UC Irvine
ICS Technical Reports

Title

BDEF : the behavioral design data exchange format

Permalink

https://escholarship.org/uc/item/43238366

Authors

Rundensteiner, Elke A.
Gajski, Daniel D.

Publication Date

1991

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Z
699
C3
no.

BDEF: THE BEHAVIORAL DESIGN DATA EXCHANGE FORMAT

Elke A. Rundensteiner and Daniel D. Gajski

Information and Computer Science Department
University of California, Irvine

April/1991

Technical Report 91-34

BDEF: THE BEHAVIORAL DESIGN DATA EXCHANGE FORMAT

Elke A. Rundensteiner and Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
April, 1991

ABSTRACT

BDDB is a Behavioral Design Data Base that manages the *design data* produced and consumed by different behavioral synthesis tools. These different design tools retrieve design data from BDDB, manipulate the data, and then store the results back into the data base. BDDB thus needs to address the following two issues: (1) a design data exchange approach and (2) customized design data interfaces. To address the first issue, we have developed a textual description format for describing design data objects and relationships. This language, referred to as the Behavioral Design Data Exchange Format (BDEF), is used as common format for exchanging design data between BDDB and the design tools in the behavioral synthesis environment. To address the second issue, we have developed a behavioral object type description language (generally referred to as schema definition language) for describing the global data structures required by design tools as well as the *desired design subviews* of this global BDDB design information. One *design view class*, namely, BDEF, is the topic of this report.

In this report we give a formal definition of the BDEF format. Then we describe a comprehensive example of applying BDEF to the behavioral synthesis domain. That is, we present the complete BDEF syntax for the Extended Control/Data Flow Graph Model (ECDFG), which is the design representation model used by most behavioral synthesis tools in the UCI CADLAB synthesis system. We also present several example descriptions of designs using this ECDFG model. A parser/graph compiler from BDEF into the generalized ECDFG design representation as well as a BDEF generator from the ECDFG data structures into the BDEF format have been implemented.

Key Words: Shared design data, Common exchange format, Design data base, Design Data Exchange, Design Data Representation.

Contents

1 INTRODUCTION **2**

2 BDDB DESIGN VIEWS **6**

3 The BDDB DESIGN OBJECT MODEL **11**

3.1 The BDDB Design Object Model 11

3.2 The Behavioral Object Type Definition Language 13

4 THE BDEF FORMAT **19**

4.1 Overview over the BDEF Syntax 19

4.2 Using BDEF to Represent Design Entity Information 22

4.3 Using BDEF to Represent Design Data Information 24

5 BDEF DESIGN VIEWS **29**

6 EXAMPLES **33**

6.1 A Simple Data Flow Graph Example 33

6.2 A Complete Control/Data Flow Graph Example 33

6.3 An Example using Two Design Entities 39

6.4 A Timing Constraint Example At the Control Flow Level 42

6.5 A Timing Constraint Example At the Data Flow Level 48

7 CONCLUSION **51**

8 REFERENCES **52**

A BDEF SYNTAX FOR THE ECDFG MODEL 53

A.1 BDDB Design Entity Graph Information Syntax 53

A.2 BNF Syntax Introduction 53

A.3 BDDB Design Entity Graph Information Syntax 53

A.4 BDDB State Graph Information Syntax 56

A.5 BDDB Control Flow Graph Information Syntax 59

A.6 BDDB Data Flow Graph Information Syntax 64

A.7 BDDB Timing Constraint Graph Syntax 73

A.8 BDDB Data Type Enumeration Values 76

A.9 Object References using Object Identity 82

A.10 General Constructs 84

B OBJECT TYPE DEFINITIONS FOR THE ECDFG SCHEMA 85

B.1 State Transition Graph 85

B.2 Control Flow Graph 86

B.3 Data Flow Graph 87

B.4 Timing Constraint Graph 90

C AN EXTENDED BDEF EXAMPLE DESCRIPTION 92

C.1 The VHDL Specification 92

C.2 The Data Flow View 94

C.3 The Control Flow View 105

C.4 The Complete Control/Data Flow View 108

D USER'S MANUAL FOR BDEF PARSER/GENERATOR TOOLS 109

List of Figures

1 Design Data Base Overview . . . 3

2 BDDB Design View Scheme . . . 7

3 BDEF Usage . . . 8

4 BIF Design Views . . . 9

5 BDDB Design Object Model . . . 11

6 The Syntax of the Behavioral Object Type Definition Language 14

7 The Syntax of the Behavioral Object Type Definition Language (cont.) . . . 15

8 An Example Using the Behavioral Object Type Definition Language . . . 18

9 BNF Syntax of BDEF . . . 20

10 Graphical Depiction of BDEF Syntax Tree . . . 21

11 BDEF Design Views . . . 29

12 BDEF Design View Generators . . . 31

13 Graphical Representation of a Data Flow Graph . . . 34

14 BDEF Description of the Data Flow Graph . . . 35

15 BDEF Description of the Simple Data Flow Graph (cont.) . . . 36

16 VHDL Design Specification . . . 37

17 A Graphical ECDFG Depiction of the VHDL Design Specification 38

18 VHDL Description of a Procedure Definition/Call . . . 39

19 Graphical Representation of a Procedure Definition/Call . . . 40

20 A BDEF Description of a Procedure Definition/Call Graph . . . 41

21 A Timing Constraint Specification at the CFG Level 43

22 A Detailed Timing Constraint Specification at the CFG Level . 45

23 BDEF Description of a Timing Constraint at the CFG Level . . 46

24 Timing Constraint Specification Using BDEF 47

25 A Timing Constraint Specification at the DFG Level 49

26 BDEF Description of a Timing Constraint at the DFG Level . . 50

27 A Graphical Depiction of the VHDL Design Specification 93

List of Figures

1 Design Data Base Overview 3

2 BDDB Design View Scheme 7

3 BDEF Usage 8

4 BIF Design Views 9

5 BDDB Design Object Model 11

6 The Syntax of the Behavioral Object Type Definition Language 14

7 The Syntax of the Behavioral Object Type Definition Language (cont.) 15

8 An Example Using the Behavioral Object Type Definition Language 18

9 BNF Syntax of BDEF 20

10 Graphical Depiction of BDEF Syntax Tree 21

11 BDEF Design Views 29

12 BDEF Design View Generators 31

13 Graphical Representation of a Data Flow Graph 34

14 BDEF Description of the Data Flow Graph 35

15 BDEF Description of the Simple Data Flow Graph (cont.) 36

16 VHDL Design Specification 37

17 A Graphical ECDFG Depiction of the VHDL Design Specification 38

18 VHDL Description of a Procedure Definition/Call 39

19 Graphical Representation of a Procedure Definition/Call 40

20 A BDEF Description of a Procedure Definition/Call Graph . . . 41

21 A Timing Constraint Specification at the CFG Level 43

22 A Detailed Timing Constraint Specification at the CFG Level . 45

23 BDEF Description of a Timing Constraint at the CFG Level . . 46

24 Timing Constraint Specification Using BDEF 47

25 A Timing Constraint Specification at the DFG Level 49

26 BDEF Description of a Timing Constraint at the DFG Level . . 50

27 A Graphical Depiction of the VHDL Design Specification 93

List of Figures

1 Design Data Base Overview 3

2 BDDB Design View Scheme 7

3 BDEF Usage 8

4 BIF Design Views 9

5 BDDB Design Object Model 11

6 The Syntax of the Behavioral Object Type Definition Language 14

7 The Syntax of the Behavioral Object Type Definition Language (cont.) 15

8 An Example Using the Behavioral Object Type Definition Language 18

9 BNF Syntax of BDEF 20

10 Graphical Depiction of BDEF Syntax Tree 21

11 BDEF Design Views 29

12 BDEF Design View Generators 31

13 Graphical Representation of a Data Flow Graph 34

14 BDEF Description of the Data Flow Graph 35

15 BDEF Description of the Simple Data Flow Graph (cont.) 36

16 VHDL Design Specification 37

17 A Graphical ECDFG Depiction of the VHDL Design Specification 38

18 VHDL Description of a Procedure Definition/Call 39

19 Graphical Representation of a Procedure Definition/Call 40

20 A BDEF Description of a Procedure Definition/Call Graph . . . 41

21 A Timing Constraint Specification at the CFG Level 43

22 A Detailed Timing Constraint Specification at the CFG Level . 45

23 BDEF Description of a Timing Constraint at the CFG Level . . 46

24 Timing Constraint Specification Using BDEF 47

25 A Timing Constraint Specification at the DFG Level 49

26 BDEF Description of a Timing Constraint at the DFG Level . . 50

27 A Graphical Depiction of the VHDL Design Specification 93

List of Figures

1 Design Data Base Overview 3

2 BDDB Design View Scheme 7

3 BDEF Usage 8

4 BIF Design Views 9

5 BDDB Design Object Model 11

6 The Syntax of the Behavioral Object Type Definition Language 14

7 The Syntax of the Behavioral Object Type Definition Language (cont.) 15

8 An Example Using the Behavioral Object Type Definition Language 18

9 BNF Syntax of BDEF 20

10 Graphical Depiction of BDEF Syntax Tree 21

11 BDEF Design Views 29

12 BDEF Design View Generators 31

13 Graphical Representation of a Data Flow Graph 34

14 BDEF Description of the Data Flow Graph 35

15 BDEF Description of the Simple Data Flow Graph (cont.) 36

16 VHDL Design Specification 37

17 A Graphical ECDFG Depiction of the VHDL Design Specification 38

18 VHDL Description of a Procedure Definition/Call 39

19 Graphical Representation of a Procedure Definition/Call 40

20 A BDEF Description of a Procedure Definition/Call Graph . . . 41

21 A Timing Constraint Specification at the CFG Level 43

22 A Detailed Timing Constraint Specification at the CFG Level . 45

23 BDEF Description of a Timing Constraint at the CFG Level . . 46

24 Timing Constraint Specification Using BDEF 47

25 A Timing Constraint Specification at the DFG Level 49

26 BDEF Description of a Timing Constraint at the DFG Level . . 50

27 A Graphical Depiction of the VHDL Design Specification 93

1 INTRODUCTION

In recent years, a great number of CAD tools of ever increasing sophistication have become available that automate the more difficult and time consuming parts of a design process. We are interested in incorporating these tools into an integrated design environment which would allow us to exploit the full potential of these tools. This requires the development of a design data management system that manages the diverse design tools and the design data used during the design process.

For this purpose we have developed BDDB [8], a design data base system for behavioral synthesis. BDDB not only manages the *design data* produced and consumed by different behavioral synthesis tools, but also maintains the *meta design information* relating these various chunks of design data according to semantic relationships, such as, equivalent, derivation, and hierarchy. BDDB thus forms the foundation for integrating different design tools into one cooperative CAD framework. BDDB provides a behavioral object type description language (generally referred to as schema definition language) for describing the global data structures required by design tools as well as the desired design subviews of this global schema. This behavioral object type description language is used to specify the BDDB design object model. The BDDB design object model [6] which represents a unified design representation for behavioral synthesis is composed of a conceptual graph model which captures the design entity organization, a behavioral graph model which describes the design behavior, and a structural graph model which represents the data path structures.

For the first generation of the behavioral design data base system, we envision a loosely-coupled architecture where the design data is shipped from BDDB to the design tools and back via design files. This proposed interaction scheme is shown in Figure 1. In this figure, the different types of design information are depicted by darker shading. The design information on the left hand side corresponds to local data structures supported by design tools. The design information on the right hand side corresponds to global design data structures that unify information produced and consumed by different design tools into one unified model, the BDDB design object model. In the middle of the figure then are the design files that are used to ship design data between the global BDDB Design Object Model and the different design tools.

This loosely-coupled architecture has been chosen for the following reasons. First, existing design tools need to interface with the design database without requiring a major rewrite. Therefore the required design data is passed from BDDB to the design tool via a design file giving the later complete control over

the design data. Once a design tool finishs its task, the possibly modified design data will be checked back into the global data structures maintained by BDDB. Secondly, a loosely-coupled system is needed due to the experimental nature of the design tools and the data base. At a university setting, we are faced with a diverse variety of development speeds of tools. Therefore, we need to isolate the programs from one another. In a tightly-coupled framework, changes in one application would invariably require changes to all applications that interface with the design tool.

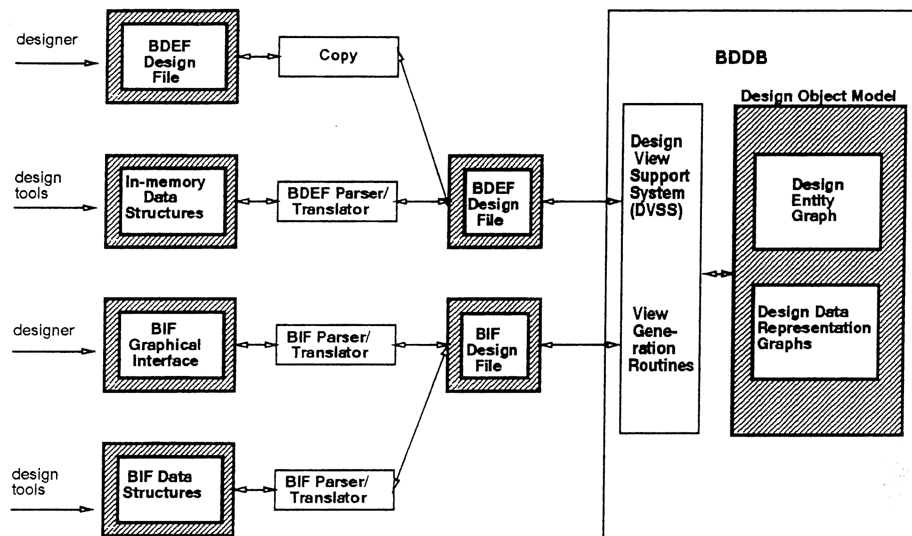

Figure 1: Design Data Base Overview

We have developed a set of rules for the textual specification of instances of design object types which have been specified by the behavioral object type description language. We propose to use this language as common format for exchanging design data between design tools in a behavioral synthesis environment. This language is thus referred to as Behavioral Design Data Exchange Format (BDEF).

As can be seen in Figure 1, BDEF is used as filter between the global data structure managed by BDDB and the local data structures maintained by design tools. Data will be restricted to flow through the predefined canonical data format. This will stabilize tool communication. Development investments to design this canonical data format and to restrict the data flow accordingly are compensated by several gains as detailed below.

First, it will simplify the creation of translators into and out of the standard text format (and the respective data structures). These translation tools will be of a similar flavor since if they would share a common file structure format. Consequently, programming efforts can be minimized by using source code from a working translator as template and by adapting it as needed by a new local data structure. Also, coupling among applications is reduced to pair-wise links from a tool's internal data structure to the canonical one and back, rather than to the data structures of all other design tools with which it shares data. A change to one application's local data organization requires changes only to one translator and not all the other translators and applications that use its design data. The possibility of data sharing is increased, as a design tool will have access to the data handled by all other design tools which have a "link" to the canonical form. Therefore, the effort for adding new tools to the system is decreased. Lastly, it is an incremental path towards closer integration of tools. Tool developers may change from their unique forms to the canonical forms at their convenience. This would allows us to switch to a tighter coupling via a programming interface in the future so that sharing of design data rather than just exchange can be accomplished.

The proposed approach towards design data exchange offers numerous advantages besides an organized way of design data exchange:

1. tool integration via an agreed upon design exchange formalism,
2. separation between the global design object types maintained by BDDB and the local data structures maintained by design tools,
3. a possibility of incremental and also manual modification of the design during all stages of the design process by changing the design file,
4. a human-readable form of the design representation (the later generally is tool-accessible),
5. a measure of consistency since all design data shipped between tools have to be cast to this common formalism, and lastly
6. a means of permanent repository of designs via the file system, if so desired.

We have found the possibility of manual modification of the design file to be very useful for the tool development phase since it allows for the isolation of the design tools from one another. A tool developer can add design information that should have been created by another design tool manually if the respective tool is not correctly working yet. For instance, state information can be added manually to a design file. Thereafter, an allocation tool can be run on this design file without having to wait for the completion of the scheduling tool. In short, a parallel tool development effort is supported.

2 BDDB DESIGN VIEWS

The design data maintained by BDDB corresponds to a global design object model, i.e., it corresponds to *all* types of design data produced and consumed by different behavioral synthesis tools. Each design tool will therefore be interested in only a *subset of the available information.* Furthermore, there generally are a number of different representation styles for one design. For instance, the usage of components in the different states of a design may be represented (1) by a state table that indicates for each state whether a component is used or (2) by a flow graph structure that has been annotated with component information. Both capture the same information content, however, they have a possibly different goal in mind and therefore are using a different data organization and representation format. For these reasons BDDB needs to provide diverse customized interfaces to the BDDB design object model. These customized interfaces, also called *design views*, generally correspond to a subset of the BDDB design information and have a possibly reorganized format and representation style.

In this section, we will outline the approach we have developed for handling design views in BDDB. We distinguish between particular *classes of design views.* Examples of potential design view classes are a textual flow graph view or a state-centered table view. Each design view class can have one or more *view parameters* that determine detailed characteristics of the representation. Once a set of parameter values has been chosen for all view parameters of a design view class, then we refer to this pair of view class and view parameter values as an instantiated *design view type.* This design view scheme is depicted in Figure 11. Clearly there is a need for view generation routines for each view class that map design data between the global information kept in the BDDB Design Object Model and the view model. These view generators are written by either a view definer or the environment administrator.

The behavioral design data base, BDDB, supports two different design view classes, namely, the BDEF design view class and the BIF design view class.

The BDEF design view class is a textual description language that directly models the graph-oriented nature of the design data stored by BDDB. Note that the design object model used by BDDB models the design specification by a set of related design objects. The BDEF format then is a direct translation from these BDDB design data object types and relationships to a textual format. Therefore, it is fairly straightforward to translate from the BDDB Design Object Model into the BDEF format, or, vice versa, to parse a design description in the BDEF format back into the BDDB data structures. Similarly, design tools

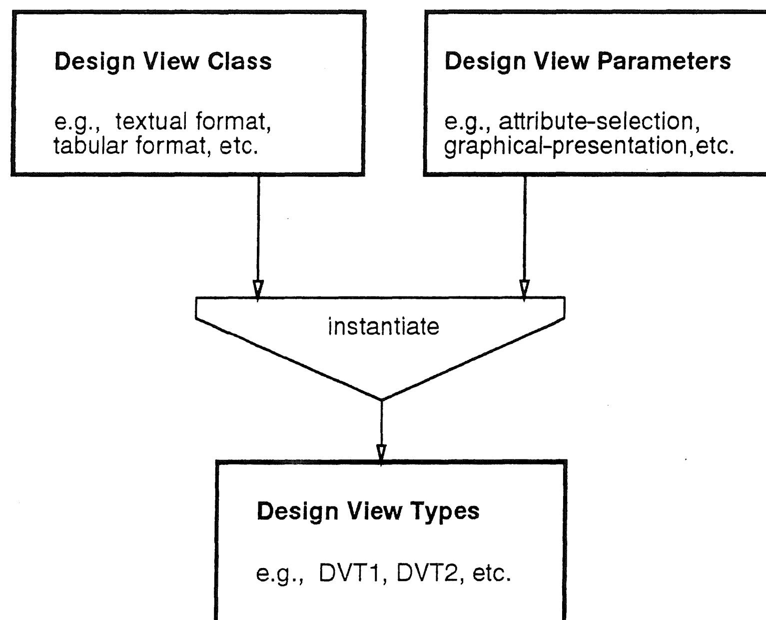

Figure 2: BDDB Design View Scheme

can easily construct their internal flow graph representation from the format. Therefore, one major criteria for the BDEF format is its 'closeness' to the design data structures utilized by these tools rather than its ease of readability by humans. The BDEF design view is the topic of this report, and therefore a more detailed discussion on the relationship between the general BDDB design view scheme introduced in this section (Figure 2 and the BDEF design view class can be found in Section 5 (in particular, Figure 11).

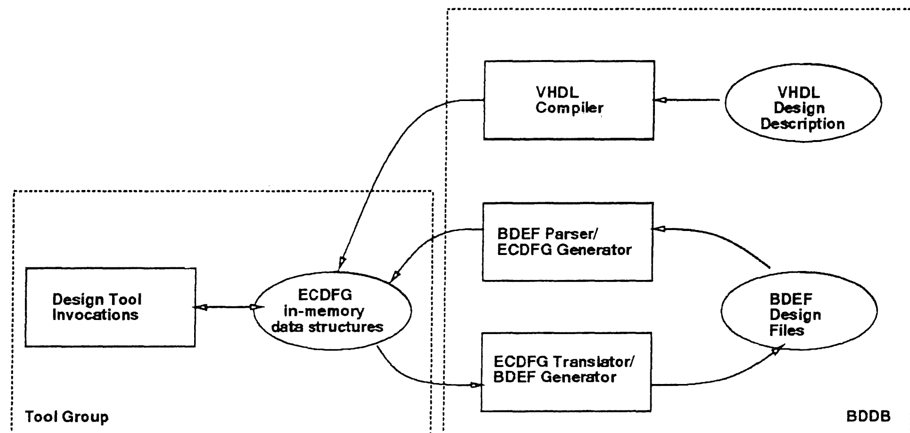

Figure 3: BDEF Usage

In Figure 3 we describe how BDEF is used in the current BDDB prototype.

The second design view class, called the BIF format, gives a state-oriented view of the BDDB design information. In other words, BIF corresponds to a tabular state table format that describes certain aspects of a design on a state-by-state basis. This state table format is what human designers are most familiar with. Indeed, a major target of BIF is ease of readability. Consequently, the BIF design view class is specially suitable for human interaction with the automated design process. It allows the designer to view and possibly manipulate the design as it evolves during the different stages of synthesis.

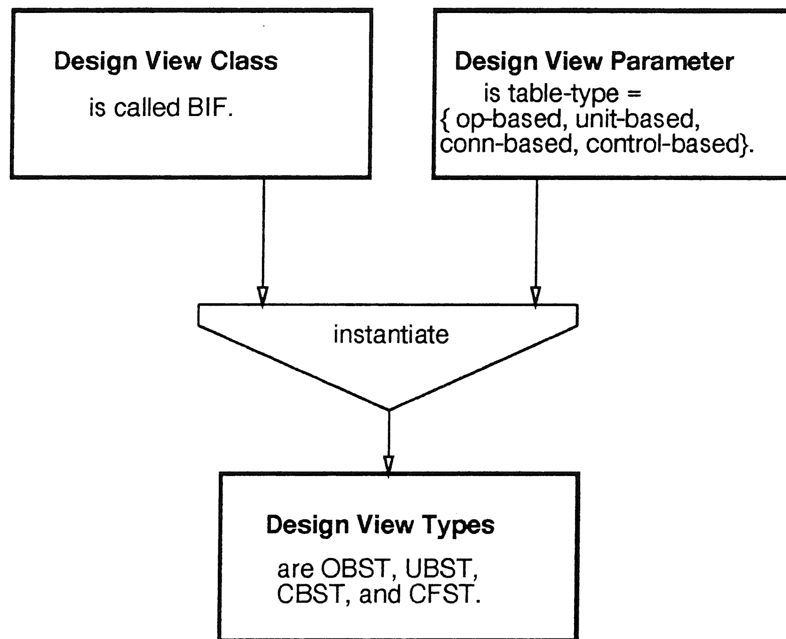

Figure 4: BIF Design Views

The relationship between the general design view scheme supported by BDDB and the BIF format is shown in Figure [?]. At present, the *BIF design view class* has one *design view parameter*, called `table-type`. The design view parameter `table-type` can take on the values `operation-based`, `unit-based`, `connection-based`, and `control-based`. It determines which aspects of the design are shown. For instance, setting the `table-type` parameter to the value `unit-based` means that the corresponding BIF view will contain information on the units that have been allocated to the actions in each state in order to perform the associated operations. For instance, the unit `ALU1` may be shown as having been allocated to perform the operation `PLUS` in state S1. Once a value is selected for the `table-type` parameter, then a fixed *BIF design view type* has been determined. We distinguish between four different design view types for BIF, which are called `operation-based state table`, `unit-based state table`, `connection-based state table`, and `control-based state table`. For a definition of the BIF format in general and these different BIF view types in particular see [1].

3 The BDDB DESIGN OBJECT MODEL

3.1 The BDDB Design Object Model

In this section we will give a short overview of the BDDB Design Object Model. For a more detailed presentation of the BDDB Design Object Model the reader is referred to [6] and [8].

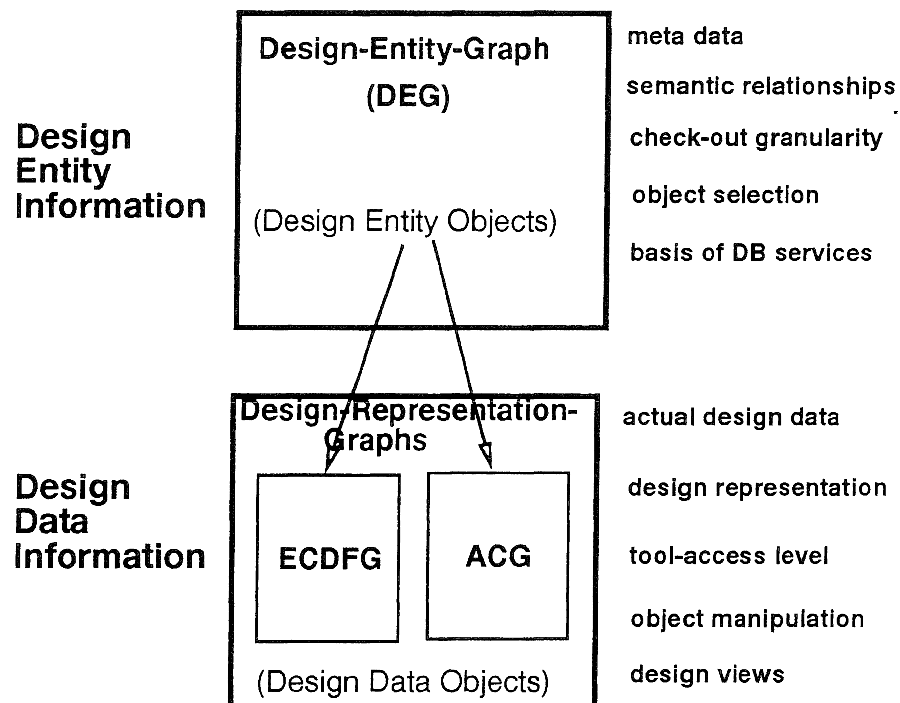

Figure 5: BDDB Design Object Model

The BDDB Design Object Model is a complete model of all information that is maintained by BDDB in order to aid the design process at the behavioral synthesis level in a CAD environment. As can be seen in Figure 5, BDDB divides the design information into two separate levels, the design entity information and the design data information.

The design entity information, represented by a Design Entity Graph Model, captures the overall organization of the design information. It covers concepts, such as, the design entity hierarchy, the version derivation tree, the levels of design abstractions, the different information domains, and configurations. Note that the design entity construct is a concept introduced by BDDB to decompose the potentially large set of design data objects that make up a design into manageable chunks. A design entity is an abstraction for a collection of interrelated design data objects, i.e., it groups together a collection of design data objects that form (a part of) a design. Design entities therefore are the granularity of design information. Design entities serve as locking granularity for data base access and as the units of data transfer between BDDB and the design tools. The Design Entity Graph (DEG) stores organizatorial attributes of a design entity, such as, its name, its version number, and the type of its content. Furthermore, the Design Entity Graph (DEG) keeps track of semantic relationships between different design entities, such as, equivalence, versioning, etc. The designer is allowed to query these relationships, however, s/he is not allowed to directly manipulate them. BDDB provides a procedural interface that supports a limited set of update operations on the design entity graph. An example operation may be to assert the equivalence between two design entities. The Design Entity Graph Model, sometimes also called the *meta-data model*, serves as foundation for most database support functions, for example, version management and schema browsing.

The second level of the BDDB Design Object Model maintains the design data information, i.e, the *actual design data* of the application domain. It describes the design at a level at which the design tools are ultimately interested in working on. BDDB represents this design data information using two graph models: the *behavioral graph model* and the *structural graph model*. The behavioral model describes the behavioral specification of the design. It corresponds to an Extended Control/Data Flow Graph (ECDFG) representation that is augmented with advanced features, such as, timing constraints, memory access, events, state transition information, and structure bindings. In short, the ECDFG model comprises the following information: (1) the VHDL input specification that describes the function of the design, (2) the flow-graph representation which captures the behavior over time, and (3) the state sequencing that shows the slicing of the behavior into states. The structural model, represented by an Annotated Component Graph, captures the hierarchical graph structure

of interconnected components augmented by timing constraints. It represents the hierarchical data path structure and its geometric implementation, called the floor-plan. A detailed discussion on these two design representation models can be found in [6].

Since these two design representation graphs capture the *actual design data* of the application domain, tool access to BDDB will primarily be at the design data level. In fact, design data stored in these two design representation graphs is generated as well as modified by design tools as well as by human designers during the design effort. BDDB itself is not concerned with creating this design data information.

BDDB provides a set of primitive access routines for these design object types, such as, the creation of such an object, the modification of an attribute value, etc. This set of primitive access routines can be used by tightly-coupled design tools to manipulate the design data objects. *Design view creation* also takes place at the design data level. Therefore, the BDEF format, which is a design view of the BDDB Design Object Model, is mainly concerned with capturing the design data found in the design representation graphs and not with representing the design entity information maintained in the design entity graph.

3.2 The Behavioral Object Type Definition Language

We need a formalism for defining all object types necessary to capture the BDDB Design Object Model. This description language needs to handle the direct representation and efficient manipulation of arbitrarily complex object types that can be deeply nested, possibly recursive, graph structures. Therefore, we have developed a Behavioral Type Definition Language[1].

The BNF description of this Behavioral Type Definition Language is given in Figures 6 and 7. In this language, a design object type is defined with the DEFINE TYPE statement by associating a name to a possibly nested type structure.

As can be seen, the *object type* definitions allow for the description of complex nested data structures typical for CAD applications. This is based on the constructors, such as, aggregation, which supports the composition of simple

[1] A language for the definition of new types is commonly referred to in the data base literature as Data Definition Language.

```
schema-definition ::= DEFINE SCHEMA  < schema-name>
   <simple-type>
   <abstract-type>
   END SCHEMA

<simple-type> ::= DEFINE SIMPLE TYPE < simple-type-name>
   <simple-type-definition>
END TYPE

<abstract-type> ::= DEFINE ABSTRACT TYPE < abstract-type-name>
     [SUPERTYPES: <type-name-list> ]
     DEFINED BY  <property-list>
END TYPE

<type-name-list> ::= '(' < type-name> { < type-name> }; ')'

<property-list> ::=
      TUPLE-OF (<property-specification>
{,<property-specification>})
      | <generic> < abstract-type-name> <characteristics>

<property-specification> ::=
     < property-name>: <domain-specification>

<domain-specification> ::=
       < simple-type-name> <characteristics-simple>
     | <generic> '('< simple-type-name> <characteristics-simple>')'
     | < abstract-type-name> <characteristics>
     | <generic> '('[REFERENCE]<
abstract-type-name><characteristics>')'

<characteristics-simple> ::= [UNIQUE] [REQUIRED]

<characteristics> ::=
   [INVERSE-OF < property-name>] [UNIQUE] [REQUIRED]

<generic> ::= SET-OF | LIST-OF
```

Figure 6: The Syntax of the Behavioral Object Type Definition Language

```
<simple-type-definition> ::= <integer-specification>
        | <real-specification>
        | <enumeration-specification>
        | STRING
        | BOOLEAN

<integer-specification> ::= INTEGER [<integer-range>]
<real-specification> ::= REAL [<real-range>]
<enumeration-specification> ::= <enumeration>
<integer-range> ::= (<integer>..<integer>)
<real-range> ::= (<real>..<real>)
```

Figure 7: The Syntax of the Behavioral Object Type Definition Language (cont.)

object types to define more *complex object types.* They allow for *shared subobjects* as well as for the description of *many-to-many* relationships that possibly can be symmetric. In fact, the model allows for *recursively defined object types* provided it leads to the definition of domains whose elements are finite. For instance, a data flow node can be described in terms of the control flow node to which it belongs while a control flow node can be described by listing its subconstructs, such as, its data flow nodes.

An object type is defined in terms of a set of attributes. An attribute specification defines the type of an attribute value. We support primitive types, user-defined abstract data types, and type constructors. The following collection of atomic types: Integer, String, Boolean, and Real, is assumed. User-defined abstract data types are constructed via type constructors as discussed below.

The Behavioral Type Definition Language supports three *generic* abstract data types: finite sets, lists and tuples. Generic (or parameterized) types are parameterized by one or more objects that can be of any type. They are a powerful tool for constructing new types; since they offer a homogeneous implementation for constructors. Below, we will describe these parameterized type constructors in more detail.

If T, T1 to Tn are abstract data types (ADTs), and A1 and A2 are names, then the generic type constructors can be used to defined complex abstract data types as follows:

- SET-OF(T) is an ADT,
- LIST-OF(T) is an ADT, and
- TUPLE-OF(A1:T1,A2:T2,..., An:Tn) is an ADT.

The parameterized type SET is commonly called a collection or an association abstraction in the data base field. It describes an unordered collection of objects of the same type T. A set can have an arbitrary number of members, i.e., it can be empty.

The parameterized type LIST is similar to the parameterized type SET. That is, like a set, it can have an arbitrary number of members. A list, however, implies an ordering on its members. Note that the list concept can be used to model a bag or multi-set; since it allows to store more than one occurrence of an element and one can simply ignore the ordering of the list.

The parameterized type TUPLE associates named objects of different types into one new type. It is commonly called an aggregation abstraction (or record) by the data base community. In the relational model, the term record implies that the field within the record are of primitive data types, while in a TUPLE type each of the fields can again be a complex attribute.

New types can then be constructed based on the predefined types and the parameter types by supplying specific parameter type to a generic type constructor. The SET constructor constructs the new object type SET-OF-TYPE1 when instantiated with the object type TYPE1. The LIST constructor constructs the new object type LIST-OF-TYPE1 when instantiated with the object type TYPE1. The TUPLE constructor constructs the new object type TUPLE(TYPE1, TYPE2,... TYPEn) when instantiated with the object types TYPE1 to TYPEn. These parameterized types have fixed sets of properties and operations. Object instances that created from such a parameterized type share the same protocol, i.e., the same set of messages, with different types of parameters.

Many different user-defined object types can be created from one parameterized type by instantiating the type parameter to a specific type. For instance, the parameterized type SET-OF(<TYPE>) may be instantiated to the specific object

types `SET-OF (EMP)` and `SET-OF (DEP)`. These instantiations of the parameterized type `SET` then share common operations. For instance, `SET-OF (EMP)` supports the operation `INSERT(se: SET-OF EMP, e: EMP);` and `SET-OF (DEP)` supports the operation `INSERT(sd: SET-OF DEP, d: DEP)`. The parameterized types thus allow the creation of new, strongly typed objects.

In the behavioral synthesis domain, we deal with several graph types. Examples are the state transition graph, the control flow graph, the data flow graph, and the structure graph. Since these graphs have different characteristics we provide only the basic type constructors discussed above out of which these graph objects can be constructed. Other features of our type definition language that are useful for constructing graph objects are reference of objects rather than containment of subobjects, bi-directionality of edges, and the symmetry of relationships.

In Figure 8, we show the example object type definitions needed to describe the schema for an example behavioral synthesis domain, namely, parts of the control/data flow graph. A more detailed description of the ECDFG object types can be found in Appendix B.

The definition has several interesting characteristics that should be noted. First, this schema consists of recursively defined object types. For instance, the object type definition for the CF-NODE is recursive, since one of its attributes Previous-Cf is defined to be of type CF-NODE. This is an example of how a graph structure can be modeled.

Secondly, the Port-Conf attribute corresponds to a port configuration, i.e., it models the input and output connections of a CF-NODE instance with other CF-NODE instances. Note that the Port-Conf attribute is a complex subobject, i.e., the port configuration is described by a separate type definition called PORT-CONF. Such a given PORT-CONF belongs to exactly one CF-NODE instance rather than being shared by more than one CF-NODE instance.

```
DEFINE SCHEMA  CFSCHEMA
  DEFINE SIMPLE TYPE portnames
   STRING(4)
  END TYPE;

  DEFINE ABSTRACT TYPE CF-GRAPH
  DEFINED BY
      SET-OF CF-NODE
  END TYPE

  DEFINE ABSTRACT TYPE CF-NODE
  SUPERTYPES: CF-CONSTRUCT
  DEFINED BY
      Name: String(20) REQUIRED,
      Previous-Cf: SET-OF  CF-NODE,
      Next-Cf: LIST-OF  CF-NODE,
      Port-Conf: PORT-CONF  UNIQUE
  END TYPE

  DEFINE ABSTRACT TYPE PORT-CONF
  SUPERTYPES:
  DEFINED BY
     InPortNum: integer,
     InPorts: SET-OF PORT
  END TYPE

  DEFINE ABSTRACT TYPE PORT
  SUPERTYPES:
  DEFINED BY
      portnum: integer,
      portname: portnames,
      portref:   CF-NODE
  END TYPE
END SCHEMA
```

Figure 8: An Example Using the Behavioral Object Type Definition Language

4 THE BDEF FORMAT

4.1 Overview over the BDEF Syntax

This section describes the textual flow graph view class, called BDEF (Behavioral Design Exchange Format). BDEF effectively corresponds to a set of rules for describing instances of design data objects defined by the Behavioral Type Description Language. Hence is also called Behavioral Instance Description Language. For the complete BNF grammar of the the Behavioral Instance Description Language applied to the ECDFG design representation model [6] the reader is referred to Appendix A. In this section, we will introduce the BDEF constructs by means of examples. In Figure 9, we show the general BNF syntax of BDEF. This syntax corresponds to general rules of textual instance description, for instance, it specifies how a list attribute is represented. It does however not give the actual attribute names and attribute values of the application domain. Consequently, this BDEF format could be applied to other design representation models.

In the rest of this section, though, we will describe how the BDEF syntax has been applied to our example application, i.e., to the BDDB Design Object Model. In Figure 9, we give a description the general BDEF format rules using BNF formalism. In Figure 10, these BDEF format rules are shown in a graphical manner. This graphical representation uses the following constructs:

- An arrow indicates that the object at the source of the arrow (higher in the tree) is represented in terms of the objects at the destination of the arrow (lower in the tree).
- A dot at the source of an arrow indicates that the object at the source of the arrow corresponds to a *set* of the objects below.
- A half cicle around all arrows leaving an object indicates that the object is composed of *one and only one* of the objects below. If there is no half circle then the object above must be composed of *all* objects below.
- Objects in bold print correspond to internal nodes. Objects in light print correspond to leaf nodes of the syntax tree. Leaf nodes are not further defined in the syntax tree, i.e., they correspond to values from a simple domain type, like, Integer or String. Non-leaf objects are further defined in the syntax tree, namely, based on the decomposition described by a set of one or more arrows leaving the node.

```
design-file ::= design-entities
design-entities ::= design-entity { design-entity }
design-entity ::= '('design-entity-header design-entity-body')'
design-entity-header ::= '[' header-attributes ']'
design-entity-body ::= design-object { design-object }
design-object ::= '('design-object-header design-object-body')'
design-object-header ::= design-object-type#object-id
design-object-body ::= design-object-attributes
design-object-attributes ::= design-object-attribute
                             {',' design-object-attribute}
attributes ::= attribute { attribute }
attribute ::=
    attribute-name ':' attribute-value
    | attribute-name ':' '<'list-of-attribute-values '>'
    | attribute-name ':' '{'set-of-attribute-values '}'

list-of-attribute-values ::= attribute-value {, attribute-value }
set-of-attribute-values ::= attribute-value {, attribute-value }

attribute-value ::=
    simple-value
    | list-of-attribute-values
    | set-of-attribute-values
    | independent-design-object
    | dependent-design-object
    | design-object-reference

design-object-reference ::= '#'design-entity-reference
     '#'design-object-type'#'object-id'#'


independent-design-object  ::= design-object
dependent-design-object ::= '[' design-object-body ']'
```

Figure 9: BNF Syntax of BDEF

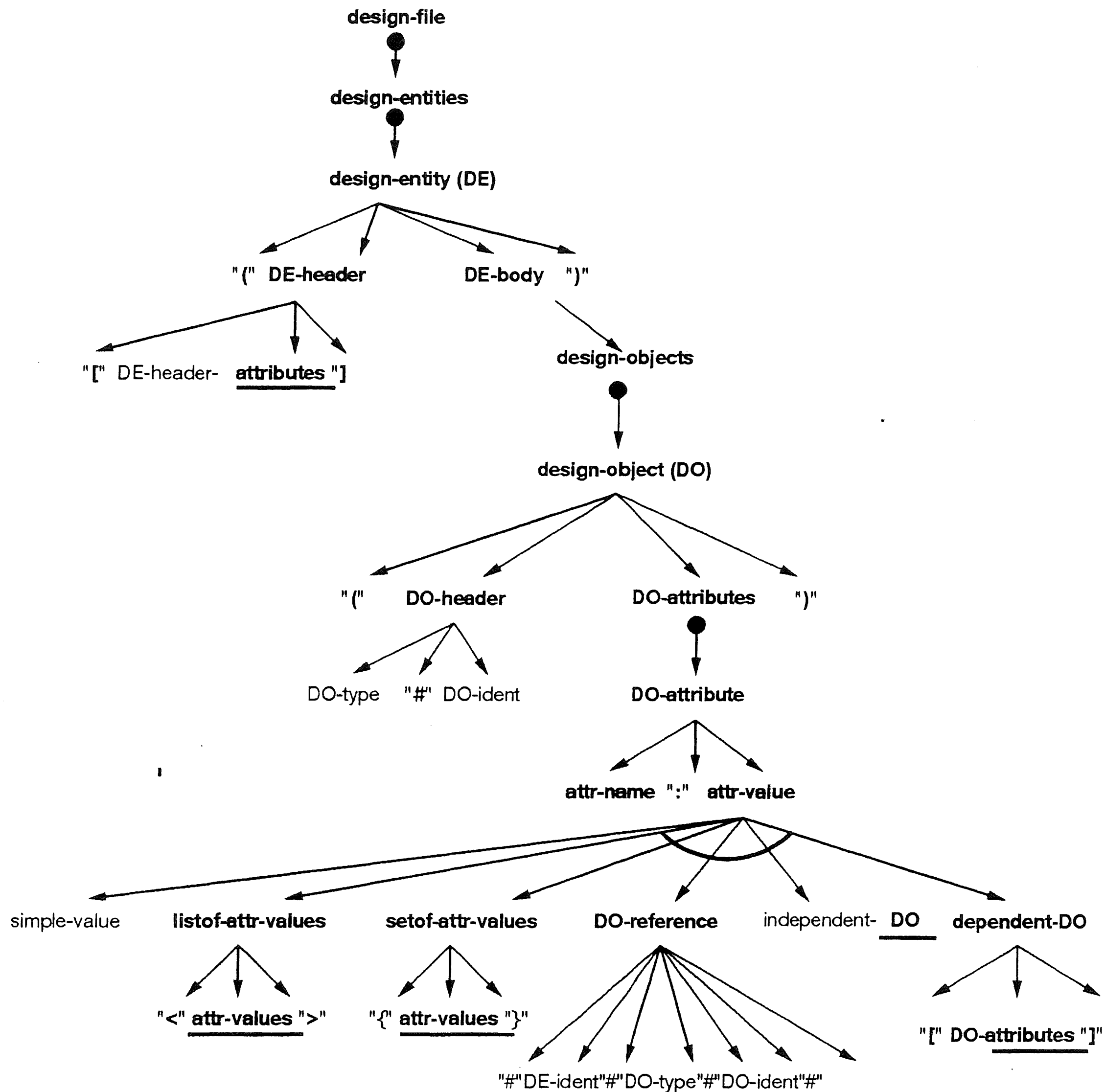

Figure 10: Graphical Depiction of BDEF Syntax Tree

- A bold underlined object indicates that the object is further defined at another location in the syntax tree.

4.2 Using BDEF to Represent Design Entity Information

As explained in Section 3.1, *design entities* are the units of data transfer between BDDB, i.e., secondary storage, and the design tools, i.e., main memory [6]. Hence, BDEF is not used to represent the complete Design-Entity Graph Model. Instead, BDEF is used to represent information associated with one design entity only. This includes the design entity attributes (stored in the DEG) and the design data objects contained in the design entity (stored in one of the Design Representation Graphs). The BDEF representation of the design entity attributes is discussed in this section and the BDEF representation of the design data objects in the next section.

A *design file* which corresponds to an ASCII file that is exchanged between BDDB and design tools holds one or more design entities. As indicated in the syntax in Figure 9 the order of design entities in a design file is insignificant. In a design file, each *design entity* is encapsulated by a pair of parentheses.

The BDEF representation of a design entity consists of a *design entity header* and a *design entity body*. The design entity header gives the general characteristics of the design entity, while the design entity body describes the composition of the design in terms of actual design data objects.

In a BDEF description, the *design entity header* is listed before the *design entity body*. The *design entity header*, encapsulated by a pair of square parentheses, lists a set of *header*-attributes. Example *header*-attributes are the following:

- the design entity name,
- the design entity version number,
- the design entity domain type,
- the design entity flavor, and
- the design entity chunk type.

The first two attributes are used to uniquely identify the design entity. The last three attributes in the design entity header characterize the design data objects out of which the design entity is composed of. For instance, the `domain-type` attribute indicates whether the design data objects belong to the behavioral or to the structural domain. If the `domain-type` attribute is behavioral, then the `behavioral-flavor` attribute gives more details on the information content of the design. For instance, it would indicate whether it is a purely behavioral design or whether state information has already been synthesized.

The domains of these attributes are:

- **domain type:** { `behavior, structure` }.
- **behavioral flavor:** { `behav-pure, behav-states, behav-allocation,`

  `behav-binding, behav-control` }.
- **behavioral design chunk:** { `data-flow, control-flow, control-data-flow, state-graph, state-control-flow, state-control-data-flow`}.
- **structural flavor:** { `comps, comps-connections, comps-geometry-estimates, comps-geometry`}.

These *header*-attributes are represented in BDEF like all other attributes. Therefore see the discussion on the attribute representation below for further details. A BDEF description of one example design entity is given next:

This design entity has three attributes that make up its header information. The actual representation of the design entity body is discussed next.

```
/* one design entity */
(
  [
    /* design entity attributes */
     DD_NAME: CONTROL_COUNTER,
     DD_VERSION: 20,
     DD_DOMAIN: BEHAVIOR,
     DD_BEHAV_FLAVOR: BEHAVIOR_STATES,
     DD_CHUNK_TYPE: STATE_CONTROL_DATA_FLOW
  ]

  /* design entity body goes here. */
  /* It is a collection of design objects. */
)
```

This design entity has five attributes that make up its header information. BDEF representation of the design entity body is discussed in the next section.

4.3 Using BDEF to Represent Design Data Information

The design entity construct is a concept introduced by BDDB to decompose the potentially large set of design data objects into manageable chunks. In other words, a design entity is an abstraction of a collection of actual design data objects which are represented in its design entity body. These design objects can be as high-level as a complex process description or as low-level as a simple logic gate. In this section, we are concerned with the BDEF representation of these design data objects.

These design data objects are instances of object types that capture our design representation model [6]. Recall that our design representation model distinguishes between the *behavioral graph model* and the *structural graph model* which correspond to the Extended Control/Data Flow Graph (ECDFG) Model and Annotated Component Graph Model, respectively. In this paper, we will use the former, the Extended Control/Data Flow Graph (ECDFG) model as example application since it is the more interesting and diverse model. The ECDFG model is described by a collection of design object types defined by the Behavioral Object Type Definition Language (see Appendix B). A complete implementation of these design data types in the C programming language has

also been developed [4]. Note however the former abstracts concepts of the application domain, like for instance, ordered lists of ports, sets of operations, whereas the later stays at a lower level of implementation details. In Appendix A, we give the complete BDEF syntax for capturing all design object types of these Design Data Representation Graphs. Whereas below we will describe some general rules on how to represent these design data objects using the BDEF view format.

In BDEF, each design object is encapsulated by a set of parenthesis. The BDEF description of a design object consists of three parts: its object type, its identifier, and its attributes. The type of the design object is given at the beginning of its BDEF description. In addition, the object identifier of the design object which uniquely identifies the object within a given design entity is appended with a '#' symbol. For instance, a data flow node with the identifier 10 would be represented in BDEF as follows:

```
(DF_NODE #10

   /*  data flow node attributes here */
)
```

A design object is described by listing one or more of its attributes. Attributes are identified by giving attribute names rather than using their position. This introduces redundant data as attribute names are repeated within the definition of each object. It is needed, however, since the number and type of attributes of a design object may vary with the requirements of design tools, i.e., the design view type. All attribute specifications correspond to an attribute name and value pair. The attribute name and value pair is separated by the ":" symbol. The attributes of a design object are listed by separating them by commas, with the ordering of the attributes being arbitrary due to the use of attribute names. Therefore, the data flow node example can now be extended as follows:

```
(DF_NODE #10

   attribute-name1: attribute-value1,
   attribute-name2: attribute-value2,
   ...
)
```

As shown in Figure 10, an attribute value can be one of the following:

1. a simple value,
2. a list of attribute values,
3. a set of attribute values,
4. an independent subobject,
5. a dependent subobject, or
6. an object reference.

The format of simple values is determined by the domain of the attribute. The domain of a simple value is a primitive data type, such as, integer, float, boolean, or string. Simple attribute values often are also restricted to a predefined domain (an enumeration type) that corresponds to a subset of one of these primitive data types.

Furthermore, If the object type of an attribute value corresponds to a list or to a set of attribute values, then the attribute values themselves can be any of the above listed options. If the object type of an attribute is of type SET, then the instance of that type has an attribute value of the form '{' e1, e2, ..., en '}'. In other words, set-valued attributes are distinguished from single values by enclosing the attribute value list into a pair of set brackets '{' and '}'. Furtheremore, two adajent data values in a set attribute are separated by a comma. If the object type of an attribute is of type LIST, then the instance of that type has an attribute value of the form '<' e1, e2, ..., en '>'. We have extended the previous example to show a simple attribute, a list attribute and a set attribute:

```
(DF_NODE #10

  /* simple attribute */
  node-type: STRING,

  /* list attribute */
  input-ports: < I1, I2, I3 >,

  /* set attribute */
  operations: { ADD, SUBTRACT },

)
```

The list attribute `input-ports` consists of the three values I1, I2, and I3. Being a list attribute, the order of these values is significant since it is a For instance, I1 might correspond to the left data input, I2 to the right data input, and I3 to the carry input to the data flow node. The order of the attribute values for the set attribute, on the other hand, is not significant. The attribute `operations` simply indicates that the data flow node may either execute an addition operation or an subtraction operation. It says nothing however about the fact when or in which order either of these two is used.

Recall that an attribute value can correspond to a dependent or an independent subobject. This attribute type is introduced to support the modeling of a hierarchy of design objects, i.e., the fact that a design object is composed of other lower-level design objects. If a design object is composed of lower-level design objects, then these lower-level design objects (subobjects) become attribute of the super-object. Subobject attributes are treated as follows. The format of independent design objects has been described earlier in this report, namely, they are encapsulated by a pair of round brackets '(' and ')'. Recall that independent design objects correspond to full-fledged design objects with their own unique identifier. Dependent design objects on the other hand correspond to design objects that exist only in the context of their 'super-' design object. Therefore, dependent design objects – similar to simple values – do not carry any identifiers. In BDEF, dependent design objects are encapsulated by a pair of angular brackets '[' and ']', while independent design objects, nested or not nested, are encapsulated by a pair of round brackets '(' and ')'. In other words, if an object type has an attribute of type TUPLE, then the instance of that type has an attribute value of the form [a1:o1, a2:o2, ..., an:on]. If, on the other hand, an object type T1 has an attribute of type T2, then the instance of that type has an attribute value of the form (t2) with t2 an instance of the object type T2. In short, if a design object is composed of lower-level design objects, then a nesting of parenthesis is created in BDEF to model this design object containment relationship. Again we have extended our data flow node example to show an example of both a dependent and an independent subobject:

```
(DF_NODE #10

  /* list of independent subobjects attribute */
  ports: <  (DF_PORT #1 ... ), (DF_PORT #2 ... )  >,

  /* dependent subobject attribute */
  condition: [  cond: STRING, value: STRING, op: OPERATION ]

)
```

1. a simple value,
2. a list of attribute values,
3. a set of attribute values,
4. an independent subobject,
5. a dependent subobject, or
6. an object reference.

The format of simple values is determined by the domain of the attribute. The domain of a simple value is a primitive data type, such as, integer, float, boolean, or string. Simple attribute values often are also restricted to a predefined domain (an enumeration type) that corresponds to a subset of one of these primitive data types.

Furthermore, If the object type of an attribute value corresponds to a list or to a set of attribute values, then the attribute values themselves can be any of the above listed options. If the object type of an attribute is of type SET, then the instance of that type has an attribute value of the form ‘{’ e1, e2, ..., en ‘}’. In other words, set-valued attributes are distinguished from single values by enclosing the attribute value list into a pair of set brackets ‘{’ and ‘}’. Furtheremore, two adajent data values in a set attribute are separated by a comma. If the object type of an attribute is of type LIST, then the instance of that type has an attribute value of the form ‘<’ e1, e2, ..., en ‘>’. We have extended the previous example to show a simple attribute, a list attribute and a set attribute:

```
(DF_NODE #10

  /* simple attribute */
  node-type: STRING,

  /* list attribute */
  input-ports: < I1, I2, I3 >,

  /* set attribute */
  operations: { ADD, SUBTRACT },

)
```

The list attribute `input-ports` consists of the three values I1, I2, and I3. Being a list attribute, the order of these values is significant since it is a For instance, I1 might correspond to the left data input, I2 to the right data input, and I3 to the carry input to the data flow node. The order of the attribute values for the set attribute, on the other hand, is not significant. The attribute `operations` simply indicates that the data flow node may either execute an addition operation or an subtraction operation. It says nothing however about the fact when or in which order either of these two is used.

Recall that an attribute value can correspond to a dependent or an independent subobject. This attribute type is introduced to support the modeling of a hierarchy of design objects, i.e., the fact that a design object is composed of other lower-level design objects. If a design object is composed of lower-level design objects, then these lower-level design objects (subobjects) become attribute of the super-object. Subobject attributes are treated as follows. The format of independent design objects has been described earlier in this report, namely, they are encapsulated by a pair of round brackets '(' and ')'. Recall that independent design objects correspond to full-fledged design objects with their own unique identifier. Dependent design objects on the other hand correspond to design objects that exist only in the context of their 'super-' design object. Therefore, dependent design objects – similar to simple values – do not carry any identifiers. In BDEF, dependent design objects are encapsulated by a pair of angular brackets '[' and ']', while independent design objects, nested or not nested, are encapsulated by a pair of round brackets '(' and ')'. In other words, if an object type has an attribute of type TUPLE, then the instance of that type has an attribute value of the form [a1:o1, a2:o2, ..., an:on]. If, on the other hand, an object type T1 has an attribute of type T2, then the instance of that type has an attribute value of the form (t2) with t2 an instance of the object type T2. In short, if a design object is composed of lower-level design objects, then a nesting of parenthesis is created in BDEF to model this design object containment relationship. Again we have extended our data flow node example to show an example of both a dependent and an independent subobject:

```
(DF_NODE #10

  /* list of independent subobjects attribute */
  ports: <  (DF_PORT #1 ... ), (DF_PORT #2 ... )  >,

  /* dependent subobject attribute */
  condition: [  cond: STRING, value: STRING, op: OPERATION ]

)
```

The `ports` attribute corresponds to a list of independent design subobjects. Each port object has its own unique identifier and could have been represented as separate object in the BDEF description. For this example, however, we have chosen to represent data flow port objects as nested subobjects of the data flow node objects, since ports are conceptually closely related to one and only one data flow node. The `condition` attribute corresponds to a dependent design subobject. This attribute does not have its own unique identifier, i.e., the condition information would be meaningless without the corresponding data flow node object. The `condition` attribute indicates the condition under which each of the operations associated with the data flow node is executed.

We introduce the concept of BDEF references in order to model relationships between design objects. In BDEF, references themselves don't carry any attributes. The format of a reference consists of three parts, each separated by a '#' symbol. The first part names the design entity in which the referenced design object is being defined. The second part gives the object type of the design object that is being referenced. Finally, the third part gives the identifier of the design object that is being referenced. The referenced design object can either be in the same design entity as the referencing design object or in a different design entity. The default of the same design entity is assumed. If the first part of the reference is null, i.e., no design entity name is specified, then the reference is assumed to be referring a design object within the current design entity. We have extended the data flow node example to reflect the fact that the data flow node is referencing the control flow node to which it belongs. Since the data flow node and the control flow node are both in the same design entity, the reference attribute only gives the type of the referenced object, CF_NODE, and its identifier, in this case, 5.

```
(DF_NODE #10

  /* reference attribute */
  associated-cf-node: ##CF_NODE#5#,

)
```

A design object can only be defined once within a design entity description, but it can be referenced multiple times. This means that an object identifier can be listed only once in an object header, but can be specified numerous times in form of an object reference.

5 BDEF DESIGN VIEWS

In section 2, we have introduced our general approach for handling design views in BDDB. In this section, we will concentrate on one example of such a design view class, namely, the textual flow graph format BDEF.

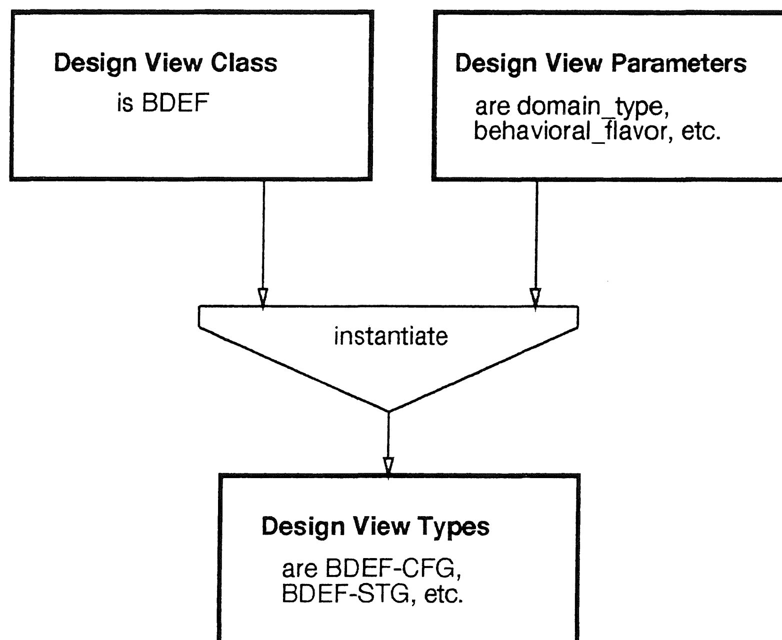

Figure 11: BDEF Design Views

The relationship between the general BDDB design view scheme and the BDEF design description is shown in Figure 11. Using the terminology introduced in Section 2, the textual flow graph language BDEF is called a *design*

view class. Furthermore, the design entity characteristics presented in Section 4.2 correspond to the *view parameters* of the BDEF design view class. Recall that the selection of values for all *view parameters* of a *design view class* will determine a particular *design view type* (Figure 11). Consequently, a particular combination of design entity characteristics determines the desired *BDEF design view type.*

A discussion of the *BDEF design view parameters* follows. The BDEF description of a design entity can either be composed of design objects from the behavioral domain or from the structural domain, i.e., the Extended Control/Data Flow Graph (ECDFG) or the Annotated Component Graph (ACG). This is determined by the **domain-type** view parameter. Another distinction between different view types of a design entity is based on the amount of information associated with the design objects that compose the design entity. For instance, if the **behavioral-flavor** parameter is set to **behav-states** then the design objects that compose the corresponding design entity will describe the design behavior augmented by state assignment information. If the **behavioral-flavor** parameter is equal to **pure-behav** then the design objects will only capture the specified behavior of the design.

The view generation process is driven by integrity rules that distinguish between the following three attribute modes:

- required,
- optional,
- and not-allowed.

These integrity rules are important since they ensure the correctness of information in a BDEF design view description. A novel feature of our integrity rules is that these attribute modes are determined **dynamically** rather than **statically**. Meaning the mode of an attribute is not fixed during object type definition but varies with the given design view type. An example of an attribute that is **required** for all design view types is the object identifier. An example of an attribute that is only sometimes **required** is the state information. If the **behavioral-flavor** view parameter is equal to **behav-states**, then the associated state attributes become **required** for both the control-flow and the data-flow nodes. Otherwise they are optional. If, however, the **behavioral-flavor** parameter is equal to **pure-behav**, then the associated state attributes are **not-allowed**. This is one important means for BDDB to

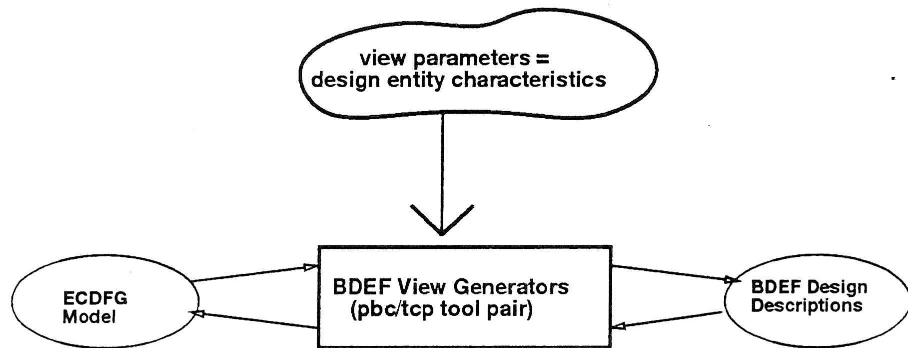

Figure 12: BDEF Design View Generators

verify the correctness of the information in a BDEF view and to guide the view generation process.

We have developed BDEF Design View Generator software. These tools extract BDEF design views from the BDDB Design Object Model using the view parameters described earlier. Vice versa, these tools also extract BDEF design views from a given BDEF design description and map this subview of the design information into the corresponding BDDB Design Object Model, i.e., the ECDFG graph structures.

The BDEF View Generator package has three view parameters, namely, design entity domain type, design entity flavor, and design entity chunk type. These parameters are set by the user of these generators, and allow him/her to request a certain type of design data and organization of this data. The view generators then check whether the requested information is available in the current design description. If it is then the desired view is generated. If the requested information is not available then the user of the BDEF view generator will be notified.

This BDEF view generator is a valuable tool for design data exchange in a behavioral synthesis environment. First, it allows to capture design data manipulated on by design tools in one common format. And secondly, it supports the mapping of this design information from the BDEF format to the shared design representation data structures and back. A manual that describes how to use this software is appended to this report (Appendix D).

6 EXAMPLES

In this section, we show several examples of how the designs in the BDDB Design Object Model (in particular, the ECDFG graph structures) are described by the BDEF exchange format.

6.1 A Simple Data Flow Graph Example

In this section we present the BDEF description of a simple data flow graph example. The behavioral specification of this design corresponds to the VHDL description "C = A + B". The graphical representation of this specification is given in Figure 13. On the left-hand side of the figure we show a high-level graphical depiction of the design representation, while on the right-hand side we show a more detailed view of the data flow graph. The more detailed presentation shows the net objects, which represent the data values that flow between data flow node objects, and the port objects. Ports are (independent) subobjects of data flow nodes and nets that model the interconnection points of these nodes. In addition, all data flow nodes, data flow nets, and data flow ports have their own object identifiers which is depicted by integer numbers to the left of each object.

The BDEF description of this design example is shown in Figures 14 and 15. As can be seen in the BDEF description, all data flow nodes, data flow nets, and data flow ports are described as independent design objects with their own object identifiers. Note that all design objects are part of the same design entity. Therefore, all references used in this BDEF description are of the form ##object-type#id#, and the default, namely, the current design entity, is assumed.

6.2 A Complete Control/Data Flow Graph Example

In this section, we present a complete example of how to use BDEF to model both control flow as well as data flow. Figure 16 shows a design specification written in VHDL. This design specification is compiled into a ECDFG design representation with a VHDL Graph Compiler [3]. The resulting graphical representation which corresponds to a control/data flow graph, is given in Figure 17. This depiction of the design representation is, of course, a high-level view

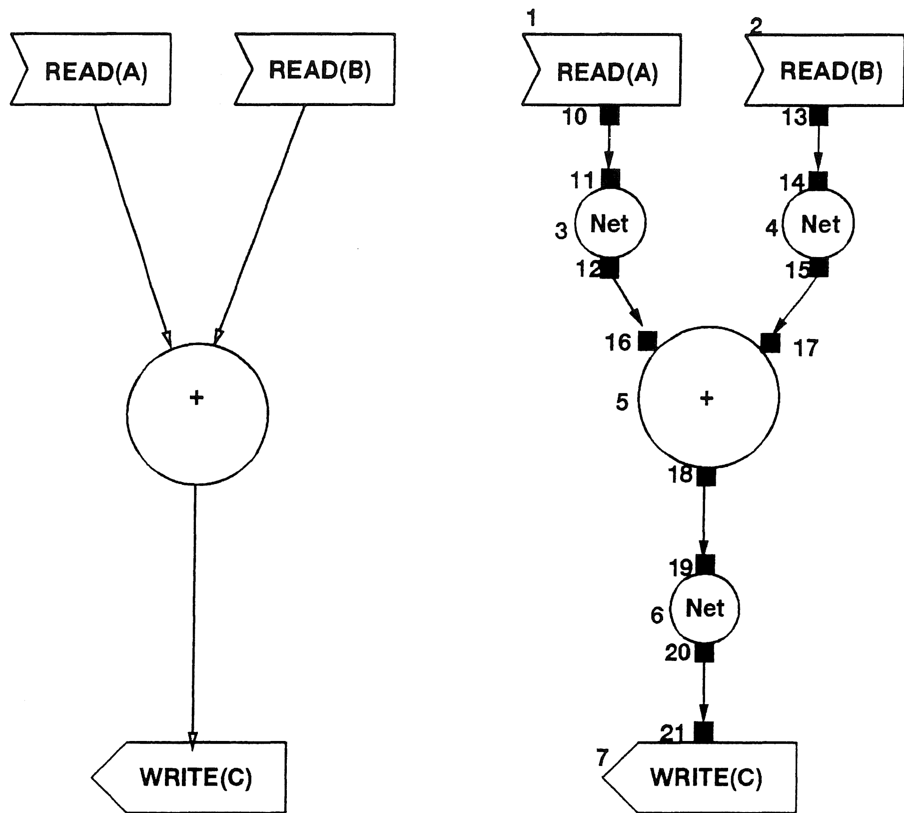

Figure 13: Graphical Representation of a Data Flow Graph

```
(
[ DE_NAME: EXAMPLE_DFG
  DD_VERSION: DEFAULT
  DD_DOMAIN_TYPE: BEHAVIOR,
  DD_FLAVOR: BEHAVIOR_PURE,
  DD_CHUNK_TYPE: DATA_FLOW
]

(DF_NODE#1
NODE_CLASS: DF_OP,
NUM_INPUTS: 0,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#1
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#2# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "A",
  SIG_TYPE: PORT,
  NODE_TYPE: DATA_ACCESS,
  DF_OP_INFO: <
    [OP_CLASS: MISC_OP,
    OP_TYPE: READ,
    OP_NAME: "A"
    ]>
  ]
)

(DF_NET#3
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#2
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#1# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#6
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#5# >
  )>,
DF_NET_INFO: [
  NET_TYPE: DATA_NET,
  NUM_REPRES: BINARY
  ]
)

(DF_NODE#2
NODE_CLASS: DF_OP,
NUM_INPUTS: 0,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#3
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#4# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "B",
  SIG_TYPE: PORT,
  NODE_TYPE: DATA_ACCESS,
  DF_OP_INFO: <
    [OP_CLASS: MISC_OP,
    OP_TYPE: READ,
    OP_NAME: "B"
    ]>
  ]
)

(DF_NET#4
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#4
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#3# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#8
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#7# >
  )>,
```

Figure 14: BDEF Description of the Data Flow Graph

```
  DF_NET_INFO: [
    NET_TYPE: DATA_NET,
    NUM_REPRES: BINARY
    ]
  )

  (DF_NODE#5
  NODE_CLASS: DF_OP,
  NUM_INPUTS: 2,
  INPUT1: <
    (DF_PORT#5
    IO_CLASS: INPUT,
    PORT_TYPE: DATA_PORT,
    NUM_CONN: 1,
    CONN1: < ##DF_PORT#6# >
    ),
    (DF_PORT#7
    IO_CLASS: INPUT,
    PORT_TYPE: DATA_PORT,
    NUM_CONN: 1,
    CONN1: < ##DF_PORT#8# >
    )>,
  NUM_OUTPUTS: 1,
  OUTPUT1: <
    (DF_PORT#9
    IO_CLASS: OUTPUT,
    PORT_TYPE: DATA_PORT,
    NUM_CONN: 1,
    CONN1: < ##DF_PORT#10# >
    )>,
  DF_NODE_INFO: [
    SIG_NAME: "+",
    NODE_TYPE: OPERATION,
    DF_OP_INFO: <
      [OP_CLASS: ADDING,
      OP_TYPE: ADD,
      OP_NAME: "+"
      ]>
    ]
)

(DF_NET#6
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#10
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#9# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#12
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#11# >
  )>,
DF_NET_INFO: [
  NET_TYPE: DATA_NET,
  NUM_REPRES: BINARY
  ]
)

(DF_NODE#7
NODE_CLASS: DF_OP,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#11
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#12# >
  )>,
NUM_OUTPUTS: 0,
DF_NODE_INFO: [
  SIG_NAME: "C",
  SIG_TYPE: PORT,
  NODE_TYPE: DATA_ACCESS,
  DF_OP_INFO: <
    [OP_CLASS: MISC_OP,
    OP_TYPE: WRITE,
    OP_NAME: "C"
    ]>
  ]
)
)
```

Figure 15: BDEF Description of a Simple Data Flow Graph (cont.)

that does not detail many of the interrelationships and attributes of these design objects.

```
entity AM2910 is
   port (
         FULL_sig: out BIT                           -- stack full
flag
         );
end AM2910;

architecture BEHAVIOR of AM2910 is
begin
process
   variable SP : BIT_VECTOR(2 downto 0);       -- stack pointer
begin

     if (SP = B''100'') then
        FULL_sig <= 0;
     else
        FULL_sig <= 1;
        SP := SP + 1;
     end if;

end process;
end BEHAVIOR;
```

Figure 16: VHDL Design Specification

The BDEF Generator and the ECDFG Generator tools (see Appendix D) now allow us to extract different BDEF design views from this design data. Due to space limitations we list some of these, i.e., the data flow view, the control flow view, and the control/data flow view in Appendix C, rather than presented them here.

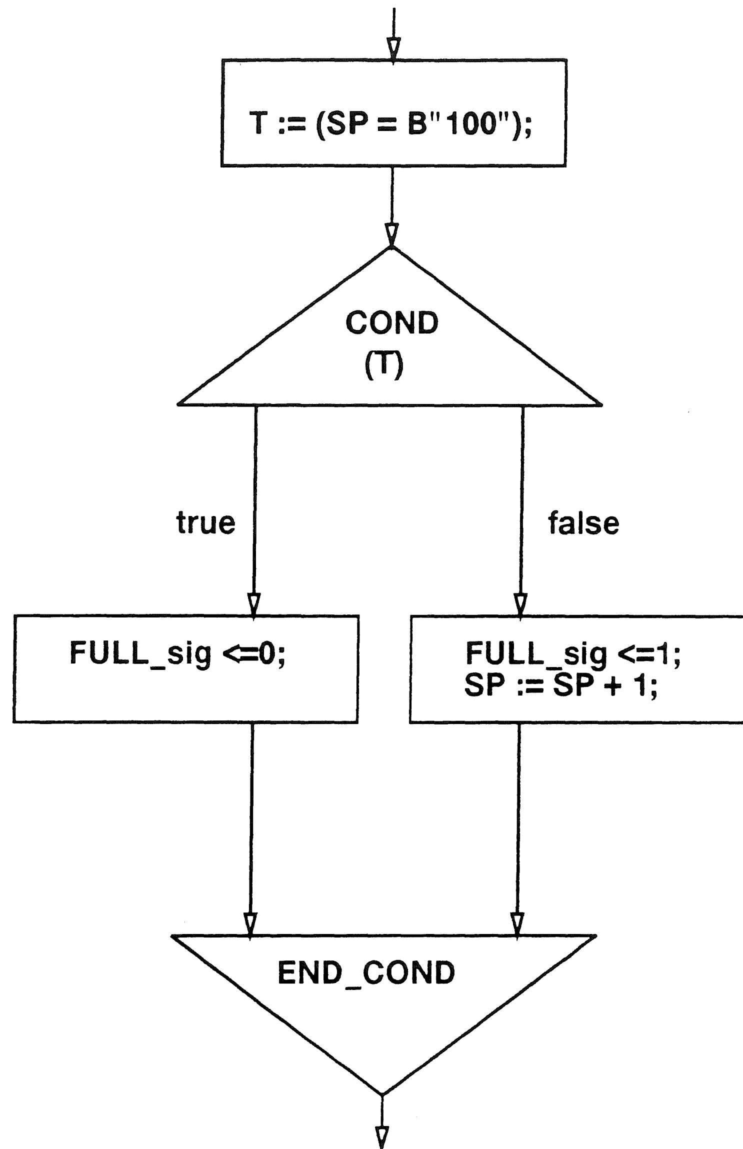

Figure 17: A Graphical ECDFG Depiction of the VHDL Design Specification

6.3 An Example using Two Design Entities

In this section, we will discuss an example design that is decomposed into two design entities.

```
architecture DE1 of Design1 is
  // body of DE1
proc ( A1, A2 );
...
end DE1;

architecture DE2 of Design2 is
 procedure proc  ( P1, P2 )  is
 begin
  // procedure body
 end proc;
end DE2;
```

Figure 18: VHDL Description of a Procedure Definition/Call

This example contains a procedure definition as well as a procedure call. In Figure 18, we show the VHDL description of the procedure definition of procedure **proc** in design entity D2 and the usage of this procedure in form of a procedure call in design entity D1. The procedure **proc** has two formal parameters P1 and P2. The procedure call uses the two actual parameters A1 and A2 which are bound to the formals P1 and P2, respectively. This VHDL description is compiled into two design entities in BDDB as can be seen in Figure 19. Design entity D1 contains the design objects for the control/data flow graph representing the procedure call, while design entity D2 holds the design data objects for the procedure definition [6].

The BDEF description of this design is shown in Figure 20. There are two different design entity objects in this design file, namely, design entity with DD_Name=D1 and design entity with DD_Name=D2. Design entity D1 contains several design objects of type CF_NODE. The first object CF_NODE#201 corresponds to the procedure call node. The second design object CF_NODE#202

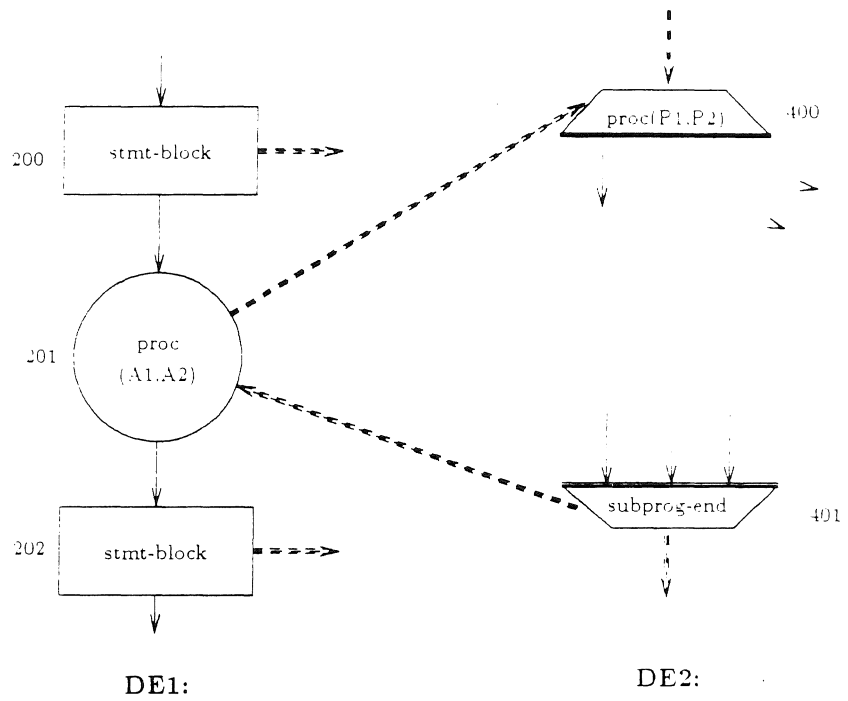

Figure 19: Graphical Representation of a Procedure Definition/Call

```
(
[ DD_NAME: DE1,
  DD_CHUNK_TYPE: CONTROL_FLOW
]
(CF_NODE#201
CF_NODE_TYPE: PROC_CALL,
PARAS: <A1, A2>,
PROCEDURE: #DE2#PROCEDURE#400#,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (CF_CONNS#1
  IO_CLASS: OUTPUT,
  CF_CONNS_REF: ##CF_CONNS#2#
  )>,
...)
(CF_NODE#202
CF_NODE_TYPE: STMT_BLK,
NUM_INPUTS: 1,
INPUT1: <
  (CF_CONNS#2
  IO_CLASS: INPUT,
  CF_CONNS_REF: ##CF_CONNS#1#
  )>,
...)
)

(
[ DD_NAME: DE2,
  DD_CHUNK_TYPE: CONTROL_FLOW
]
(CF_NODE#400
CF_NODE_TYPE: PROC_DECL,
PARAS: <P1, P2>,
...)
)
```

Figure 20: A BDEF Description of a Procedure Definition/Call Graph

models the statement block node that is executed after returning from the execution of the procedure call. In other words, it is connected via sequencing arcs to the procedure call node in the control/data flow graph in DE1. This connection between the objects CF_NODE#201 and CF_NODE#202 is described in terms of CF_CONNS objects in the BDEF description. The design object CF_CONNS#1, which is a nested subobject of CF_NODE#201, has an object reference CF_CONNS_REF to the object CF_CONNS#2. CF_CONNS#2 on the other hand is a nested subobject of CF_NODE#202 and it has a reference back to the object CF_CONNS#1. Hence, it is a symmetric relationship.

In this example there are relationships between design data objects that reside in different design entities. Namely, the procedure call and its corresponding procedure declaration are in the design entities DE1 and DE2, respectively. In Figure 19, this reference across design entity boundaries is represented by bold dashed arrows. In a BDEF description, such an inter-design-entity reference is specified by an extended object reference which indicates the design entity in which the referenced object resides. Therefore, in this example BDEF description there is an object reference of the form "#DE2#PROCEDURE#400#" that relates the procedure call node in design entity D1 with the procedure declaration node in design entity D2. The reference specifies the design entity identifier "DE2" whereas all references that refer to objects within the same design entity can simply omit this identifier.

6.4 A Timing Constraint Example At the Control Flow Level

This section describes the representation of timing constraints in the ECDFG model and the specification of these constraints in the BDEF format.

The discussion of timing constraints at the control flow level is based on the ECDFG example depicted in Figure 17. This design has been augmented by a timing constraint as shown in Figure 21. This timing constraint T1 specifies a minimum delay of 10ns and a maximum delay of 250ns for the execution of the false branch of the if-statement. A timing constraint at the control flow level originates and ends at a control flow graph connection, a sequencing arc. In Figure 21, this is displayed by the bold timing arcs labeled timing_start and timing_finish that connect control flow sequencing arcs with the timing specification node. The timing specification then is a constraint on the execution of all behavior specified between these two points in the control flow graph. If the timing constraint is supposed to constrain some but not all of the threads of computation between these two points, then a path expression needs to be

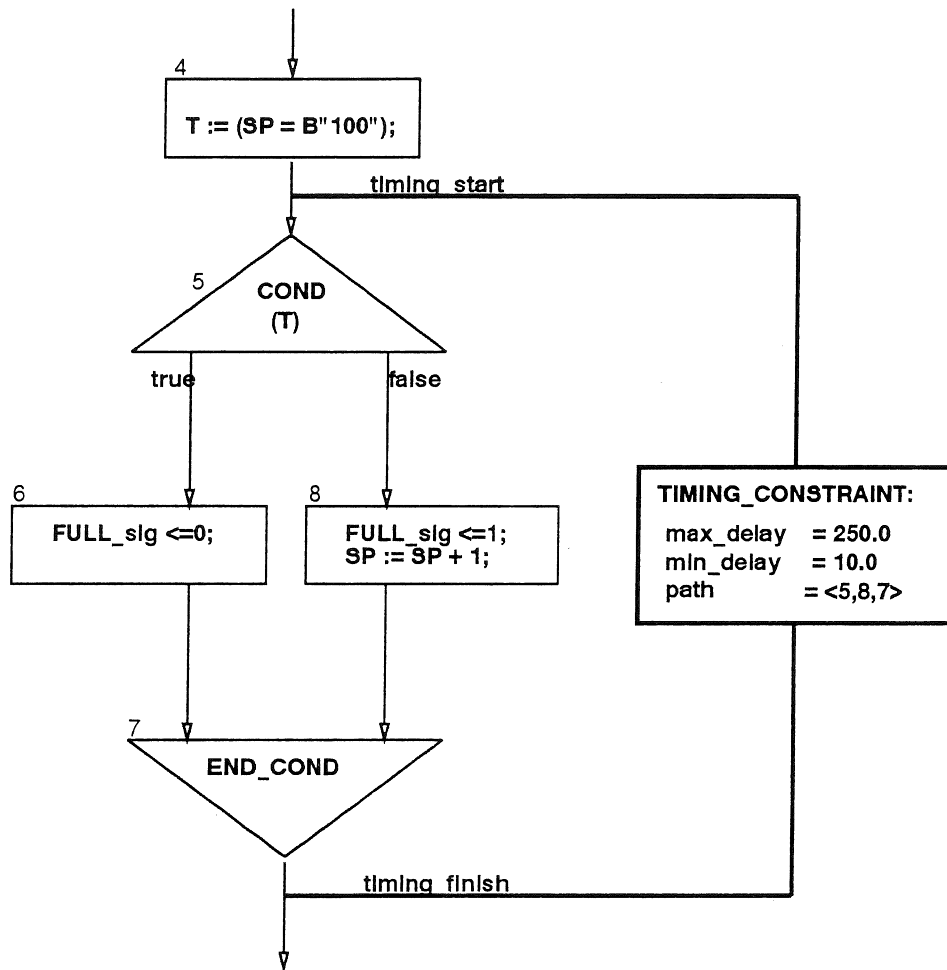

Figure 21: A Timing Constraint Specification at the CFG Level

specified to indicate this subset of possible paths. In this example, the timing constraint specifies delay values for the execution of the false branch only (that is, the right-hand side branch through the graph) rather than for the execution of the complete if-statement. Therefore, the list “$< 5, 8, 7 >$” which corresponds to the nodes on the constrained path is given. If no path has been specified, then the default of all possible paths that lie between the TIMING_START and the TIMING_FINISH is assumed. In this example, this would mean that the timing specification would impose a constraint on both the true and the false branch of the if-statement.

Figure 22 gives a more detailed view of the timing constraint shown in Figure 21. In particular, Figure 22 describes the details of the timing attributes as well as the implementation of the path expression. Since the timing constraint specifies delay values for the execution of the false branch only, a path expression indicating this path is needed. This path expression is implemented by an ordered list of object references to the control flow nodes that make up the path, namely, control flow nodes CF_NODE#6, CF_NODE#8, and CF_NODE#7. Another timing attribute is the timing type, which in this case corresponds to the value TT_CF_TO_CF. It indicates the object type of the source and the destination of the timing constraint. This is important since the same timing constraint construct is used to model timing constraints at both the control flow and the data flow level.

The BDEF description for this example design is shown in Figure 23. The timing constraints are inserted as independent design objects into the BDEF description. The specification of timing attributes for the nodes in the control/data flow graph, namely, the attributes TIMING_START and TIMING_FINISH is optional. Therefore, a designer can add timing constraints to an existing design representation graph by simply appending one or more of these timing constraint design objects to a BDEF description. This allows for a more precise and fine-grained timing specification than the specification of these constraints via high-level hardware description languages. In VHDL, for instance, the specification of timing constraints is extremely limited [6], since VHDL's timing constructs address simulation time only. Therefore, researchers have resolved to adding special-purpose constructs to the VDHL language to provide for a more suitable method of timing specification for synthesis. Our BDEF approach offers a simple solution to this problem, since the specification of timing could take place directly on the design representation level.

A BDEF SYNTAX FOR THE ECDFG MODEL

In this section, BDEF is used to specify Control/Data Flow Graph format that describe Control/Data Flow Graph design objects. This format is used to facilitate the exchange of design data between the Behavioral Design Data Base (BDDB) and design tools. A BNF syntax for this format is given next.

A.1 BDDB Design Entity Graph Information Syntax

A.2 BNF Syntax Introduction

The table below lists the meaning of the meta-symbols used in any BNF notation found in this paper.

<table>
<tr><td>::=</td><td>Indicates the equivalence of the left hand side of a statement to the right hand side of the statement.</td></tr>
<tr><td>|</td><td>Indicates an alternative.</td></tr>
<tr><td><></td><td>Encloses schema constructs to be supplied by the designer.</td></tr>
<tr><td>{}</td><td>Encloses an optional construct than can appear zero or more times.</td></tr>
<tr><td>[]</td><td>Encloses an optional construct than can appear at most once.</td></tr>
<tr><td>‘ ’</td><td>Encloses a symbol that is to be taken as literal.</td></tr>
</table>

A.3 BDDB Design Entity Graph Information Syntax

Design Entity File

```
top                          : '(' BDDB_entity_header
                               BDDB_design_data_objects
                               ')'
                             ;
```

Design Entity Header

```
BDDB_entity_header           : '[' BDDB_object_entity_defs ']'
                             ;
```

List of Characteristics of the Design Entity Object

```
BDDB_object_entity_defs      : BDDB_object_entity_def
                             | BDDB_object_entity_defs "," BDDB_object_entity_def
                             ;
```

Characteristics of the Design Entity Object

```
BDDB_object_entity_def       : DE_NAME ":" identifier
                             | DE_VERSION_NUM ":" unsigned_integer
                             | DD_DOMAIN_TYPE
                               ":" DD_domain_type_spec
                             | DD_BEHAVIORAL_FLAVOR
                               ":" DD_behavioral_flavor_spec
                             | DD_CHUNK_TYPE
                               ":" DD_chunk_type_spec
                             ;
```

Cf_Node Reference

cf_node_list	:	cf_node_ref
	;	

A.5 BDDB Control Flow Graph Information Syntax

Control Flow Node

```
cf_node_object          :  '(' cf_node_header cf_node_attributes ')'
                        ;
```

Control Flow Node Header

```
cf_node_header          :  CF_NODE '#' unsigned_integer
                        ;
```

Control Flow Node Attributes

```
cf_node_attributes      :  cf_node_attribute
                        |  cf_node_attributes ',' cf_node_attribute
                        ;
```

Control Flow Node Attribute

```
cf_node_attribute       :  STATE_REF ':' state_ref
                        |  CF_NODE_TYPE ':' cf_node_type_spec
                        |  NUM_CF_COND_ITEMS ':' unsigned_integer
                        |  NUM_INPUTS ':' unsigned_integer
                        |  NUM_OUTPUTS ':' unsigned_integer
                        |  DF_STYLE ':' df_style_spec
                        |  GRAPHICS_INFO ':' graphics_info_spec
                        |  TOOL_INFO ':' '[' tool_info_attributes ']'
                        |  INPUT1 ':' '<' listof_cf_conns '>'
                        |  OUTPUT1 ':''<' listof_cf_conns '>'
                        |  CF_COND_ITEM ':' '<'listof_cf_cond_item '>'
                        |  DF_NODE_GROUP ':''<'listof_df_node_group '>'
                        ;
```

List of Data Flow Node Groups

```
listof_df_node_group        :  df_node_group
                            |  listof_df_node_group ',' 
                               df_node_group
                            ;
```

Data Flow Node Group

```
df_node_group               :  '[' df_node_group_attributes ']'
                            ;
```

Data Flow Node Group Attributes

```
df_node_group_attributes    :  df_node_group_attribute
                            |  df_node_group_attributes ','
                               df_node_group_attribute
                            ;
```

Data Flow Node Group Attribute

```
df_node_group_attribute     :  DF_STMNTS ':'
                               '<' listof_df_node_net_refs '>'
                            |  CF_NODE_REF ':' cf_node_ref
                            |  CONDITION ':' identifier
                            ;
```

List of Data Flow Node Reference Objects

```
listof_df_node_net_refs     :  df_node_net_ref
                            |  listof_df_node_net_refs ',' df_node_net_ref
                            ;
```

Data Flow Node Reference Object

```
df_node_net_ref                      :  '[' df_node_net_ref_header
                                        df_node_net_ref_attributes ']'
                                     ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header               :  /* empty */
                                     ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes       :
                                        df_node_net_ref_attribute
                                     |  listof_df_node_net_ref_attributes ','
                                        df_node_net_ref_attribute
                                     ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute        :
                                        SYMBOL ':' string_literal
                                     |  TDF_NODE_REF ':'df_node_ref
                                     ;
```

List of Control Flow Node Connections

```
listof_cf_conns                      :  cf_conns_object
                                     |  listof_cf_conns ',' cf_conns_object
                                     ;
```

Control Flow Connection Node

```
cf_conns_object                      :  '(' cf_conns_header cf_conns_attributes ')'
                                     ;
```

List of Data Flow Node Groups

```
listof_df_node_group        :  df_node_group
                            |  listof_df_node_group ',' 
                               df_node_group
                            ;
```

Data Flow Node Group

```
df_node_group               :  '[' df_node_group_attributes ']'
                            ;
```

Data Flow Node Group Attributes

```
df_node_group_attributes    :  df_node_group_attribute
                            |  df_node_group_attributes ','
                               df_node_group_attribute
                            ;
```

Data Flow Node Group Attribute

```
df_node_group_attribute     :  DF_STMNTS ':'
                               '<' listof_df_node_net_refs '>'
                            |  CF_NODE_REF ':' cf_node_ref
                            |  CONDITION ':' identifier
                            ;
```

List of Data Flow Node Reference Objects

```
listof_df_node_net_refs     :  df_node_net_ref
                            |  listof_df_node_net_refs ',' df_node_net_ref
                            ;
```

Data Flow Node Reference Object

```
df_node_net_ref                 :  '[' df_node_net_ref_header
                                   df_node_net_ref_attributes ']'
                                ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header          :  /* empty */
                                ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes       :
                                   df_node_net_ref_attribute
                                |  listof_df_node_net_ref_attributes ','
                                   df_node_net_ref_attribute
                                ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute        :
                                   SYMBOL ':' string_literal
                                |  TDF_NODE_REF ':'df_node_ref
                                ;
```

List of Control Flow Node Connections

```
listof_cf_conns                 :  cf_conns_object
                                |  listof_cf_conns ',' cf_conns_object
                                ;
```

Control Flow Connection Node

```
cf_conns_object                 :  '(' cf_conns_header cf_conns_attributes ')'
                                ;
```

Control Flow Connection Node Header

```
cf_conns_header               :  CF_CONNS '#' unsigned_integer
                              ;
```

Control Flow Connection Node Attributes

```
cf_conns_attributes           :  cf_conns_attribute ',' cf_conns_attributes
                              ;
```

Control Flow Connection Node Attribute

```
cf_conns_attribute            :  IO_CLASS ':' io_class_spec
                              |  PROCESSED ':' unsigned_integer ','
                              |  GUARD_VAL ':' identifier ','
                              |  TIMING_START ':' '<' listof_timing_info_refs '>'
                              |  TIMING_FINISH ':' <' listof_timing_info_refs '>'
                              |  CF_CONNS_REF ':' cf_conns_ref
                              ;
```

List of Control Flow Node Conditions

```
listof_cf_cond_item           :  cf_cond_item
                              |  listof_cf_cond_item ',' cf_cond_item
                              ;
```

Control Flow Node Condition

```
cf_cond_item                  :  '[' CF_COND_ITEM_EXPR ':' identifier ','
                                 CF_COND_ITEM_VALUE ':' identifier ']'
                              ;
```

Data Flow Node Reference Object

```
df_node_net_ref                 : '[' df_node_net_ref_header
                                  df_node_net_ref_attributes ']'
                                ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header          : /* empty */
                                ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes       :
                                  df_node_net_ref_attribute
                                | listof_df_node_net_ref_attributes ','
                                  df_node_net_ref_attribute
                                ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute        :
                                  SYMBOL ':' string_literal
                                | TDF_NODE_REF ':'df_node_ref
                                ;
```

List of Control Flow Node Connections

```
listof_cf_conns                 : cf_conns_object
                                | listof_cf_conns ',' cf_conns_object
                                ;
```

Control Flow Connection Node

```
cf_conns_object                 : '(' cf_conns_header cf_conns_attributes ')'
                                ;
```

Control Flow Connection Node Header

```
cf_conns_header          :  CF_CONNS '#' unsigned_integer
                         ;
```

Control Flow Connection Node Attributes

```
cf_conns_attributes      :  cf_conns_attribute ',' cf_conns_attributes
                         ;
```

Control Flow Connection Node Attribute

```
cf_conns_attribute       :  IO_CLASS ':' io_class_spec
                         |  PROCESSED ':' unsigned_integer ','
                         |  GUARD_VAL ':' identifier ','
                         |  TIMING_START ':' '<' listof_timing_info_refs '>'
                         |  TIMING_FINISH ':' <' listof_timing_info_refs '>'
                         |  CF_CONNS_REF ':' cf_conns_ref
                         ;
```

List of Control Flow Node Conditions

```
listof_cf_cond_item      :  cf_cond_item
                         |  listof_cf_cond_item ',' cf_cond_item
                         ;
```

Control Flow Node Condition

```
cf_cond_item             :  '[' CF_COND_ITEM_EXPR ':' identifier ','
                            CF_COND_ITEM_VALUE ':' identifier ']'
                         ;
```

Data Flow Node Reference Object

```
df_node_net_ref                 :  '[' df_node_net_ref_header
                                   df_node_net_ref_attributes ']'
                                ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header          :  /* empty */
                                ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes       :
                                   df_node_net_ref_attribute
                                |  listof_df_node_net_ref_attributes ','
                                   df_node_net_ref_attribute
                                ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute        :
                                   SYMBOL ':' string_literal
                                |  TDF_NODE_REF ':'df_node_ref
                                ;
```

List of Control Flow Node Connections

```
listof_cf_conns                 :  cf_conns_object
                                |  listof_cf_conns ',' cf_conns_object
                                ;
```

Control Flow Connection Node

```
cf_conns_object                 :  '(' cf_conns_header cf_conns_attributes ')'
                                ;
```

Control Flow Connection Node Header

```
cf_conns_header           :  CF_CONNS '#' unsigned_integer
                          ;
```

Control Flow Connection Node Attributes

```
cf_conns_attributes       :  cf_conns_attribute ',' cf_conns_attributes
                          ;
```

Control Flow Connection Node Attribute

```
cf_conns_attribute        :  IO_CLASS ':' io_class_spec
                          |  PROCESSED ':' unsigned_integer ','
                          |  GUARD_VAL ':' identifier ','
                          |  TIMING_START ':' '<' listof_timing_info_refs '>'
                          |  TIMING_FINISH ':' <' listof_timing_info_refs '>'
                          |  CF_CONNS_REF ':' cf_conns_ref
                          ;
```

List of Control Flow Node Conditions

```
listof_cf_cond_item       :  cf_cond_item
                          |  listof_cf_cond_item ',' cf_cond_item
                          ;
```

Control Flow Node Condition

```
cf_cond_item              :  '[' CF_COND_ITEM_EXPR ':' identifier ','
                             CF_COND_ITEM_VALUE ':' identifier ']'
                          ;
```

Data Flow Node Reference Object

```
df_node_net_ref                 :  '[' df_node_net_ref_header
                                   df_node_net_ref_attributes ']'
                                ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header          :  /* empty */
                                ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes   :
                                   df_node_net_ref_attribute
                                |  listof_df_node_net_ref_attributes ','
                                   df_node_net_ref_attribute
                                ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute    :
                                   SYMBOL ':' string_literal
                                |  TDF_NODE_REF ':'df_node_ref
                                ;
```

List of Control Flow Node Connections

```
listof_cf_conns                 :  cf_conns_object
                                |  listof_cf_conns ',' cf_conns_object
                                ;
```

Control Flow Connection Node

```
cf_conns_object                 :  '(' cf_conns_header cf_conns_attributes ')'
                                ;
```

Control Flow Connection Node Header

```
cf_conns_header          :  CF_CONNS '#' unsigned_integer
                         ;
```

Control Flow Connection Node Attributes

```
cf_conns_attributes      :  cf_conns_attribute ',' cf_conns_attributes
                         ;
```

Control Flow Connection Node Attribute

```
cf_conns_attribute       :  IO_CLASS ':' io_class_spec
                         |  PROCESSED ':' unsigned_integer ','
                         |  GUARD_VAL ':' identifier ','
                         |  TIMING_START ':' '<' listof_timing_info_refs '>'
                         |  TIMING_FINISH ':' <' listof_timing_info_refs '>'
                         |  CF_CONNS_REF ':' cf_conns_ref
                         ;
```

List of Control Flow Node Conditions

```
listof_cf_cond_item      :  cf_cond_item
                         |  listof_cf_cond_item ',' cf_cond_item
                         ;
```

Control Flow Node Condition

```
cf_cond_item             :  '[' CF_COND_ITEM_EXPR ':' identifier ','
                            CF_COND_ITEM_VALUE ':' identifier ']'
                         ;
```

Data Flow Node Reference Object

```
df_node_net_ref               :  '[' df_node_net_ref_header
                                 df_node_net_ref_attributes ']'
                              ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header        :  /* empty */
                              ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes   :
                                 df_node_net_ref_attribute
                              |  listof_df_node_net_ref_attributes ','
                                 df_node_net_ref_attribute
                              ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute    :
                                 SYMBOL ':' string_literal
                              |  TDF_NODE_REF ':'df_node_ref
                              ;
```

List of Control Flow Node Connections

```
listof_cf_conns               :  cf_conns_object
                              |  listof_cf_conns ',' cf_conns_object
                              ;
```

Control Flow Connection Node

```
cf_conns_object               :  '(' cf_conns_header cf_conns_attributes ')'
                              ;
```

Control Flow Connection Node Header

```
cf_conns_header          :  CF_CONNS '#' unsigned_integer
                         ;
```

Control Flow Connection Node Attributes

```
cf_conns_attributes      :  cf_conns_attribute ',' cf_conns_attributes
                         ;
```

Control Flow Connection Node Attribute

```
cf_conns_attribute       :  IO_CLASS ':' io_class_spec
                         |  PROCESSED ':' unsigned_integer ','
                         |  GUARD_VAL ':' identifier ','
                         |  TIMING_START ':' '<' listof_timing_info_refs '>'
                         |  TIMING_FINISH ':' <' listof_timing_info_refs '>'
                         |  CF_CONNS_REF ':' cf_conns_ref
                         ;
```

List of Control Flow Node Conditions

```
listof_cf_cond_item      :  cf_cond_item
                         |  listof_cf_cond_item ',' cf_cond_item
                         ;
```

Control Flow Node Condition

```
cf_cond_item             :  '[' CF_COND_ITEM_EXPR ':' identifier ','
                            CF_COND_ITEM_VALUE ':' identifier ']'
                         ;
```

Data Flow Node Reference Object

```
df_node_net_ref                 : '[' df_node_net_ref_header
                                  df_node_net_ref_attributes ']'
                                ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header          : /* empty */
                                ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes       :
                                  df_node_net_ref_attribute
                                | listof_df_node_net_ref_attributes ','
                                  df_node_net_ref_attribute
                                ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute        :
                                  SYMBOL ':' string_literal
                                | TDF_NODE_REF ':'df_node_ref
                                ;
```

List of Control Flow Node Connections

```
listof_cf_conns                 : cf_conns_object
                                | listof_cf_conns ',' cf_conns_object
                                ;
```

Control Flow Connection Node

```
cf_conns_object                 : '(' cf_conns_header cf_conns_attributes ')'
                                ;
```

Control Flow Connection Node Header

```
cf_conns_header          :  CF_CONNS '#' unsigned_integer
                         ;
```

Control Flow Connection Node Attributes

```
cf_conns_attributes      :  cf_conns_attribute ',' cf_conns_attributes
                         ;
```

Control Flow Connection Node Attribute

```
cf_conns_attribute       :  IO_CLASS ':' io_class_spec
                         |  PROCESSED ':' unsigned_integer ','
                         |  GUARD_VAL ':' identifier ','
                         |  TIMING_START ':' '<' listof_timing_info_refs '>'
                         |  TIMING_FINISH ':' <' listof_timing_info_refs '>'
                         |  CF_CONNS_REF ':' cf_conns_ref
                         ;
```

List of Control Flow Node Conditions

```
listof_cf_cond_item      :  cf_cond_item
                         |  listof_cf_cond_item ',' cf_cond_item
                         ;
```

Control Flow Node Condition

```
cf_cond_item             :  '[' CF_COND_ITEM_EXPR ':' identifier ','
                            CF_COND_ITEM_VALUE ':' identifier ']'
                         ;
```

Data Flow Node Reference Object

```
df_node_net_ref                    : '[' df_node_net_ref_header
                                     df_node_net_ref_attributes ']'
                                   ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header             : /* empty */
                                   ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes       :
                                     df_node_net_ref_attribute
                                   | listof_df_node_net_ref_attributes ','
                                     df_node_net_ref_attribute
                                   ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute        :
                                     SYMBOL ':' string_literal
                                   | TDF_NODE_REF ':'df_node_ref
                                   ;
```

List of Control Flow Node Connections

```
listof_cf_conns                    : cf_conns_object
                                   | listof_cf_conns ',' cf_conns_object
                                   ;
```

Control Flow Connection Node

```
cf_conns_object                    : '(' cf_conns_header cf_conns_attributes ')'
                                   ;
```

Control Flow Connection Node Header

```
cf_conns_header          :  CF_CONNS '#' unsigned_integer
                         ;
```

Control Flow Connection Node Attributes

```
cf_conns_attributes      :  cf_conns_attribute ',' cf_conns_attributes
                         ;
```

Control Flow Connection Node Attribute

```
cf_conns_attribute       :  IO_CLASS ':' io_class_spec
                         |  PROCESSED ':' unsigned_integer ','
                         |  GUARD_VAL ':' identifier ','
                         |  TIMING_START ':' '<' listof_timing_info_refs '>'
                         |  TIMING_FINISH ':' <' listof_timing_info_refs '>'
                         |  CF_CONNS_REF ':' cf_conns_ref
                         ;
```

List of Control Flow Node Conditions

```
listof_cf_cond_item      :  cf_cond_item
                         |  listof_cf_cond_item ',' cf_cond_item
                         ;
```

Control Flow Node Condition

```
cf_cond_item             :  '[' CF_COND_ITEM_EXPR ':' identifier ','
                            CF_COND_ITEM_VALUE ':' identifier ']'
                         ;
```

Data Flow Node Reference Object

```
df_node_net_ref                  :  '[' df_node_net_ref_header
                                    df_node_net_ref_attributes ']'
                                 ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header           :  /* empty */
                                 ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes        :
                                    df_node_net_ref_attribute
                                 |  listof_df_node_net_ref_attributes ','
                                    df_node_net_ref_attribute
                                 ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute         :
                                    SYMBOL ':' string_literal
                                 |  TDF_NODE_REF ':'df_node_ref
                                 ;
```

List of Control Flow Node Connections

```
listof_cf_conns                  :  cf_conns_object
                                 |  listof_cf_conns ',' cf_conns_object
                                 ;
```

Control Flow Connection Node

```
cf_conns_object                  :  '(' cf_conns_header cf_conns_attributes ')'
                                 ;
```

Control Flow Connection Node Header

```
cf_conns_header          :  CF_CONNS '#' unsigned_integer
                         ;
```

Control Flow Connection Node Attributes

```
cf_conns_attributes      :  cf_conns_attribute ',' cf_conns_attributes
                         ;
```

Control Flow Connection Node Attribute

```
cf_conns_attribute       :  IO_CLASS ':' io_class_spec
                         |  PROCESSED ':' unsigned_integer ','
                         |  GUARD_VAL ':' identifier ','
                         |  TIMING_START ':' '<' listof_timing_info_refs '>'
                         |  TIMING_FINISH ':' <' listof_timing_info_refs '>'
                         |  CF_CONNS_REF ':' cf_conns_ref
                         ;
```

List of Control Flow Node Conditions

```
listof_cf_cond_item      :  cf_cond_item
                         |  listof_cf_cond_item ',' cf_cond_item
                         ;
```

Control Flow Node Condition

```
cf_cond_item             :  '[' CF_COND_ITEM_EXPR ':' identifier ','
                            CF_COND_ITEM_VALUE ':' identifier ']'
                         ;
```

Data Flow Node Reference Object

```
df_node_net_ref                 : '[' df_node_net_ref_header
                                  df_node_net_ref_attributes ']'
                                ;
```

Data Flow Node Reference Header

```
df_node_net_ref_header          : /* empty */
                                ;
```

List of Data Flow Node Reference Attributes

```
listof_df_node_net_ref_attributes   :
                                  df_node_net_ref_attribute
                                | listof_df_node_net_ref_attributes ','
                                  df_node_net_ref_attribute
                                ;
```

List of Data Flow Node Reference Attribute

```
listof_df_node_net_ref_attribute    :
                                  SYMBOL ':' string_literal
                                | TDF_NODE_REF ':'df_node_ref
                                ;
```

List of Control Flow Node Connections

```
listof_cf_conns                 : cf_conns_object
                                | listof_cf_conns ',' cf_conns_object
                                ;
```

Control Flow Connection Node

```
cf_conns_object                 : '(' cf_conns_header cf_conns_attributes ')'
                                ;
```

Control Flow Connection Node Header

cf_conns_header : CF_CONNS '#' unsigned_integer
;

Control Flow Connection Node Attributes

cf_conns_attributes : cf_conns_attribute ',' cf_conns_attributes
;

Control Flow Connection Node Attribute

cf_conns_attribute : IO_CLASS ':' io_class_spec
| PROCESSED ':' unsigned_integer ','
| GUARD_VAL ':' identifier ','
| TIMING_START ':' '<' listof_timing_info_refs '>'
| TIMING_FINISH ':' <' listof_timing_info_refs '>'
| CF_CONNS_REF ':' cf_conns_ref
;

List of Control Flow Node Conditions

listof_cf_cond_item : cf_cond_item
| listof_cf_cond_item ',' cf_cond_item
;

Control Flow Node Condition

cf_cond_item : '[' CF_COND_ITEM_EXPR ':' identifier ','
CF_COND_ITEM_VALUE ':' identifier ']'
;

Enumerated Values for Control Flow Node Type

```
cf_node_type_spec          :  CF_START
                           |  CF_END
                           |  IF_TEST
                           |  IF_JOIN
                           |  LOOP_TEST
                           |  STMT_BLK
                           |  CASE_SELECT
                           |  CASE_JOIN
                           |  TEST
                           |  COND
                           |  ASYNC_EVENT
                           |  ASYNC_OP
                           |  BLK_START
                           |  BLK_END
                           |  PROC_START
                           |  PROC_END
                           ;
```

Enumerated Values for Data Flow Style

```
df_style_spec              :  SEQ
                           |  CONCUR
                           ;
```

A.6 BDDB Data Flow Graph Information Syntax

Data Flow Node Object

df_node_object	:	'(' df_node_header df_node_net_attributes df_node_info ')'
	;	

Data Flow Net Object

df_net_object	:	'(' df_net_header df_node_net_attributes df_net_info ')'
	;	

Data Flow Node Header

df_node_header	:	DF_NODE '#' unsigned_integer
	;	

Data Flow Net Header

df_net_header	:	DF_NET '#' unsigned_integer
	;	

Data Flow Node Attributes

df_node_net_attributes	:	df_node_net_attribute
	|	df_node_net_attributes ',' df_node_net_attribute
	;	

Data Flow Node Attribute

```
df_node_net_attribute     : NODE_CLASS ':' node_class_spec
                          | GRAPH_TYPE ':' graph_type_spec
                          | CF_NODE_REF ':' cf_node_ref
                          | NUM_INPUTS ':' unsigned_integer
                          | NUM_OUTPUTS ':' unsigned_integer
                          | GRAPHICS_INFO ':' graphics_info_spec
                          | COMP_INFO ':' '<' comp_info_objects '>'
                          | TOOL_INFO ':' '[' tool_info_attributes ']'
                          | INPUT1 ':' '<' listof_df_io_ports '>'
                          | OUTPUT1 ':' '<' listof_df_io_ports '>'
                          | TIMING_START ':' '<' listof_timing_info_refs '>'
                          | TIMING_FINISH ':' <' listof_timing_info_refs '>'
                          ;
```

List of References to Timing Constraints

```
listof_timing_info_refs   : timing_info_ref
                          | listof_timing_info_refs ',' timing_info_ref
                          ;
```

Information specific to the Data Flow Node

```
df_node_info              : /*empty*/
                          | ',' DF_NODE_INFO ':'
                            '[' df_node_info_spec ']'
                          ;
```

Information specific to the Data Flow Net

```
df_net_info               : /*empty*/
                          | ',' DF_NET_INFO ':'
                            '[' df_net_info_spec ']'
                          ;
```

List of Data Flow Connections

```
listof_df_io_conns        : df_io_conn
                          | listof_df_io_conns ',' df_io_conn
                          ;
```

Data Flow Connection

```
df_io_conn                : df_port_ref
                          ;
```

List of Data Flow Guard Information

```
listof_guard_info         : guard_info
                          | listof_guard_info ',' guard_info
                          ;
```

Data Flow Guard Information

```
guard_info                : '<' listof_guard_val '>'
                          ;
```

List of Data Flow Guard Values

```
listof_guard_val          : guard_val
                          | listof_guard_val ',' guard_val
                          ;
```

Data Flow Guard Value

```
guard_val                 : identifier
                          ;
```

Graphics Information Specification

```
graphics_info_spec          :  '[' graphics_info_attribute ']'
                            ;
```

Graphics Information Attribute

```
graphics_info_attribute     :  BID_NUMBER ':' unsigned_integer
                            ;
```

Tool Information Attributes

```
tool_info_attributes        :  tool_info_attribute
                            |  tool_info_attributes ',' tool_info_attribute
                            ;
```

Tool Information Attribute

```
tool_info_attribute         :  VSS_TOOL_INFO ':' vss_tool_info_spec
                            ;
```

Tool Information State Number

```
vss_tool_info_spec          :  unsigned_integer
                            ;
```

List of Component Allocation Information

```
comp_info_objects           :  comp_info_object
                            |  comp_info_objects ',' comp_info_object
                            ;
```

Component Allocation Information

```
comp_info_object          :  '[' comp_info_attributes ']'
                          ;
```

Component Allocation Attributes

```
comp_info_attributes      :  comp_info_attribute
                          |  comp_info_attributes ',' comp_info_attribute
                          ;
```

Component Allocation Attribute

```
comp_info_attribute       :  NET_NUM ':' unsigned_integer
                          |  COMMUTE_INPS ':' unsigned_integer
                          |  INST_NAME ':' string_literal
                          ;
```

List of Switchbox Bits

```
listof_switchbox_bits     :  switchbox_bits
                          |  listof_switchbox_bits ',' switchbox_bits
                          ;
```

Switchbox Bits

```
switchbox_bits            :  '[' LEFT_BIT ':' unsigned_integer ','
                             RIGHT_BIT ':' unsigned_integer ']'
                          ;
```

List of Data Flow Operation Information

```
listof_df_op_info         :  df_op_info
                          |  listof_df_op_info ',' df_op_info
                          ;
```

Data Flow Operation Information Object

```
df_op_info               :  '[' df_op_info_header df_op_info_attributes ' ']'
                         ;
```

Data Flow Operation Information Attributes

```
df_op_info_attributes    :  df_op_info_attribute
                         |  df_op_info_attributes ',' df_op_info_attribute
                         ;
```

Data Flow Operation Information Attributes

```
df_op_info_attribute     :  OP_CLASS ':' op_class_spec
                         |  OP_TYPE ':' op_type_spec
                         |  OP_NAME ':' string_literal
                         |  COND_INFO ':' '<' listof_cond_val_info '>'
                         ;
```

List of Data Flow Condition Value Information

```
listof_cond_val_info     :  cond_val_info
                         |  listof_cond_val_info ',' cond_val_info
                         ;
```

Data Flow Condition Value Information

```
cond_val_info            :  '<' listof_cond_val_pair '>'
                         ;
```

List of Data Flow Condition Value Pairs

```
listof_cond_val_pair     :  cond_val_pair
                         |  listof_cond_val_pair ',' cond_val_pair
                         ;
```

Data Flow Condition Value Pair

cond_val_pair	:	'[' CONDITION ':' identifier ',' VALUE ':' identifier ']'
	;	

A.7 BDDB Timing Constraint Graph Syntax

Timing Constraints for CFGs and DFGs

```
timing_info_object          :  '(' timing_info_header timing_info_attributes ')'
                            ;
```

Timing Information Object Header

```
timing_info_header          :  TIMING_INFO # unsigned_integer
                            ;
```

Timing Information Object Attributes

```
timing_info_attributes      :  timing_info_attribute
                            |  timing_info_attributes ',' timing_info_attribute
                            ;
```

Timing Information Object Attribute

```
timing_info_attribute       :  SOURCE_DF ':' df_net_ref
                            |  DEST_DF ':' df_net_ref
                            |  SOURCE_CF ':' cf_conns_ref
                            |  DEST_CF ':' cf_conns_ref
                            |  MAX_DELAY ':' real_number
                            |  MIN_DELAY ':' real_number
                            |  NOM_DELAY ':' real_number
                            |  SOURCE_EVENT ':' timing_event_spec
                            |  DEST_EVENT ':' timing_event_spec
                            |  TIMING_TYPE ':' timing_type_spec
                            |  DF_PATHS ':' '<' listof_df_path_info '>'
                            |  CF_PATHS ':' '<' listof_cf_path_info '>'
                            ;
```

DFG Paths Expression for Timing

```
listof_df_path_info         :  df_path_info
                            ;
```

DFG Path Expression for Timing

```
df_path_info                :  '<' listof_df_path_refs '>'
                            ;
```

DFG References in a Path Expression For Timing

```
listof_df_path_refs         :  df_path_ref
                            |  listof_df_path_refs ',' df_path_ref
                            ;
```

DFG Reference in a Path Expression

```
df_path_ref                 :  df_node_ref
                            ;
```

CFG Paths Expression for Timing

```
listof_cf_path_info         :  cf_path_info
                            ;
```

CFG Path Expression for Timing

```
cf_path_info                :  '<' listof_cf_path_refs '>'
                            ;
```

CFG References in a Path Expression For Timing

```
listof_cf_path_refs    :  cf_path_ref
                       |  listof_cf_path_refs ',' cf_path_ref
                       ;
```

CFG Reference in a Path Expression

```
cf_path_ref            :  cf_node_ref
                       ;
```

A.8 BDDB Data Type Enumeration Values

Enumerated Values for Node Class

```
node_class_spec        : DF_OP
                       | DF_ARC
                       ;
```

Enumerated Values for Graph Type

```
graph_type_spec        : ASYNC
                       | SYNC
                       ;
```

Enumerated Values for Signal Type

```
sig_type_spec          : SIMPLE
                       | CONSTANT
                       | SIGNAL
                       | WIRE
                       | REGISTER
                       | BUS
                       | PORT
                       | WIRED_OR
                       | VARIABLE
                       | MEMORY
                       ;
```

Enumerated Values for Node Type

```
node_type_spec        : DATA_ACCESS
                      | SELECT
                      | DELAY
                      | OPERATION
                      | MARKER
                      | FUNC_CALL
                      | MISC
                      ;
```

Enumerated Values for Operation Class

```
op_class_spec         : RELATIONAL
                      | ADDING
                      | LOGICAL
                      | MULTIPLYING
                      | SHIFTING
                      | MISC_OP
                      ;
```

Enumerated Values for Operation Type

```
op_type_spec        : EQ
                    | NEQ
                    | LT
                    | LEQ
                    | TGT
                    | GEQ
                    | AND
                    | NAND
                    | NOR
                    | NOT
                    | OR
                    | XOR
                    | ADD
                    | SUB
                    | CONCAT
                    | INC
                    | DEC
                    | MULT
                    | DIV
                    | SHL0
                    | SHL1
                    | SHR0
                    | SHR1
                    | SHL
                    | SHR
                    | ABS
                    | MOD
                    | READ
                    | WRITE
                    | CH_VALUE
                    | SWITCH_BOX
                    | DECODER
                    | DELTA
                    | TRI_STATE
                    | DF_BLK_START
                    | DF_BLK_END
                    | FUNC
                    | RISING
                    | FALLING
                    | TIMER
                    | TIMEOUT
                    | EVENT
                    | TRUTH_TABLE
                    ;
```

Enumerated Values for Net Type

```
net_type_spec        : DATA_NET
                     | CLK_NET
                     | SET_NET
                     | RESET_NET
                     | CONTROL_NET
                     | DATA_DEP_NET
                     ;
```

Enumerated Values for Sensitivity

```
sense_spec           : LEVEL
                     | EDGE
                     ;
```

Enumerated Values for Active Edge

```
act_edge_spec        : POSITIVE
                     | NEGATIVE
                     ;
```

Enumerated Values for Port Type

```
port_type_spec       : CLOCK_PORT
                     | SELECT_PORT
                     | DATA_PORT
                     | INDEX_PORT
                     | RESET_PORT
                     | SET_PORT
                     | INC_PORT
                     | DEC_PORT
                     | DEP_PORT
                     ;
```

Enumerated Values for Number Representation

```
num_repres_spec        : BINARY
                       | FLOATING_PT
                       | INTEGER
                       ;
```

Enumerated Values for Numeric Format

```
num_format_spec        : ONES_COMP
                       | TWOS_COMP
                       | SIGN_MAG
                       | MAG
                       ;
```

Enumerated Values for Declaration Types

```
decl_types_spec        : INT
                       | BIT
                       | BIT_VECTOR
                       | BOOLEAN
                       | CHARACTER
                       | NATURAL
                       | POS
                       | REAL
                       | STRING
                       | TIME
                       | VSS_CLOCK
                       | SET
                       | RESET
                       | ATTRIBUTE
                       | TYPE_DEF
                       | MEM
                       | OCTAL
                       | HEX
                       | LITERAL
                       ;
```

Enumerated Values for IO Class

```
io_class_spec          : INPUT
                       | OUTPUT
                       ;
```

Enumerated Values for Timing Events

```
timing_events_spec     : TE_UNDEFINED
                       | TE_CHANGING
                       | TE_RISING
                       | TE_FALLING
                       | TE_CONTROL_FLOW
                       ;
```

Enumerated Values for Timing Types

```
timing_types_spec      : TT_UNDEFINED
                       | TT_DF_TO_DF
                       | TT_DF_TO_CF
                       | TT_CF_TO_DF
                       | TT_CF_TO_CF
                       ;
```

A.9 Object References using Object Identity

Lex rule for state object reference

```
state_ref          :  "#"STATE"#"digit+"#"
                   ;
```

Lex rule for data flow port object reference

```
df_port_ref        :  "#"DF_PORT"#"digit+"#"
                   ;
```

Lex rule for control flow connection object reference

```
cf_conns_ref       :  "#"CF_CONNS"#"digit+"#"
                   ;
```

Lex rule for data flow node object reference

```
df_node_ref        :  "#"DF_NODE"#"digit+"#"
                   ;
```

Lex rule for data flow net object reference

```
df_net_ref         :  "#"DF_NET"#"digit+"#"
                   ;
```

Lex rule for control flow node object reference

```
cf_node_ref        :  "#"CF_NODE"#"digit+"#"
                   ;
```

Lex rule for timing info object reference

```
timing_info_ref        :  "#"TIMING_INFO"#"digit+"#"
                       ;
```

A.10 General Constructs

Lex rule for identifier

identifier	:	[a-zA-Z](”_”?([a-zA-Z]|[0-9]))*
	;	

Lex rule for string literals

string_literal	:	(‘”’([a-zA-Z]|[0-9]|(‘+’)|(‘-’)|‘_’|” ”)* ‘”’)
	;	

Lex rule for unsigned integer

unsigned_integer	:	([0-9])+
	;	

Lex rule for real number

real_number	:	[+-]?([0-9])+“.”([0-9])+ “E”[+-]?([0-9])+
	|	[+-]?([0-9])+ “E”[+-]?([0-9])+
	|	[+-]?([0-9])+“.”([0-9])+
	;	

B OBJECT TYPE DEFINITIONS FOR THE ECDFG SCHEMA

This section describes the schema for the ECDFG model. Note that the term "REFERENCE" means that the referenced object is defined independently, i.e., that there is an identifier for the referenced object type. If an object type name is used directly to define the attribute of another object type then the definition of the former object is directly nested within the definition of the later.

B.1 State Transition Graph

This section gives the schema for the State Transition Graph.

```
DEFINE TYPE StateGraph
DEFINED BY TUPLEOF (
  Id: INTEGER,
  Num_States: INTEGER,
  Num_Cf_Nodes: INTEGER,
  Graph_Type: GrType,
  States:  LISTOF ( REFERENCE StateNode  )
  Cf_Nodes:  LISTOF ( REFERENCE CfNode )
)
```

```
DEFINE TYPE StateNode
DEFINED BY TUPLEOF (
  Id: INTEGER,
  Graph_Ptr: REFERENCE StateGraph,
  Graph_Type: GrType,
  async_succ_state: INTEGER,
  num_cf_nodes: INTEGER,
  cf_node1: LISTOF (REFERENCE CfNode ),
  async_event_cf: REFERENCE CfNode,
  async_op_cf: REFERENCE CfNode,
)
```

B.2 Control Flow Graph

In this section, I describe the schema for the Control Flow object types.

```
DEFINE TYPE CfNode
SUPERTYPES: CfConstruct
DEFINED BY TUPLEOF (
  Id: INTEGER,
  State: REFERENCE StateNode,
  type: Cf_Node_Type,
  num_cf_cond_items: INTEGER,
  cf_cond_item1: LISTOF ( CfCondItem ),
  num_inputs: INTEGER,
  num_outputs: INTEGER,
  input1: SETOF (REFERENCE CfConns) ,
  output1: LISTOF (REFERENCE CfConns),
  df_stmnt1: LISTOF ( DfNodeList ),
  df_style: Df_Style,
  trav_flag: INTEGER,
  graph_info: GraphicsInfo,
  tool:  ToolInfo
)

DEFINE TYPE  CfConns
DEFINED BY TUPLEOF (
  Id: INTEGER,
  type: IOClass,
  conn: REFERENCE CfConns,
  guard_val: STRING,
  cf_node: REFERENCE CfNode,
  processed: INTEGER,
  timing_start: LISTOF (  REFERENCE  TimingInfo ),
  timing_finish: LISTOF (  REFERENCE  TimingInfo )
)

DEFINE TYPE  DfNodeList
DEFINED BY LISTOF ( REFERENCE DfNode )


DEFINE TYPE  CfCondItem
DEFINED BY TUPLEOF (
  expr: CHAR,
```

```
  value: CHAR
)
```

B.3 Data Flow Graph

In this section, I describe the schema for the Data Flow object types.

```
DEFINE TYPE DataFlowGraph
DEFINED BY TUPLEOF (
  Graph: SETOF ( DFGroup ),
  First: DFConstruct,
  Type: STRING
)

DEFINE TYPE DFGroup
DEFINED BY TUPLEOF (
  StateId: INTEGER,
  Condition: Condition,
  DFNodes: SETOF ( REFERENCE DFConstruct )
)


DEFINE TYPE DfConstruct
DEFINED BY TUPLEOF (
  Id: INTEGER,
  node_class: NodeClass,
  graph_type: GrType,
  trav_flag: INTEGER,
  cf_node: REFERENCE CfNode,
  num_inputs: INTEGER,
  num_outputs: INTEGER,
  input1: LISTOF (REFERENCE DfIoPort ),
  output1: LISTOF (REFERENCE DfIoPort ),
  comp_info: DfCompInfo;
  graph_info: GraphicsInfo,
  tool: ToolInfo,
  StrucAlloc: LISTOF ( StrucAllocationInfo ),
  ControlAlloc: LISTOF ( ControlAllocationInfo )
```

```
)

DEFINE TYPE DfNode
SUPERTYPES: DFConstruct
DEFINED BY TUPLEOF (
  node_info: DfNodeInfo,
)

DEFINE TYPE DfNet
SUPERTYPES: DFConstruct
DEFINED BY TUPLEOF (
  net_info: DfNetInfo,
  timing_start: LISTOF (  REFERENCE  TimingInfo ),
  timing_finish: LISTOF (  REFERENCE  TimingInfo )
)

DEFINE TYPE DfNodeInfo
DEFINED BY TUPLEOF (
   sig_name: STRING,
   sig_type: SigType,
   decl_type: DeclTypes,
   node_type: NodeType,
   num_ops: INTEGER,
   op1: LISTOF ( DfOpInfo  )
   truth_tbl: LISTOF ( DfOpInfo  )
   df_style: DfStyle,
   need_control: INTEGER,
   bit_field1:  SwitchboxBits
)

DEFINE TYPE DfOpInfo
DEFINED BY TUPLEOF (
  op_class: OpClass,
  op_type: OpType,
  op_name: STRING,
  cond1: LISTOF ( CondValInfo  ),
  node_info: REFERENCE DfNodeInfo
)

DEFINE TYPE CondValInfo
DEFINED BY TUPLEOF (
  cond_pair1: LISTOF ( CondValPair ),
  op_info: REFERENCE DfOpInfo
```

```
)

DEFINE TYPE CondValPair
DEFINED BY TUPLEOF (
  condition: STRING,
  value: STRING
)

DEFINE TYPE DfNetInfo
DEFINED BY TUPLEOF (
   bit_width: INTEGER,
   min_index: INTEGER,
   max_index: INTEGER,
   net_type: NetType,
   representation: NumRepres,
   format: NumFormat,
   sensitivity: Sense,
   active_edge: ActEdge,
   branch_expr: STRING
)

DEFINE TYPE DfCompInfo
DEFINED BY TUPLEOF (
   net_num: INTEGER,
   inst_name: STRING,
   commute_inps: INTEGER
)

DEFINE TYPE GraphicsInfo
DEFINED BY TUPLEOF (
   bid_number: INTEGER
)

DEFINE TYPE ToolInfo
DEFINED BY TUPLEOF (
  vss: INTEGER,
  extend: INTEGER,
  bif: INTEGER,
  csa: INTEGER,
  sehwa: INTEGER,
  timing_tool: INTEGER,
  transform: INTEGER
)
```

```
DEFINE TYPE DfIoPort
DEFINED BY TUPLEOF (
   id: INTEGER,
   port_name: STRING,
   port_class: IOClass,
   port_type: PortType,
   bit_width: INTEGER,
   num_conn: INTEGER,
   guard_val1:  LISTOF ( GuardInfo ),
   alloc_done: INTEGER,
   node_net: REFERENCE DfNodeNet,
   connected: SETOF ( REFERENCE  DfIoPort )
)

DEFINE TYPE GuardInfo
DEFINED BY TUPLEOF (
  guard_value: LISTOF ( GuardVal )
)

DEFINE TYPE GuardVal
DEFINED BY TUPLEOF (
  value: CHARACTER,
)

DEFINE TYPE StrucAllocationInfo
DEFINED BY TUPLEOF (
  StrucId: STRING,
  Operation: STRING,
  StrucIn: LISTOF ( STRING )
)

DEFINE TYPE ControlAllocationInfo
DEFINED BY TUPLEOF (
  ControlLineId: STRING,
  ControlValue: INTEGER
)
```

B.4 Timing Constraint Graph

The timing constraint graph represents timing constraints for both the control and the data flow graph.

```
DEFINE TYPE TimingInfo
DEFINED BY TUPLEOF (
  Id: INTEGER,
  Type:  TimingType,
  Source_Df: REFERENCE DFNet,
  Dest_Df: REFERENCE DFNet,
  Source_Cf: REFERENCE CFConns,
  Dest_Cf: REFERENCE CFConns,
  Min_Delay: double,
  Max_Delay: double,
  Nom_Delay: double,
  Source_Event: Timing_Event,
  Dest_Event: Timing_Event,
  Path: LISTOF ( PathInfo )
)

DEFINE TYPE PathInfo
DEFINED BY TUPLEOF (
  Df_Path: LISTOF ( DfNode ),
  Cf_Path: LISTOF ( CfNode )
)
```

C AN EXTENDED BDEF EXAMPLE DESCRIPTION

C.1 The VHDL Specification

A VHDL specification of the example design is given in below:

```
entity AM2910 is
  port (
      FULL_sig: out BIT
      );
end AM2910;

architecture BEHAVIOR of AM2910 is
begin
process
  variable SP : BIT_VECTOR(2 downto 0);

begin
    if (SP = B"100") then
      FULL_sig <= 0;
    else
      FULL_sig <= 1;
      SP := SP + 1;
    end if;
end process;
end BEHAVIOR;
```

A ECDFG representation of this VHDL specification generated by the VHDL Graph Compiler is shown in Figure 27.

```
DEFINE TYPE TimingInfo
DEFINED BY TUPLEOF (
  Id: INTEGER,
  Type:  TimingType,
  Source_Df: REFERENCE DFNet,
  Dest_Df: REFERENCE DFNet,
  Source_Cf: REFERENCE CFConns,
  Dest_Cf: REFERENCE CFConns,
  Min_Delay: double,
  Max_Delay: double,
  Nom_Delay: double,
  Source_Event: Timing_Event,
  Dest_Event: Timing_Event,
  Path: LISTOF ( PathInfo )
)

DEFINE TYPE PathInfo
DEFINED BY TUPLEOF (
  Df_Path: LISTOF ( DfNode ),
  Cf_Path: LISTOF ( CfNode )
)
```

C.2 The Data Flow View

There are three different data flow graphs that can be extracted from the design entity. The first graph implements the condition evaluation for the if-statement, i.e., the expression "SP = B'100". The second corresponds to the assignment statement "FULL_sig <= 0" that is evaluated when the if-condition is true, and the third graph corresponds to the two assignment statements "FULL_sig <= 1; SP := SP + 1;" that are evaluated when the if-condition is false. Below, we show the BDEF description for the second graph that has been generated by the BDEF Generator.

```
(
[
  DD_DOMAIN_TYPE: BEHAVIOR,
  DD_FLAVOR: BEHAVIOR_PURE,
  DD_CHUNK_TYPE: DATA_FLOW
]

(DF_NODE#18
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#8#,
NUM_INPUTS: 2,
INPUT1: <
  (DF_PORT#56
  IO_CLASS: INPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#57# >
  ),
  (DF_PORT#62
  IO_CLASS: INPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#63# >
  )>,
NUM_OUTPUTS: 0,
DF_NODE_INFO: [
  SIG_NAME: "DF_BLK_END",
  SIG_TYPE: SIGNAL,
  DECL_TYPES: INT,
  NODE_TYPE: MARKER,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: DF_BLK_END,
     OP_NAME: "DF_BLK_END"
     ]>
  ]
)

(DF_NODE#17
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#8#,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#48
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#49# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#60
  IO_CLASS: OUTPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#61# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "SP",
  SIG_TYPE: VARIABLE,
  DECL_TYPES: INT,
  NODE_TYPE: DATA_ACCESS,
  NUM_OPS: 0,
```

```
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: WRITE,
     OP_NAME: "SP"
     ]>
  ]
)

(DF_NET#15
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#61
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#60# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#63
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#62# >
  )>,
DF_NET_INFO: [
  BIT_WIDTH: 1,
  NET_TYPE: DATA_DEP_NET,
  NUM_REPRES: BINARY,
]
)

(DF_NODE#16
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#8#,
NUM_INPUTS: 2,
INPUT1: <
  (DF_PORT#42
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#43# >
  ),
  (DF_PORT#44
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#45# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#46
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#47# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "+",
  SIG_TYPE: SIGNAL,
  DECL_TYPES: INT,
  NODE_TYPE: OPERATION,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: ADDING,
     OP_TYPE: ADD,
     OP_NAME: "+"
     ]>
  ]
)

(DF_NET#12
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#47
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#46# >
  )>,
```

```
 NUM_OUTPUTS: 1,
 OUTPUT1: <
   (DF_PORT#49
   IO_CLASS: OUTPUT,
   PORT_TYPE: DATA_PORT,
   BIT_WIDTH: 3,
   NUM_CONN: 1,
   CONN1: < ##DF_PORT#48# >
   )>,
 DF_NET_INFO: [
   BIT_WIDTH: 3,
   NET_TYPE: DATA_NET,
   NUM_REPRES: BINARY,
 ]
 )

(DF_NODE#15
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#8#,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#59
  IO_CLASS: INPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#53# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#40
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#41# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "1",
  SIG_TYPE: CONSTANT,
  DECL_TYPES: INT,
  NODE_TYPE: DATA_ACCESS,
  NUM_OPS: 0,
  DF_OP_INFO: <
      [OP_CLASS: MISC_OP,
      OP_TYPE: READ,
      OP_NAME: "1"
      ]>
  ]
)

(DF_NET#11
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#41
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#40# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#45
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#44# >
  )>,
DF_NET_INFO: [
  BIT_WIDTH: 1,
  NET_TYPE: DATA_NET,
  NUM_REPRES: BINARY,
]
)

(DF_NODE#14
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#8#,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#58
  IO_CLASS: INPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
```

```
  CONN1: < ##DF_PORT#53# >
  )>,
 NUM_OUTPUTS: 1,
 OUTPUT1: <
  (DF_PORT#38
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#39# >
  )>,
 DF_NODE_INFO: [
  SIG_NAME: "SP",
  SIG_TYPE: VARIABLE,
  DECL_TYPES: INT,
  NODE_TYPE: DATA_ACCESS,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: READ,
     OP_NAME: "SP"
     ]>
  ]
)

(DF_NET#10
NODE_CLASS: DF_ARC,
GRAPH_TYPE: ASYNC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#39
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#38# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#43
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#42# >
  )>,
DF_NET_INFO: [
  BIT_WIDTH: 3,
  NET_TYPE: DATA_NET,
  NUM_REPRES: BINARY,
]
)

(DF_NODE#13
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#8#,
GRAPH_TYPE: SYNC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#36
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#37# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#54
  IO_CLASS: OUTPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#55# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "FULL_SIG",
  SIG_TYPE: PORT,
  DECL_TYPES: INT,
  NODE_TYPE: DATA_ACCESS,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: WRITE,
     OP_NAME: "FULL_SIG"
     ]>
  ]
)
```

```
(DF_NET#14
NODE_CLASS: DF_ARC,
GRAPH_TYPE: ASYNC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#55
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#54# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#57
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#56# >
  )>,
DF_NET_INFO: [
  BIT_WIDTH: 1,
  NET_TYPE: DATA_DEP_NET,
  NUM_REPRES: BINARY,
]
)

(DF_NODE#12
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#8#,
GRAPH_TYPE: SYNC,
NUM_INPUTS: 0,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#50
  IO_CLASS: OUTPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#51# >
  )>,
DF_NODE_INFO: [
```

```
  SIG_NAME: "DF_BLK_START",
  SIG_TYPE: SIGNAL,
  DECL_TYPES: INT,
  NODE_TYPE: MARKER,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: DF_BLK_START,
     OP_NAME: "DF_BLK_START"
     ]>
  ]
)

(DF_NET#13
NODE_CLASS: DF_ARC,
GRAPH_TYPE: ASYNC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#51
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#50# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#53
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 3,
  CONN1: < ##DF_PORT#52#,
           ##DF_PORT#58#,
           ##DF_PORT#59# >
  )>,
DF_NET_INFO: [
  BIT_WIDTH: 1,
  NET_TYPE: DATA_DEP_NET,
  NUM_REPRES: BINARY,
]
)

(DF_NODE#11
```

```
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#8#,
GRAPH_TYPE: SYNC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#52
  IO_CLASS: INPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#53# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#34
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#35# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "1",
  SIG_TYPE: CONSTANT,
  DECL_TYPES: INT,
  NODE_TYPE: DATA_ACCESS,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: READ,
     OP_NAME: "1"
     ]>
  ]
)

(DF_NET#9
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#35
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#34# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#37
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#36# >
  )>,
DF_NET_INFO: [
  BIT_WIDTH: 1,
  NET_TYPE: DATA_NET,
  NUM_REPRES: BINARY,
]
)

(DF_NODE#10
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#6#,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#32
  IO_CLASS: INPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#33# >
  )>,
NUM_OUTPUTS: 0,
DF_NODE_INFO: [
  SIG_NAME: "DF_BLK_END",
  SIG_TYPE: SIGNAL,
  DECL_TYPES: INT,
  NODE_TYPE: MARKER,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: DF_BLK_END,
     OP_NAME: "DF_BLK_END"
     ]>
  ]
)
```

```
]
)

(DF_NODE#6
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#4#,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#20
  IO_CLASS: INPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#21# >
  )>,
NUM_OUTPUTS: 0,
DF_NODE_INFO: [
  SIG_NAME: "DF_BLK_END",
  SIG_TYPE: SIGNAL,
  DECL_TYPES: INT,
  NODE_TYPE: MARKER,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: DF_BLK_END,
     OP_NAME: "DF_BLK_END"
     ]>
  ]
)

(DF_NODE#4
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#4#,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#11
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#12# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#18
  IO_CLASS: OUTPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#19# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "T1",
  SIG_TYPE: REGISTER,
  DECL_TYPES: INT,
  NODE_TYPE: DATA_ACCESS,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: WRITE,
     OP_NAME: "T1"
     ]>
  ]
)

(DF_NET#5
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#19
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#18# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#21
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#20# >
  )>,
DF_NET_INFO: [
  BIT_WIDTH: 1,
  NET_TYPE: DATA_DEP_NET,
```

```
  NUM_REPRES: BINARY,
]
)

(DF_NODE#3
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#4#,
NUM_INPUTS: 2,
INPUT1: <
  (DF_PORT#5
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#6# >
  ),
  (DF_PORT#7
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#8# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#9
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#10# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "=",
  SIG_TYPE: SIGNAL,
  DECL_TYPES: INT,
  NODE_TYPE: OPERATION,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: RELATIONAL,
     OP_TYPE: EQ,
     OP_NAME: "="
     ]>
  ]
)

(DF_NET#3
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#10
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#9# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#12
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#11# >
  )>,
DF_NET_INFO: [
  BIT_WIDTH: 1,
  NET_TYPE: DATA_NET,
  NUM_REPRES: BINARY,
]
)

(DF_NODE#2
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#4#,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#17
  IO_CLASS: INPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#16# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#3
```

```
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#4# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "100",
  SIG_TYPE: CONSTANT,
  DECL_TYPES: BIT_VECTOR,
  NODE_TYPE: DATA_ACCESS,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: READ,
     OP_NAME: "100"
     ]>
  ]
)

(DF_NET#2
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#4
  IO_CLASS: INPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#3# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#8
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#7# >
  )>,
DF_NET_INFO: [
  BIT_WIDTH: 3,
  NET_TYPE: DATA_NET,
  NUM_REPRES: BINARY,
  ]
)

(DF_NODE#1
NODE_CLASS: DF_OP,
CF_NODE_REF: ##CF_NODE#4#,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#15
  IO_CLASS: INPUT,
  PORT_TYPE: DEP_PORT,
  BIT_WIDTH: 1,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#16# >
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (DF_PORT#1
  IO_CLASS: OUTPUT,
  PORT_TYPE: DATA_PORT,
  BIT_WIDTH: 3,
  NUM_CONN: 1,
  CONN1: < ##DF_PORT#2# >
  )>,
DF_NODE_INFO: [
  SIG_NAME: "SP",
  SIG_TYPE: VARIABLE,
  DECL_TYPES: INT,
  NODE_TYPE: DATA_ACCESS,
  NUM_OPS: 0,
  DF_OP_INFO: <
     [OP_CLASS: MISC_OP,
     OP_TYPE: READ,
     OP_NAME: "SP"
     ]>
  ]
)

(DF_NET#1
NODE_CLASS: DF_ARC,
NUM_INPUTS: 1,
INPUT1: <
  (DF_PORT#2
  IO_CLASS: INPUT,
```

```
    PORT_TYPE: DATA_PORT,
    BIT_WIDTH: 3,
    NUM_CONN: 1,
    CONN1: < ##DF_PORT#1# >
    )>,
  NUM_OUTPUTS: 1,
  OUTPUT1: <
    (DF_PORT#6
    IO_CLASS: OUTPUT,
    PORT_TYPE: DATA_PORT,
    BIT_WIDTH: 3,
    NUM_CONN: 1,
    CONN1: < ##DF_PORT#5# >
    )>,
  DF_NET_INFO: [
    BIT_WIDTH: 3,
    NET_TYPE: DATA_NET,
    NUM_REPRES: BINARY,
  ]
  )

  (DF_NODE#5
  NODE_CLASS: DF_OP,
  CF_NODE_REF: ##CF_NODE#4#,
  NUM_INPUTS: 0,
  NUM_OUTPUTS: 1,
  OUTPUT1: <
    (DF_PORT#13
    IO_CLASS: OUTPUT,
    PORT_TYPE: DEP_PORT,
    BIT_WIDTH: 1,
    NUM_CONN: 1,
    CONN1: < ##DF_PORT#14# >
    )>,
  DF_NODE_INFO: [
    SIG_NAME: "DF_BLK_START",
    SIG_TYPE: SIGNAL,
    DECL_TYPES: INT,
    NODE_TYPE: MARKER,
    NUM_OPS: 0,
    DF_OP_INFO: <
       [OP_CLASS: MISC_OP,
       OP_TYPE: DF_BLK_START,
       OP_NAME: "DF_BLK_START"
       ]>
    ]
  )

  (DF_NET#4
  NODE_CLASS: DF_ARC,
  NUM_INPUTS: 1,
  INPUT1: <
    (DF_PORT#14
    IO_CLASS: INPUT,
    PORT_TYPE: DATA_PORT,
    BIT_WIDTH: 1,
    NUM_CONN: 1,
    CONN1: < ##DF_PORT#13# >
    )>,
  NUM_OUTPUTS: 1,
  OUTPUT1: <
    (DF_PORT#16
    IO_CLASS: OUTPUT,
    PORT_TYPE: DATA_PORT,
    BIT_WIDTH: 1,
    NUM_CONN: 2,
    CONN1: < ##DF_PORT#15#, ##DF_PORT#17# >
    )>,
  DF_NET_INFO: [
    BIT_WIDTH: 1,
    NET_TYPE: DATA_DEP_NET,
    NUM_REPRES: BINARY,
  ]
  )
)
```

C.3 The Control Flow View

The control flow graph (without any data flow nodes) is given below.

```
(
[
  DD_DOMAIN_TYPE: BEHAVIOR,
```

```
  DD_FLAVOR: BEHAVIOR_PURE,
  DD_CHUNK_TYPE: CONTROL_FLOW
]

(CF_NODE#1
CF_NODE_TYPE: CF_START,
NUM_INPUTS: 0,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (CF_CONNS#1
  IO_CLASS: OUTPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#2#
  )>,
NUM_CF_COND_ITEMS: 0,
)

(CF_NODE#2
CF_NODE_TYPE: CF_END,
NUM_INPUTS: 1,
INPUT1: <
  (CF_CONNS#17
  IO_CLASS: INPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#18#
  )>,
NUM_OUTPUTS: 0,
NUM_CF_COND_ITEMS: 0,
)

(CF_NODE#3
CF_NODE_TYPE: PROC_START,
NUM_INPUTS: 1,
INPUT1: <
  (CF_CONNS#2
  IO_CLASS: INPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#1#
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (CF_CONNS#3
  IO_CLASS: OUTPUT,
  PROCESSED: 0,
```

```
  CF_CONNS_REF: ##CF_CONNS#4#
  )>,
NUM_CF_COND_ITEMS: 0,
)

(CF_NODE#4
CF_NODE_TYPE: STMT_BLK,
NUM_INPUTS: 1,
INPUT1: <
  (CF_CONNS#4
  IO_CLASS: INPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#3#
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (CF_CONNS#5
  IO_CLASS: OUTPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#6#
  )>,
NUM_CF_COND_ITEMS: 0,
DF_NODE_GROUP: <
  [ DF_STMNTS:
     <[ SYMBOL: "DF_BLK_START",
      DF_NODE_REF: ##DF_NODE#5# ]
     >
  ],
  [ DF_STMNTS:
     <[ SYMBOL: "SP",
        DF_NODE_REF: ##DF_NODE#1# ],
     [ SYMBOL: "100",
       DF_NODE_REF: ##DF_NODE#2# ],
     [ SYMBOL: "=",
       DF_NODE_REF: ##DF_NODE#3# ],
     [ SYMBOL: "T1",
       DF_NODE_REF: ##DF_NODE#4# ],
     [ SYMBOL: "DF_BLK_END",
       DF_NODE_REF: ##DF_NODE#6# ]
     >
  ]>,
)

(CF_NODE#5
```

```
CF_NODE_TYPE: IF_TEST,
NUM_INPUTS: 1,
INPUT1: <
  (CF_CONNS#6
  IO_CLASS: INPUT,
  PROCESSED: 0,
  GUARD_VAL: "T1",
  CF_CONNS_REF: ##CF_CONNS#5#
  )>,
NUM_OUTPUTS: 2,
OUTPUT1: <
  (CF_CONNS#7
  IO_CLASS: OUTPUT,
  PROCESSED: 0,
  GUARD_VAL: "1",
  CF_CONNS_REF: ##CF_CONNS#8#
  ),
  (CF_CONNS#11
  IO_CLASS: OUTPUT,
  PROCESSED: 0,
  GUARD_VAL: "0",
  CF_CONNS_REF: ##CF_CONNS#12#
  )>,
NUM_CF_COND_ITEMS: 0,
)

(CF_NODE#6
CF_NODE_TYPE: STMT_BLK,
NUM_INPUTS: 1,
INPUT1: <
  (CF_CONNS#8
  IO_CLASS: INPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#7#
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (CF_CONNS#9
  IO_CLASS: OUTPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#10#
  )>,
NUM_CF_COND_ITEMS: 0,
DF_NODE_GROUP: <
  [ DF_STMNTS:
     <[ SYMBOL: "DF_BLK_START",
        DF_NODE_REF: ##DF_NODE#8# ]
     >
  ],
  [ DF_STMNTS:
     <[ SYMBOL: "0",
        DF_NODE_REF: ##DF_NODE#7# ],
     [ SYMBOL: "FULL_SIG",
       DF_NODE_REF: ##DF_NODE#9# ]
     >
  ],
  [ DF_STMNTS:
     <[ SYMBOL: "DF_BLK_END",
        DF_NODE_REF: ##DF_NODE#10# ]
     >
  ]>,
)

(CF_NODE#7
CF_NODE_TYPE: IF_JOIN,
NUM_INPUTS: 2,
INPUT1: <
  (CF_CONNS#10
  IO_CLASS: INPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#9#
  ),
  (CF_CONNS#13
  IO_CLASS: INPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#14#
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (CF_CONNS#15
  IO_CLASS: OUTPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#16#
  )>,
NUM_CF_COND_ITEMS: 0,
)

(CF_NODE#8
```

```
CF_NODE_TYPE: STMT_BLK,
NUM_INPUTS: 1,
INPUT1: <
  (CF_CONNS#12
  IO_CLASS: INPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#11#
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (CF_CONNS#14
  IO_CLASS: OUTPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#13#
  )>,
NUM_CF_COND_ITEMS: 0,
DF_NODE_GROUP: <
  [ DF_STMNTS:
     <[ SYMBOL: "DF_BLK_START",
        DF_NODE_REF: ##DF_NODE#12# ]
     >
  ],
  [ DF_STMNTS:
     <[ SYMBOL: "1",
        DF_NODE_REF: ##DF_NODE#11# ]
     [ SYMBOL: "FULL_SIG",
       DF_NODE_REF: ##DF_NODE#13# ]
     >
  ],
  [ DF_STMNTS:
     <[ SYMBOL: "SP",
        DF_NODE_REF: ##DF_NODE#14# ]
     [ SYMBOL: "1",
       DF_NODE_REF: ##DF_NODE#15# ]
     [ SYMBOL: "+",
       DF_NODE_REF: ##DF_NODE#16# ]
     [ SYMBOL: "SP",
       DF_NODE_REF: ##DF_NODE#17# ]
     >
  ],
  [ DF_STMNTS:
     <[ SYMBOL: "DF_BLK_END",
        DF_NODE_REF: ##DF_NODE#18# ]
     >
  ]>,
)

(CF_NODE#9
CF_NODE_TYPE: PROC_END,
NUM_INPUTS: 1,
INPUT1: <
  (CF_CONNS#16
  IO_CLASS: INPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#15#
  )>,
NUM_OUTPUTS: 1,
OUTPUT1: <
  (CF_CONNS#18
  IO_CLASS: OUTPUT,
  PROCESSED: 0,
  CF_CONNS_REF: ##CF_CONNS#17#
  )>,
NUM_CF_COND_ITEMS: 0,
)
)
```

C.4 The Complete Control/Data Flow View

Finally, the control/data flow graph would be constructed by combining the control flow graph presented in the previous section with the design objects of the three data flow graphs (one of which has been shown in an earlier section). Therefore, we do not repeat this information here.

D USER'S MANUAL FOR BDEF PARSER/GENERATOR TOOLS

USER'S MANUAL
PARSER/GENERATOR TOOLS FOR THE BEHAVIORAL DESIGN DATA EXCHANGE FORMAT (BDEF)

Elke A. Rundensteiner

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

email: rundenstics.uci.edu
telephone: (714) 856-4101
March, 1991

ABSTRACT

The Behavioral Design Data Exchange Format (BDEF) is a textual format for the design representation of the Extended Control/Data Flow Graph (ECDFG) Model. BDEF has been developed to serve as exchange format of design data between the Behavioral Design Data Base (BDDB) and the design tools in the behavioral synthesis environment at the University of California, Irvine. This user's manual describes how to use the Behavioral Design Data Exchange Format (BDEF) parser and generator tools. The BDEF parser compiles a BDEF description of a design into its corresponding ECDFG data structures. The BDEF translator on the other hand maps the ECDFG data structures of a design into its corresponding BDEF description.

1 Introduction

This user's manual describes the software support tools that have been developed at the University of California, Irvine for handling the Behavioral Design Data Exchange Format (BDEF). BDEF is a textual format for the design representation of the Extended Control/Data Flow Graph (ECDFG) Model [1]. BDEF is used as design data exchange format [2] in the behavioral synthesis environment at the University of California, Irvine. In particular, BDEF serves as exchange format of design data between the Behavioral Design Data Base (BDDB) and the design tools. This user's manual describes how to use the Behavioral Design Data Exchange Format (BDEF) parser and generator tools. The BDEF Parser compiles a BDEF description of a design into the ECDFG data structures. The BDEF Translator maps the ECDFG data structures of a design into its corresponding BDEF description. The BDEF Parser/Translator pair is implemented in the C programming language. It is currently running on SUN 3/Sun 4 workstations under the UNIX operation system.

2 PBC: The Parser from BDEF to ECDFG Data Structures

PBC is a parser/compiler that parses a BDEF design description [2] and generates in-memory data structures for the design using the Extended Control/Data Flow Graph data structures [1]. This tool is provided by the Behavioral Design Data Base (BDDB) to the design tools so that they can move a ECDFG from a textual format to in-memory data structures.

2.1 Where is What?

The source code for the BDEF Parser (PBC) can be found in:

/cz/ua/elke/BDEF/dbcode/parser

The source code for utility functions and variable and type definitions used by both PBC and TBC is in:

/cz/ua/elke/BDEF/dbcode/utils

The public PBC object code that has to be linked with your program in order to run the parser is:

/cz/ua/elke/BDEF/pub_objs/sun3/pbc_sun3.o

/cz/ua/elke/BDEF/pub_objs/sun3/utils_module_sun3.o

The matching sun4 executables are:

/cz/ua/elke/BDEF/pub_objs/sun4/pbc_sun4.o

/cz/ua/elke/BDEF/pub_objs/sun4/utils_module_sun4.o

An example Makefile and an example program that invokes the parser can be found in:

/cz/ua/elke/BDEF/vss_test

2.2 Input/Output Data

PBC has one input parameter, namely, a file that contains a textual BDEF design description. For a discussion of the BDEF format see [2]. BDEF input files have the following naming convention:

< design-name >.bdef

The PBC parser returns in-memory data structures using the Extended Control/Data Flow Graph Model [1]. This is done via initializing global variables defined in the "utils" directory. The definition of these global variables that will hold the resulting data structures is in the file /cz/ua/elke/BDEF/dbcode/utils/UTILS_BDDB_variables.h. It is listed next.

```
/*********************************************************/
/* Global variables to hold the ECDFG when created by PBC  */
/*********************************************************/

/* state graph  */

struct graph *    BDDB_state_graph;

/* data flow  */

struct df_node_net_list *    BDDB_df_nodes;

/* control flow  */

struct cf_node_list *    BDDB_cf_nodes;

/* list of timing constraints  */

struct timing_spec *         BDDB_timing;
```

When PBC creates a state graph, then the variable BDDB_state_graph will point to the resulting data structures. When PBC creates a control flow graph, then the variable BDDB_cf_nodes will point to the generated control flow data structures. When PBC creates only a data flow graph, then the

variable BDDB_df_nodes will point to the generated data flow data structures. If there is timing constraint information that is generated by PBC, then the global variable BDDB_timing will point to a list of all timing specification structures.

2.3 How to Execute PBC

To execute PBC as a stand-alone system, enter the following command:

% pbc_sun3 < design-name >

or

% pbc_sun4 < design-name >

You can also include PBC in form of a function call into your favorite synthesis system. PBC then will generated an internal data flow graph representation from a given BDEF description that can be directly used by your program. In this case, the executable file "pbc.o" and "utils_module.o" have to be loaded with your C code. The file "pbc.h" contains the function definition "pbc()" and therefore must be included into the file in which you want to call PBC.

The function pbc() has the following declaration:

```
void pbc(pbc_file,num_argc)
char *pbc_file;
int num_argc;
{
...
}
```

The first argument of pbc() corresponds to the name of the design file that is being parsed. The second argument is set to 1 when the parser reads from standard input (stdin) and to 2 when the parser reads its input from a text file.

Execute PBC with the following function call:

pbc(design_file_name,2);

PBC uses global variables to determine the desired design view of the design data. I provide for a function that allows the user to set the design data characteristics that he/she is interested in, that is, that are to be extracted from the BDEF description. If the requested design data is available in the design file then the information is extracted and put into data structures. If the requested design information is available in the BDEF description then the user is notified. The design data characteristics defined in the "utils" model are given in a later section.

2.4 An Example PBC Execution

```
-----------------------------------------------
-------    BDEF Support Tools (BDDB)    -------
-------       March 1991 Prototype      -------
-----------------------------------------------


 Select between tools:
    0 = Quit
    1 = Graph Compiler
    2 = TCB
    3 = PBC
    4 = Print in-memory ECDFG data structures
    5 = Print values of DD characteristics
 =>3

Invoke PBC (y/n)? y

Specify input file name for design data to be parsed:simple_cfg

Using <simple_cfg> as input_file.

==============================================
(PBC) Parser from BDEF to ECDFG running  ...
==============================================

Parsing of file <simple_cfg.bdef> ...

Should all Design Data in the design entity be compiled (y/n)? y


Symbol table 5: No timing constraints specified!

Symbol List 1
cf_conns: 3
cf_conns: 2
cf_conns: 4
cf_conns: 1
Symbol table 1
Cf_conns_source: 3,cf_node: 3,cf_conns_dest: 4
Cf_conns_source: 1,cf_node: 1,cf_conns_dest: 2

Successful Parsing of file <simple_cfg_2.bdef>.

==============================================
PBC Parser Execution completed.
==============================================

 Select between tools:
    0 = Quit
    1 = Graph Compiler
```

```
    2 = TCB
    3 = PBC
    4 = Print in-memory ECDFG data structures
    5 = Print values of DD characteristics
 =>0

Quitting!
<sun>
```

3 TCB: The Translator from the ECDFG Data Structures to BDEF

TCB is a BDEF Generator that translates a ECDFG design representation into its corresponding BDEF format. This BDEF generator is a valuable tool for design data exchange since it allows the Behavioral Design Data Base to capture design data used by different design tools in a unified format.

3.1 Where is What?

The source code for the BDEF Generator (TCB) can be found in:

/cz/ua/elke/BDEF/dbcode/translator

The source code for utility functions and global definitions used by both TCB and PBC is in:

/cz/ua/elke/BDEF/dbcode/utils

The public TCB object code that has to be linked with your program in order to run the generator is:

/cz/ua/elke/BDEF/pub_objs/sun3/tcb_sun3.o

/cz/ua/elke/BDEF/pub_objs/sun3/utils_module_sun3.o

The matching sun4 executables are:

/cz/ua/elke/BDEF/pub_objs/sun4/tcb_sun4.o

/cz/ua/elke/BDEF/pub_objs/sun4/utils_module_sun4.o

An example Makefile and an example program that invokes the generator can be found in:

/cz/ua/elke/BDEF/vss_test

3.2 Input/Output Data

TCB has several input parameters:

```
void tcb (designfile,state_graph,cfg,dfg)
FILE * designfile;
struct  graph * state_graph;
struct  cf_node_list * cfg;
struct  df_node_net_list * dfg;
{
...
}
```

The first argument corresponds to the name of the design file that will hold the generated BDEF design description. The other three parameters correspond to pointers to the in-memory data structures that are to be translated to BDEF. The second parameter state_graph holds a pointer to the state graph, the third parameter cfg holds a pointer to the control flow graph, and the fourth parameter dfg holds a pointer to the data flow graph to be printed out. Only one of these three parameters is not equal NULL at a given time. When the internal data structure is a state graph, then the second parameter will point to this graph while the other parameters will be NULL. If the user is interested in only generating the BDEF description of a control flow graph or of a data flow graph, then s/he nees to enter a pointer to the respective graph as parameter to the tcb function.

The TCB generator returns a BDEF description of the design represented by the given in-memory data structures. Due to the following naming convention for BDEF files, TCB will append a ".bdef" to the first parameter, the input file name. The generated BDEF design description will then be stored in this file.

3.3 How to Execute TCB

To execute TCB as a stand-alone system, enter the following command:

% tcb_sun3 < design-name >

or

% tcb_sun4 < design-name >

You can also include TCB in form of a function call into your synthesis system. TCB then will generated a BDEF description that captures the internal data flow graph representation produced by your program. This allows for the sharing of the design data with other design tools. In this case, the executable file "tcb.o" and "utils_module.o" have to be loaded with your C code. The file "tcb.h" contains the function definition "tcb()" and therefore must be included into the file in which you want to call TCB.

Execute TCB with the following function call:

tcb(design_file_name,stg,cfg,dfg);

with either stg, cfg or dfg a pointer to the current ECDFG data structures.

The BDEF Generator has the three parameters, design entity domain type, design entity flavor, and design entity chunk type. They determine the type of design data that is to be captured in the BDEF description. If the requested information is not available in the design representation, then the user of this BDEF Generator will be notified. The design data characteristics defined in the "utils" model are given in a later section.

3.4 An Example TCB Execution

```
-----------------------------------------------------------
-------      BDEF Support Tools (BDDB)          -------
-------         March 1991 Prototype            -------
-----------------------------------------------------------


 Select between tools:
    0 = Quit
    1 = Graph Compiler
    2 = TCB
    3 = PBC
    4 = Print in-memory ECDFG data structures
    5 = Print values of DD characteristics
 =>2

Invoke TCB (y/n)? y


Set the characteristics to be dealt with.

The current BDDB Design Data Characteristics are:
===========================================================


 Design Data Domain Type = BEHAVIOR
 Design Data Behavioral Flavor = BEHAVIOR_PURE
 Design Data  Chunk Type = CONTROL_DATA_FLOW
```

```
 Use the current values (y/n)?n



 Setting of BDDB Design Data Characteristics...
 ==============================================

  Use default parameters (y/n)? n


  Set the Design Data Domain Type:
     0 = BEHAVIOR (default)
     1 = STRUCTURE
  =>0


  Design Data Domain Type = BEHAVIOR

  Set the Design Data Behavioral Flavor:
     0 = BEHAVIOR_PURE (default)
     1 = BEHAVIOR_WITH_STATES
     2 = BEHAVIOR_WITH_ALLOCATION
     3 = BEHAVIOR_WITH_BINDING
     4 = BEHAVIOR_WITH_CONTROL
  =>0


  Design Data Behavioral Flavor = BEHAVIOR_PURE

  Set the Design Data Chunk Type:
     0 = DATA_FLOW
     1 = CONTROL_FLOW
     2 = CONTROL_DATA_FLOW (default)
     3 = STATE_GRAPH
     4 = STATE_CONTROL_FLOW
     5 = STATE_CONTROL_DATA_FLOW
  =>1


  Design Data  Chunk Type = CONTROL_FLOW

Specify file name for design data to be stored in: simple_cfg


==============================================
(TCB) Translator from ECDFG to BDDB running  ...
==============================================


BDDB Design Description Result stored in the file simple_cfg.bdef


==============================================
```

```
TCB Translator Execution completed.
==============================================

 Select between tools:
    0 = Quit
    1 = Graph Compiler
    2 = TCB
    3 = PBC
    4 = Print in-memory ECDFG data structures
    5 = Print values of DD characteristics
 =>
```

4 Putting it all together

An example of how PBC and TCB can be used together to read data from file into main memory and to put the data back onto the file has been put together. This example uses VSS (version 4), i.e., PBC and TCB are used in between the design tool invocations to keep track of the design as it evolves. The design tools used in this example are the VHDL graph compiler, the allocator, scheduler, and resource binder. The source code and Makefile for this example application can be found in:

/cz/ua/elke/BDEF/vss_test

The resulting executables that can be used for testing purposes are:

/cz/ua/elke/BDEF/vss_test/bdef_sun3

/cz/ua/elke/BDEF/vss_test/bdef_sun4

For this example, I have developed the function invoke_tools() which provides a menu for choosing the invocation of the following tools: VHDL graph compiler, pbc, tcb, and debug print routines. Include this function invoke_tools() into your main file in places where you would want to save the current in-memory data structures onto files as well as in places where you would like to bring data from a BDEF file into in-memory data structures.

5 BDEF Global Type Definitions/Variables

The BDEF tools use the following global type definitions and variables defined in the files UTILS_BDDB_variables. and UTILS_BDDB_typedefs.h.

```
#define _UTILS_BDDB_typedefs
/*****************************************************************/
```

References

[1] Rundensteiner, E. A., & Gajski, D. D., "A Design Representation for High-Level Synthesis", Info. & Computer Science Dept., UCI, Tech. Rep. 90-27, Sep. 1990.

[2] Rundensteiner, E. A., & Gajski, D. D., "BDEF: The Behavioral Design Data Exchange Format", Info. & Computer Science Dept., UCI, Tech. Rep. 90-34, 1991.

USER'S MANUAL PARSER/GENERATOR TOOLS FOR THE BEHAVIORAL DESIGN DATA EXCHANGE FORMAT (BDEF)

Elke A. Rundensteiner

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

email: rundenstics.uci.edu
telephone: (714) 856-4101
March, 1991

ABSTRACT

The Behavioral Design Data Exchange Format (BDEF) is a textual format for the design representation of the Extended Control/Data Flow Graph (ECDFG) Model. BDEF has been developed to serve as exchange format of design data between the Behavioral Design Data Base (BDDB) and the design tools in the behavioral synthesis environment at the University of California, Irvine. This user's manual describes how to use the Behavioral Design Data Exchange Format (BDEF) parser and generator tools. The BDEF parser compiles a BDEF description of a design into its corresponding ECDFG data structures. The BDEF translator on the other hand maps the ECDFG data structures of a design into its corresponding BDEF description.

variable BDDB_df_nodes will point to the generated data flow data structures. If there is timing constraint information that is generated by PBC, then the global variable BDDB_timing will point to a list of all timing specification structures.

2.3 How to Execute PBC

To execute PBC as a stand-alone system, enter the following command:

% pbc_sun3 < design-name >

or

% pbc_sun4 < design-name >

You can also include PBC in form of a function call into your favorite synthesis system. PBC then will generated an internal data flow graph representation from a given BDEF description that can be directly used by your program. In this case, the executable file "pbc.o" and "utils_module.o" have to be loaded with your C code. The file "pbc.h" contains the function definition "pbc()" and therefore must be included into the file in which you want to call PBC.

The function pbc() has the following declaration:

```
void pbc(pbc_file,num_argc)
char *pbc_file;
int num_argc;
{
...
}
```

The first argument of pbc() corresponds to the name of the design file that is being parsed. The second argument is set to 1 when the parser reads from standard input (stdin) and to 2 when the parser reads its input from a text file.

Execute PBC with the following function call:

pbc(design_file_name.2);

PBC uses global variables to determine the desired design view of the design data. I provide for a function that allows the user to set the design data characteristics that he/she is interested in, that is, that are to be extracted from the BDEF description. If the requested design data is available in the design file then the information is extracted and put into data structures. If the requested design information is available in the BDEF description then the user is notified. The design data characteristics defined in the "utils" model are given in a later section.

2.4 An Example PBC Execution

```
-----------------------------------------------
-------   BDEF Support Tools (BDDB)     -------
-------      March 1991 Prototype       -------
-----------------------------------------------


 Select between tools:
    0 = Quit
    1 = Graph Compiler
    2 = TCB
    3 = PBC
    4 = Print in-memory ECDFG data structures
    5 = Print values of DD characteristics
 =>3

Invoke PBC (y/n)? y

Specify input file name for design data to be parsed:simple_cfg

Using <simple_cfg> as input_file.

==============================================
(PBC) Parser from BDEF to ECDFG running  ...
==============================================

Parsing of file <simple_cfg.bdef> ...

Should all Design Data in the design entity be compiled (y/n)? y


Symbol table 5: No timing constraints specified!

Symbol List 1
cf_conns: 3
cf_conns: 2
cf_conns: 4
cf_conns: 1
Symbol table 1
Cf_conns_source: 3,cf_node: 3,cf_conns_dest: 4
Cf_conns_source: 1,cf_node: 1,cf_conns_dest: 2

Successful Parsing of file <simple_cfg_2.bdef>.

==============================================
PBC Parser Execution completed.
==============================================

 Select between tools:
    0 = Quit
    1 = Graph Compiler
```

```
    2 = TCB
    3 = PBC
    4 = Print in-memory ECDFG data structures
    5 = Print values of DD characteristics
 =>0

Quitting!
<sun>
```

3 TCB: The Translator from the ECDFG Data Structures to BDEF

TCB is a BDEF Generator that translates a ECDFG design representation into its corresponding BDEF format. This BDEF generator is a valuable tool for design data exchange since it allows the Behavioral Design Data Base to capture design data used by different design tools in a unified format.

3.1 Where is What?

The source code for the BDEF Generator (TCB) can be found in:

/cz/ua/elke/BDEF/dbcode/translator

The source code for utility functions and global definitions used by both TCB and PBC is in:

/cz/ua/elke/BDEF/dbcode/utils

The public TCB object code that has to be linked with your program in order to run the generator is:

/cz/ua/elke/BDEF/pub_objs/sun3/tcb_sun3.o

/cz/ua/elke/BDEF/pub_objs/sun3/utils_module_sun3.o

The matching sun4 executables are:

/cz/ua/elke/BDEF/pub_objs/sun4/tcb_sun4.o

/cz/ua/elke/BDEF/pub_objs/sun4/utils_module_sun4.o

An example Makefile and an example program that invokes the generator can be found in:

/cz/ua/elke/BDEF/vss_test

3.2 Input/Output Data

TCB has several input parameters:

```
void tcb (designfile,state_graph,cfg,dfg)
FILE * designfile;
struct  graph * state_graph;
struct  cf_node_list * cfg;
struct  df_node_net_list * dfg;
{
...
}
```

The first argument corresponds to the name of the design file that will hold the generated BDEF design description. The other three parameters correspond to pointers to the in-memory data structures that are to be translated to BDEF. The second parameter state_graph holds a pointer to the state graph, the third parameter cfg holds a pointer to the control flow graph, and the fourth parameter dfg holds a pointer to the data flow graph to be printed out. Only one of these three parameters is not equal NULL at a given time. When the internal data structure is a state graph, then the second parameter will point to this graph while the other parameters will be NULL. If the user is interested in only generating the BDEF description of a control flow graph or of a data flow graph, then s/he nees to enter a pointer to the respective graph as parameter to the tcb function.

The TCB generator returns a BDEF description of the design represented by the given in-memory data structures. Due to the following naming convention for BDEF files, TCB will append a ".bdef" to the first parameter, the input file name. The generated BDEF design description will then be stored in this file.

3.3 How to Execute TCB

To execute TCB as a stand-alone system, enter the following command:

```
% tcb_sun3 < design-name >
```

or

```
% tcb_sun4 < design-name >
```

You can also include TCB in form of a function call into your synthesis system. TCB then will generated a BDEF description that captures the internal data flow graph representation produced by your program. This allows for the sharing of the design data with other design tools. In this case, the executable file "tcb.o" and "utils_module.o" have to be loaded with your C code. The file "tcb.h" contains the function definition "tcb()" and therefore must be included into the file in which you want to call TCB.

Execute TCB with the following function call:

tcb(design_file_name,stg,cfg,dfg);

with either stg, cfg or dfg a pointer to the current ECDFG data structures.

The BDEF Generator has the three parameters, design entity domain type, design entity flavor, and design entity chunk type. They determine the type of design data that is to be captured in the BDEF description. If the requested information is not available in the design representation, then the user of this BDEF Generator will be notified. The design data characteristics defined in the "utils" model are given in a later section.

3.4 An Example TCB Execution

```
-----------------------------------------------
-------     BDEF Support Tools (BDDB)   -------
-------       March 1991 Prototype      -------
-----------------------------------------------


 Select between tools:
    0 = Quit
    1 = Graph Compiler
    2 = TCB
    3 = PBC
    4 = Print in-memory ECDFG data structures
    5 = Print values of DD characteristics
 =>2

Invoke TCB (y/n)? y


Set the characteristics to be dealt with.

The current BDDB Design Data Characteristics are:
=================================================


 Design Data Domain Type = BEHAVIOR
 Design Data Behavioral Flavor = BEHAVIOR_PURE
 Design Data  Chunk Type = CONTROL_DATA_FLOW
```

```
Use the current values (y/n)?n


Setting of BDDB Design Data Characteristics...
==============================================

Use default parameters (y/n)? n


Set the Design Data Domain Type:
   0 = BEHAVIOR (default)
   1 = STRUCTURE
=>0


Design Data Domain Type = BEHAVIOR

Set the Design Data Behavioral Flavor:
   0 = BEHAVIOR_PURE (default)
   1 = BEHAVIOR_WITH_STATES
   2 = BEHAVIOR_WITH_ALLOCATION
   3 = BEHAVIOR_WITH_BINDING
   4 = BEHAVIOR_WITH_CONTROL
=>0


Design Data Behavioral Flavor = BEHAVIOR_PURE

Set the Design Data Chunk Type:
   0 = DATA_FLOW
   1 = CONTROL_FLOW
   2 = CONTROL_DATA_FLOW (default)
   3 = STATE_GRAPH
   4 = STATE_CONTROL_FLOW
   5 = STATE_CONTROL_DATA_FLOW
=>1


Design Data  Chunk Type = CONTROL_FLOW

Specify file name for design data to be stored in: simple_cfg


==============================================
(TCB) Translator from ECDFG to BDDB running  ...
==============================================


BDDB Design Description Result stored in the file simple_cfg.bdef


==============================================
```

```
TCB Translator Execution completed.
==============================================

 Select between tools:
    0 = Quit
    1 = Graph Compiler
    2 = TCB
    3 = PBC
    4 = Print in-memory ECDFG data structures
    5 = Print values of DD characteristics
=>
```

4 Putting it all together

An example of how PBC and TCB can be used together to read data from file into main memory and to put the data back onto the file has been put together. This example uses VSS (version 4), i.e., PBC and TCB are used in between the design tool invocations to keep track of the design as it evolves. The design tools used in this example are the VHDL graph compiler, the allocator, scheduler, and resource binder. The source code and Makefile for this example application can be found in:

/cz/ua/elke/BDEF/vss_test

The resulting executables that can be used for testing purposes are:

/cz/ua/elke/BDEF/vss_test/bdef_sun3

/cz/ua/elke/BDEF/vss_test/bdef_sun4

For this example, I have developed the function invoke_tools() which provides a menu for choosing the invocation of the following tools: VHDL graph compiler, pbc, tcb, and debug print routines. Include this function invoke_tools() into your main file in places where you would want to save the current in-memory data structures onto files as well as in places where you would like to bring data from a BDEF file into in-memory data structures.

5 BDEF Global Type Definitions/Variables

The BDEF tools use the following global type definitions and variables defined in the files UTILS_BDDB_variabl and UTILS_BDDB_typedefs.h.

```
#define _UTILS_BDDB_typedefs
/*****************************************************************/
```

```
/* UTILS_BDDB_typedefs.h                                               */
/*                         TYPE DEFINITIONS FOR THE                    */
/*                   TEXTUAL ECDFG FILEFORMAT PARSER MODULE             */
/*                      Symbol Table data structures                   */
/*                                                                     */
/* Behavioral Design Data Base (BDDB)                                  */
/* Copyright (c) 1990                  by Elke A. Rundensteiner        */
/* Last updated: Jan/91                                                */
/*********************************************************************/



/*********************************************************************/
/* design data domain type                                           */
/*********************************************************************/

typedef enum { BEHAVIOR, STRUCTURE }  BDDB_DD_Domain_Type;

/*********************************************************************/
/* design data behavioral flavor if design data type is BEHAVIOR*/
/*********************************************************************/

typedef enum
                 {BEHAVIOR_PURE,
                  BEHAVIOR_WITH_STATES,
                  BEHAVIOR_WITH_ALLOCATION,
                  BEHAVIOR_WITH_BINDING,
                  BEHAVIOR_WITH_CONTROL
                }
           BDDB_DD_Behavioral_Flavor;

/*********************************************************************/
/* design data behavioral chunk type                                 */
/*********************************************************************/

typedef enum
                  { DATA_FLOW,
                    CONTROL_FLOW,
                    CONTROL_DATA_FLOW,
                    STATE_GRAPH,
                    STATE_CONTROL_FLOW,
                    STATE_CONTROL_DATA_FLOW
                   }
                  BDDB_DD_Chunk_Type;

/*********************************************************************/
/* design data behavioral flavor if design data type is BEHAVIOR*/
/*********************************************************************/

typedef enum
                { STRUCTURE_PURE,
                  STRUCTURE_WITH_ESTIMATES,
                  STRUCTURE_WITH_GEOMETRY
                }
```

```
              BDDB_DD_Structural_Flavor;

#endif


#ifndef _UTILS_BDDB_variables
#define _UTILS_BDDB_variables

/*****************************************************************/
/* UTILS_BDDB_variables.h                                        */
/*                      GLOBAL VARIABLES  FOR THE                */
/*                  TEXTUAL ECDFG FILEFORMAT PARSER MODULE        */
/*                    Symbol Table data structures               */
/*                                                               */
/* Behavioral Design Data Base (BDDB)                            */
/* Copyright (c) 1990             by Elke A. Rundensteiner       */
/* Last updated: Jan/91                                          */
/*****************************************************************/


/*****************************************************************/
/* Variables describing the design data characteristics          */
/*****************************************************************/

/* design data domain type          */

 BDDB_DD_Domain_Type   DD_Domain_Type;

/* design data behavioral flavor if design data type is BEHAVIOR    */

  BDDB_DD_Behavioral_Flavor DD_Behavioral_Flavor;


/* design data behavioral chunk type    */

 BDDB_DD_Chunk_Type DD_Chunk_Type;


/* design data behavioral flavor if design data type is BEHAVIOR     */

 BDDB_DD_Structural_Flavor DD_Structural_Flavor;

/* determine whether timing constraints are used  */
/* if design data type is BEHAVIOR     */

 bool DD_Timing;

/* design entity name */

 char * DE_Name;

/* design entity version number */

 int DE_Version_Num;
```

References

[1] Rundensteiner, E. A., & Gajski, D. D., "A Design Representation for High-Level Synthesis", Info. & Computer Science Dept., UCI, Tech. Rep. 90-27, Sep. 1990.

[2] Rundensteiner, E. A., & Gajski, D. D., "BDEF: The Behavioral Design Data Exchange Format", Info. & Computer Science Dept., UCI, Tech. Rep. 90-34, 1991.

3 1970 00882 5322