derived from the *compiler macro* defined by the user. The function *optimization* is a *virtual* function for global system optimization based on the *constraint-list*. This function is not specified in SYSTEM, but inherited and implemented by its descendants, such as the DDS. System contains some generic functions which are further defined by the descendant classes. PRIMITIVE, MI, MO, and VAR [27] were described previously.

The lattice structure of classes is the crucial asset of the system outlined here. This structure offers the following advantages : (1) *Better conceptual modeling.* Since conceptual hierarchies are very common in digital system, direct modeling of such hierarchies makes the conceptual structure of DDS easier to com-

prehend. (2) *Factorization.* Inheritance allows the common properties of classes to be "factorized" -- that is, described only once and re-used when needed. For example, all SYSTEM family members share the properties of class SYSTEM. The redundancy of description is avoided. (3) *Polymorphism.* The hierarchical organization of the ADL classes provides a basis for introduction of polymorphism (in the sense that the same function name may bind to different code when applied to different objects at run time, and a procedure with a formal parameter of class C will accept any instance of the C family as an actual parameter. The uniform storage of objects is a good example of polymorphism. (4) *Stepwise refinement in design and verification.* Inheritance hierarchies support a technique where the most general classes containing common properties of different classes are designed and verified first, and then more specific classes are developed top-down by adding more details to the existing classes. This feature makes TAG90 easy to expand and refine.

3.7 Object-Oriented Approach in High-Level Synthesis System Design

As a summary of this section, the methodology used in our design is characterized as a different approach to a high-level synthesis system.

The approach of the TAG90 system to construct the *translator* stage of the high-level synthesis system is shown in Fig.3. The ADL Analyzer and its relationships with other entities are shown.

The traditional top-down approach to SLA design has the following drawbacks: (1) *Lack of low level information.* At the *translator* stage (which corresponds to our ADL Analyzer shown in Fig.3), the computational elements are simply pieces of passive data representing fixed black boxes, with certain functional capabilities and certain abstract costs. The decisions about what physical modules are to be used and how they are to be placed are deferred until after the RT-level structure has been set. (2) *Lack of expansibility.* The *translator* is usually *procedure oriented* and the *IR* generated by it is procedure- and technology- dependent. Therefore, it is difficult to introduce new technology into the system. A slight change in either the behavioral description format or the IR format usually means a great software alteration within the *translator*.

In contrast, the approach of the TAG90 system uses *active code* (ADL objects) to represent the design entities. The approach is object-oriented, allowing easy expansibility and maintainability.

In Fig.3 the the ADL analyzer is illustrated by the box that contains the *class lattice* and the *object manager*. Therefore, our approach, called *Program-Class-Manager,* can be viewed as the pair PCM = $< L, M >$, where L represents the class lattice and M represents the object manager. *The Class Lattice* consists of the ADL class definition and the corresponding method definition. The design entities of DIADES are established by setting up the ADL class lattice. A set of transformations on the ADL objects is designed by defining the methods of the ADL classes. The constructions and transformations of the objects are controlled by the *Object Manager.* The *Object Manager* is like a post office sending messages to invoke transformations on the elements of the object lattice. The *Object Manager* consists of semantic routines attached to the Yacc grammar rules. The function of the object manager is to manage the process of ADL object construction and transformation. The construction of an ADL object is conducted by the *constructor* method associated with the object's class. the class whose instance is to be constructed. The transformations on the ADL objects are performed by the class methods that are defined in the *class lattice.* The *Object Manager* invokes a class method by sending a *message* to an object or several objects.

3.8 The Advantages of PCM

Using the PCM model, the design tasks are distributed among different object classes. The data and methods are connected and organized through *the class lattice* as explained in ht eprevious section.

The advantages of this new methodology are the following: **(a)** The *object* can simulate the *design entity* at various design levels. The object encapsulates data and methods, thus becoming an *active* design *subject* instead of a traditional data model, which is a passive, abstract, and separated *black box.* **(b)** The design process and the coding process are organized in the way modern software engineering theory requires. Actually, the L (the *class lattice* presented in this section) is very close to the actual code of the PCM. Similarly, M (the *Object Manager*) consists of modular semantic routines attached to standard grammar rules of the Yacc specification file. **(c)** The greatly improved expansibility is realized (see details in the next section). **(d)** Low level information is accessible at the PCM level. The *supervision* relation between objects makes it possible for the objects at the higher level to access the lower level information through the class lattice and the ASL. The *supervision* relationship between the objects is a new creation in the concept of OOP.

4 The Expansibility of ADL

Two kinds of expansion process should be distinguished. **System level expansion** of a software system involves modifications, additions, or deletions of the internal data structures and procedures. This kind of expansion process has to be performed by the expert who knows the internal implementation of the current software system. On the other hand, **the user level expansion** of a software system does not change the internal implementations of the objects, but simply adds new knowledge to the knowledge base of the system. This expansion process can be controlled by the user through the use of simple commands. Both kinds of expansion processes require recompilation of the system's compiler. The two kinds of ADL expansion process are shown in Fig.4. Conventionally, *the expansion of a system* refers to the system level expansion, shown in Fig.4a. Additionally, the *user-controlled expansion* method proposed here allows the user to control and manage the expansion of ADL (Fig.4b).

4.1 The System Expansion of ADL

In PCM, the generation and manipulation of a single object has three steps: (1) An object is constructed; (2) The object is put into a stack called *aos*; and (3) the object is fetched from the stack and is manipulated. Assume that the methods for the *aos* are the procedure *put* and the function *get*. The *virtual method* defined in class PRIMITIVE to manipulate an instance of Family(PRIMITIVE) is called *manipulate*. Method *manipulate* is also included in each member of the Family(PRIMITIVE). Assuming $D\_class(x) = PRIMITIVE$ and p is the list of primitive objects needed for the generation of x, the *code notation* for the object generation and manipulation process is

step 1: $x := \mathbf{S}_C^+(p)$
step 2: $put(x)\_aos$
step 3: $x = get()\_aos$
$manipulate()\_x$

At step 1, an object variable x is bound to a *C-object*, where $C \subset PRIMITIVE$. This is a legal assignment according to the *polymorphic assignment rule*. At step 2, x, which represents a set of objects

$$x = \{o \mid o =< i\ C\ n > \textbf{ and } C \in Family(PRIMITIVE)\}$$

is stored into the *aos*. At step 3, $manipulate(\ )\_x$ is interpreted as the following *semantic notation* according to the *polymorphic transformation rule*:

$$\mathbf{P}_C^{manipulate}(\ )\_< i\ C\ n > = < i\ C\ n' >$$
$$iff \quad C \in Family(PRIMITIVE).$$

The ADL object manipulation process at step 3 is polymorphic. It dynamically binds the manipulation operation to a different implementation; the exact manipulation method depends on the *actual type of x*. The program for ADL object manipulation will not be changed if new ADL classes are added as descendants of Class PRIMITIVE.

4.2 User-Controlled Expansion of ADL

Relevant research on organizing digital devices into an object-oriented library can be found in [2, 26]. The *ASL* is the knowledge base of TAG90. The *user controlled expansion of ADL* adds knowledge into the ASL through the user interface. A DDS contains digital resources. An ADL program describes the resource elements and connections between them. There exist two kinds of resource elements: *storage elements* and *functional elements*. An atomic ADL operation is represented by the triple $< F\ OP1\ OP2 >$, where F is the functional element of the operation; and $OP1$ and $OP2$ are two storage resources which are the input of F. In current ADL, F must be one of the following operators: +, -, *, or /, $OP1$ and $OP2$ must be one of the variables declared in the declaration section of the ADL program. In a practical application, *F, OP1, and OP2* can be any kind of variables, blocks, or structures.

As VLSI technology improves, new storage elements and functional units come out. Some sophisticated user-defined ADL systems and blocks are very useful and should be made reusable. ASL serves as the warehouse of these new elements.

There are currently two basic ASL items: **function** and **resource.** ASL divides its items into different items. Each *item* contains one or more elements, which are the instances of the item. For each element in the data base, there is a wealth of information available. The related information could include size, aspect ratio, power, pin count and pin definitions, delays, input impedance, output drive, clocking requirements, and functions performed [11].

Let us discuss now the *subclass of F_ELEM.* An *F_ELEM-object* F_i represents conventional function calls in ADL. Actually, any ADL statement can be represented in such a function call. For example:

```
a = add (x, y); // represents a = x + y;
a = sin (x) ;
a = a_block(x, y, z)
    // represents the call to a block called 'a_block'.
```

In the ADL program, if a *block* is called in the algorithm and the *block* is not defined, the block may be an ASL functional element. Therefore, TAG90 should be able to look up the called *block* in the ASL and verify that the ASL item exists and is appropriately used. Additionally, the ASL should be able to be expanded by the ADL user, so that new technologies and mature design methodologies can be easily incorporated into the system.

4.3 Adding a New F_ELEM-OBJECT Into ASL

Whenever a new ASL element is defined, some new ADL statements become legal and the ASL and TAG90 need to be expanded. In the ASL, all instances of the *F_ELEM* family are stored in one hash table. The hash table for *F_ELEM* is called **F_Hash_Table.**

To add a new *F_ELEM-object* into the ASL is to store this object into the F_Hash_Table. For this purpose, a separate file called ADL Extension Format (AEF) is defined to process the addition of a new *F_ELEM-object.* An *AEF* consists of user defined *macros*. The format of the macro is *F_ext(name, op_number, op_type, return_type, time, implm)* where *name* is a character string representing the mnemonic of the new functional element; *op_number* is an integer representing the number of parameters of the new functional element; *op_type* is an integer vector indicating the types of the parameters. The length of the vector equals *op_number*. In ADL 1 indicates type *int*, 2 indicates type *float*, and 3 indicates type *logical*. *return_type* is an integer indicating the type of the return value of the new functional element. *time* is an integer indicating the average execution time of the functional element. *implm* is a function that describes the implementation details of the new functional element.

The macros are transformed by the C preprocessor into *F_ELEM-object* constructors. When the AEF and the main program are compiled and linked together, the ASL *F_Hash_Table* is built. Following construction, the *F_ELEM* instances place themselves into the *F_Hash_Table.* In an OOP language, an object is able to control its own behavior, such as placing itself into a hash table, by manipulating the pointer *this*, which

points to the object itself at run time. After the AEF and the TAG90 main program are compiled together, the new TAG90, which is able to interpret the new functional elements, is formed.

4.4 Using ASL During The ADL Analyzing Process

When an ADL program is compiled, the functional elements in the source ADL program are frequently checked to see if they are defined in the ASL. This process is called **ASL checking.** The class *F_ELEM* method *check_usage* checks the legal usage of an *F_ELEM* element. The instances of *F_ELEM* are stored in *F_Hash_Table* according to their mnemonic names. An *F_ELEM* object can thus be found from the *F_Hash_Table* by its name, then the *F_ELEM* method *check_usage* is then applied to the instance to verify correct use.

5 Conclusion And Future Work

The object-oriented C++ SLA for the ADL language has been implemented as the input to the design automation system DIADES. It is fully operational. We found the design methodology presented here to be very useful -- we were able to expand the language in a short time using this approach. For instance, the parallel primitives FORK, DAND and DEXOR were added in only few days. Moreover, we find this methodology to be applicable in other high- and logic-level synthesis programs as well. Since ADL is a wide-spectrum language that includes properties of many well-known HDLs, it seems that similar approaches can be used to write SLAs for VHDL and new high-level, system-oriented descriptive languages.

Currently, only digital circuits can be described. It is planned, however, to expand ADL to analog circuits as well. Since the syntax of many statements will be the same or similar, and the IR constructs will often be similar, implementation of analog SLA will become a real test of the approach presented above.

Extensive literature research reveals that very little has been published in this area, and that in the **SLA** design of *high-level synthesis systems* there are still many issues that need to be investigated. The advantages of OOP on TAG90 design process, which the authors learned to appreciate during the design process, can be summarized as follows: the complexity of TAG90 design has been tremendously reduced; the expansibility of the TAG90 has been displayed in the stepwise refinement of the TAG90 itself; and the simplicity of the expansion of the TAG90 system was found to be very impressive.

In our opinion the methodologies employed in the TAG90 design are expected to transcend the traditional *top-down* design strategies of HDL compilers.

We see the following immediate research possibilities of an OOP approach to DIADES: **(1)** Adding OOP language features into ADL. So far, ADL is not an OOP language. The OOP implementation of the TAG90 is thus not very straightforward and natural. If ADL is expanded or even redefined as an OOP language, some new and creative features of our entire system will have to be realized. For example, the new ADL could include user defined class types. Adding new methods to ADL classes. Currently, the methods defined for the ADL classes are limited to the formatting and data initialization aspects. More sophisticated *procedures* and *functions* related to the *optimization*, *parallelism*, and *communication* between objects are required [22]. **(2)** Increased understanding of the role of the objects. Currently the concept of the object is limited to that found in object-oriented programming languages, which package operations for data manipulation with the data itself. To model the hardware *design entities*, a more sophisticated object model will need to be established in the future. More precise definitions and implementations of *CAD objects* need to be found. Some help may be found by studying object oriented CAD data modelling discussed in [7].

Acknowledgment

The authors would like to thank very much the anonymous Referee Number 81 for very careful reading of the manuscript and remarks that helped us to improve the paper.

References

[1] H. Afsarmanesh et al. "An Extensible Object-Oriented Approach to Databases for VLSI/CAD," *Proc. 11th Intl. Conf. Very Large Databases,* Aug. 1985.

[2] K.E. Ayers, "An Object-Oriented Logic Simulator", *Dr. Dobb's Softw Tools,* Vol.14, No.12, pp.72-75-6, Dec. 1989.

[3] J.H. Aylor, et al, "VHDL - Feature Description and Analysis", *IEEE Design and Test,* April 1986.

[4] G. Goossens, et al, "An Efficient Microcode Compiler for Application Specific DSP Processors" *IEEE Trans. on CAD,* Vol.9, No.9, Sept. 1990.

[5] R. Gupta, et al, "An Object-Oriented VLSI CAD Framework", *IEEE Computer,* May 1989.

[6] L.J. Hafer and A. Parker, "Automated Synthesis of Digital Hardware", *IEEE Trans. Comput.,* Vol.C-31, pp.33-43, Febr. 1982.

[7] R.H. Katz, et al, and V. Trijanto, "Design Version Management", *IEEE Design Test,* 1987, pp.12 - 22.

[8] R. Lipsett, et al, "VHDL-The language", *IEEE Design and Test,* Vol.3, pp.28-41, April 1986.

[9] J. Liu, "A Finite State Machine Synthesizer", *M.S. Thesis,* Depat. EE, PSU, 1989.

[10] P. Marwedel, "The MIMOLA Design System: Detailed Description of the Software System." *Proc. 16th DAC,* pp.59-63, 1979.

[11] M.C. McFarland and T.J. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis", *IEEE Trans. CAD,* Vol.9, No.9, pp.938-50, Sept. 1990.

[12] M.C. McFarland and A.C. Parker, "An Abstract Model of Behavior for Hardware Descriptions", *IEEE Trans. Comput.*, Vol.C-32, pp.621-631, July 1983.

[13] *Mentor Graphics, Inc.*, Manuals of Genesil, GDT, Lsim, Genesil Compiler Libraries, 1988-91.

[14] B. Meyer, "Object-oriented Software Construction", *Prentice Hall Intern.*, 1988.

[15] H. Mili, et al, "An Object-Oriented Model Based on Relations", *The Journal of Systems and Software,* Vol.12, No.2, pp.139-155, May 1990.

[16] N. Park and A. Parker, "Sehwa: A Software Package for Synthesis of Digital Hardware from Behavioral Specifications", *IEEE Trans. CAD,* March 1988.

[17] A. Parker, et al, "MAHA: A Program for Data Path synthesisS, *Proc. 23th DAC,* 1986, pp.461-466.

[18] M.A. Perkowski, et al, "DIADES - A High Level Synthesis System", *Proc. ISCAS,* Portland, 1989, pp. 1895-1898.

[19] M.A. Perkowski, et al, "Integration of Logic Synthesis and High-Level Synthesis into the DIADES Design Automation System", *Proc. ISCAS*, Portland, 1989, pp.748-751.

[20] M.A. Perkowski, and J. Liu, "Generation of Finite State Machines from Parallel Program Graphs in DIADES", *Proc. ISCAS,* New Orleans, 1990, pp. 1139-1142.

[21] D. Smith, "ADL, a Behavioral Description Language", *Report,* PSU EE Dept., 1988.

[22] D. Smith, *Forthcoming Ph.D. Dissertation,* PSU EE Dept., 1992.

[23] J.R. Southard, "MacPitts: An Approach to Silicon Compilation", *Computer,* Vol. 16, No. 12, pp. 74-82, December 1983.

[24] A. Sugimoto, et al, "An Object-Oriented Visual Simulator for Microprogram Development", in *Proc. 25th DAC,* pp. 138-144, June 1988.

[25] H.W. Trickey, "Compiling Pascal Programs into Silicon," *Ph.D. Dissertation,* Stanford Univ., July 1985.

[26] W.H. Wolf, "Fred: A Procedural Data Base for VLSI design", *Proc. 23th DAC,* 1986.

[27] L. Yang, "The Object-Oriented Design of a Hardware Description Language for the DIADES Silicon Compiler System", *M.S. Thesis,* Dept. EE, PSU, 1990.

[28] G. Zimmerman, "The MIMOLA Design System: A Computer-Aided Digital Processor Design Method", *Proc. 16th DAC,* 1979.

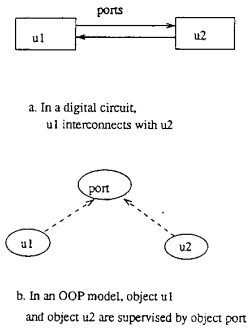

Figure 1. The Supervised_by Relationship in the Hardware Design

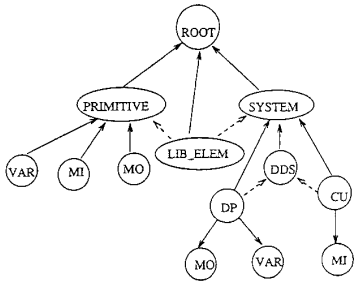

Figure 2. The lattice of ADL classes

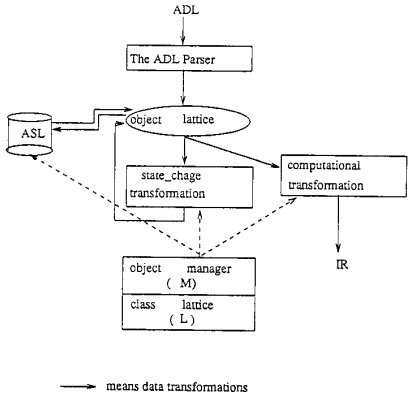

Figure 3. The Diagram of ADL Analyser

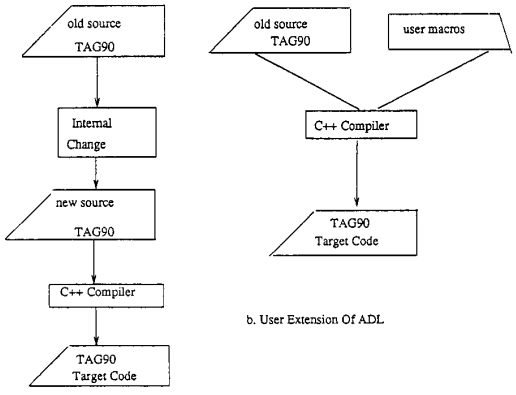

Figure 4. Two kinds of ADL extension process