Translation of hardware description languages to structured representation: a tool for digital system analysis

Daniela C. Peixoto, Diógenes Silva Jr., José M. Mata, Claudionor N. Coelho Jr., Antônio O. Fernandes
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Av. Antônio Carlos 6627, Belo Horizonte, Minas Gerais - Brasil
{cascini,dgns,mata,coelho,otavio}@dcc.ufmg.br

Abstract—

SDS (System Data Structure), part of a digital synthesis system, is an internal representation that captures data/operation and communication/synchronization aspects from the behavioral specification of a digital system. SDS divides design information into two graphs that describe the data flow behavior and control timing behavior. This representation is useful for synthesis and validation/verification.

This paper presents the translation process from the hardware description language VHDL to SDS, basically the transformation of a behavioral specification to a structured representation.

Keywords— **system and high-level synthesis, behavioral design, control flow analysis, data flow analysis**

I. Introduction

The design of a digital system involves specification, validation/verification and synthesis. Usually the specification is done at the behavioral level in a hardware description language like VHDL. The synthesis process involves several steps, going from behavioral level through structural level up to hardware implementation.

Validation and verification are techniques used to determine whether a design is correct, consistent and complete. Validation corresponds to checking static aspects of the system, like data, interconnection, and communication ports; checking is based on a set of design rules. Verification is used to check dynamic aspects of a system, such as timing, communication patterns, and interconnection compatibility.

SDS (System Data Structure) [SIL97] is an internal representation that captures data/operation and communication/synchronization aspects from the behavioral specification of a digital system. This representation is useful for synthesis and validation/verification.

SDS is used to represent a design at the system level. SDS divides design information into two graphs which describe the data flow behavior and control timing behavior. This representation allows a direct analysis of the program structure, for example the level of parallelism among operations, and an analysis of resource allocation.

This paper presents the translation process from the hardware description language VHDL to SDS, basically the transformation of a behavioral specification to a structured representation. This transformation is done by data flow analysis, control flow analysis, and graph optimization. Scheduling, which is part of the synthesis process, is also realized.

The behavioral specification of a digital system can be described in a hardware description language like VHDL [IEE87, LIP89], Verilog [THO91], and HardwareC [KU90, DEM90]. In this work we use a subset of VHDL for behavioral system specification.

II. SDS Representation

SDS models digital circuits as a set of concurrent and communicating processes. It has a higher abstraction level than algorithms or high-level representations, such as DDS (Design Data Structure) [KNA85], SAW (System Architect's Workbench) [WAL87] and BIF (Behavioral Intermediate Format) [DUT89].

SDS is a multi-level design representation. This representation divides design information into two subspaces, without an implicit relation between the objects of each subspace. The two subspaces represent respectively the data flow behavior (DFG) and the control timing behavior (CTG). The entities of the two subspaces can be explicit related through bindings.

A. Data flow subspace

The data flow subspace is used to represent the behavior of data transformation. It is modeled using a data flow graph (DFG) that resembles the flow graphs used in compilers for data flow machines. This graph is a directed acyclic bipartite graph which is constituted of a set of

operations, a set of values, and a set of directed edges that connect operations and values.

Special operations are defined in DFG to represent conditional data dependencies and loops. They are: distribute and join operations, which are used to support conditional data dependencies, and iterate operators, which are used to support loop commands.

B. Control flow and timing subspace

The control timing flow graph consists of ranges and points. A range represents a limit or a timing duration, an order relationship, or a casual relationship among points. Points represent an infinitesimal events duration, which separate the ranges.

Special points are defined in CTG to represent conditional branches and loops. They are: *or-fork* and *or-join* points.

In conditional branches, each outgoing range of an *or-fork* point is associated with a condition. The flow of execution follows the range whose condition is satisfied.

Loops begin at a first point, which is followed by the *or-fork* point, and iterate until the end point, which is followed by the last point of a while loop which is the *or-join*, at which time the flow of execution returns to the first point. Each pair of loop begin and end points are uniquely identified.

C. Bindings

The explicit relation that describes the connection between the elements of the two subspaces are called bindings. The binding is a 2-tuple (pair of values) that assigns data flow operations to control timing ranges.

Figure 1 illustrates the behavioral specification of a module in VHDL and figure 2 shows the corresponding graphs in SDS and one binding example.

```
architecture behavior of COMPARE is
begin process
  begin
     if (A=B) then
       N <= 0 ;
     else
       N <= N + 1;
     end if;
     C <= N;
  end process;
end behavior;
```

Fig.1 A sample VHDL description

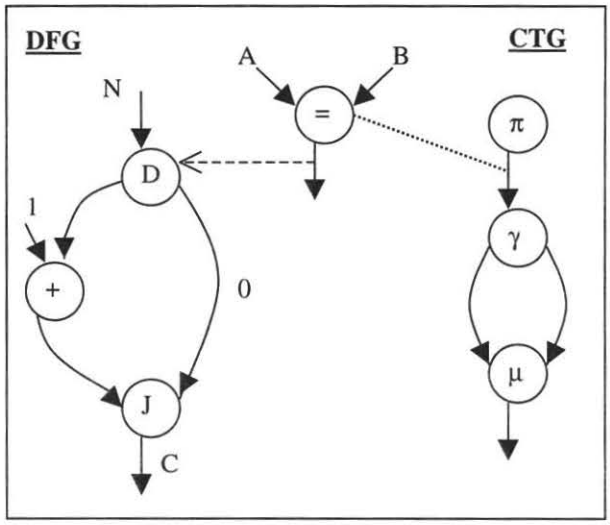

Fig.2 The DFG and CTG

II. VHDL TO SDS TRANSFORMATION

The task of VHDL to SDS translation can be described as following:

1. Extract the control information from a VHDL description and construct a flow graph for later data flow analysis. A flow graph is a directed graph of basic blocks. A basic block represents a sequence of computations that are bound through edges that represent the flow of control.
2. Data flow analysis can be performed in two steps. First, we use a local data flow analysis procedure to collect intra-block data dependencies. Then, a global data flow analysis procedure is used to analyze the inter-block dependencies. After this phase, the annotated flow graph becomes a combination of a DFG and a CTG.
3. The annotated flow graph may contain many redundant operations of the form x = y which must be eliminated in order to produce an optimized DFG.
4. The CTG and DFG are produced separately.
5. After setting the separated flow graphs up, the scheduling becomes straightforward.

A. Parsing

All entries in VHDL must first be analyzed and transformed in an intermediate representation: a syntax graph. The parser VAUL[VAU94] is used for this.

VAUL (VHDL Analyzer and Utility Library) was developed at the University of Dortmund, with the aim to provide a front-end to tools that uses VHDL. It analyzes each VHDL program's design unit separately, resulting in a

syntax graph, which is represented by a collection of C++ objects, connected by pointers.

The VHDL2SDS translator generates a representation using as entry the syntax graph. The translating process is shown in figure 3.

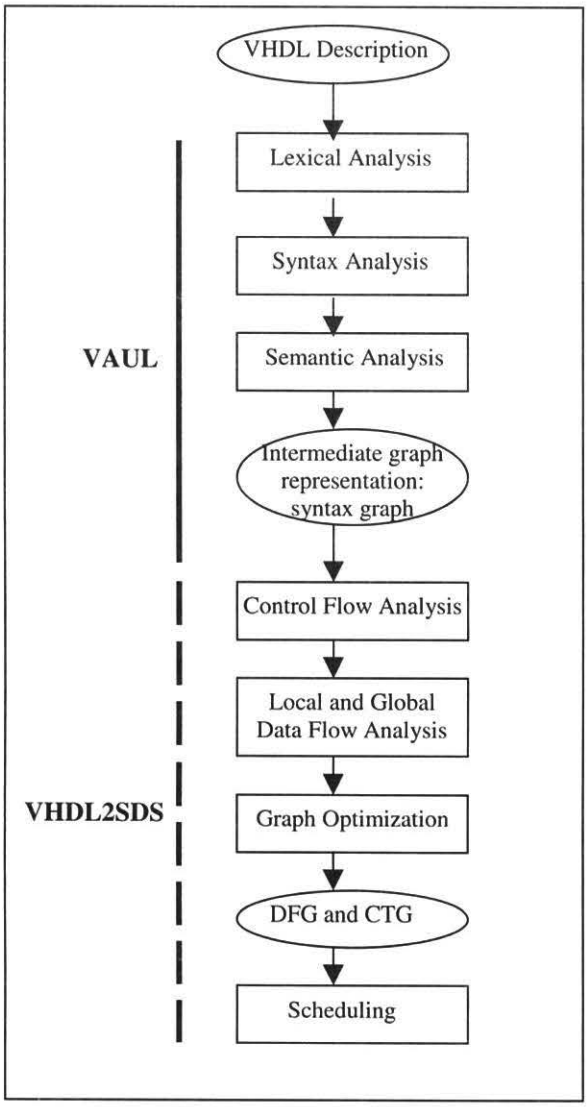

Fig.3 The translation steps

B. Control Flow Analysis

Control flow analysis is a procedure that partitions a sequence of statements into basic blocks and constructs a flow graph that represents the flow of control.

Each basic block is constituted by statements in a sequence from the last branch target to the next one altering the flow of execution.

C. Data Flow Analysis

Data flow analysis is performed in two steps: local data flow analysis and global data flow analysis.

C.1 Local Data Flow Analysis

Each basic block B in the flow graph is a computation unit. Sequential statements are the only forms of statements that will appear in a basic block.

Here, we will analyze the inter-statement data dependency within each basic block. That is, we want to know which statements within a basic block **define** the values for each set of variables used in the statements and who will **use** the values of each set of update variables.

In addition, we need to know the input set of a basic block and the output set. These two sets are important in the global data flow analysis and are determined for the following expressions [CHE90]:

$$I(B) = I(S_i) \cup \{ \bigcup_{K=i+1}^{j} (I(S_k) - \bigcup_{l=i}^{k-1} O(S_l)) \}$$

$$O(B) = \bigcup_{K=i}^{j} O(S_k)$$

Where,
B is a basic block
$B = \{S_i, \ldots, S_j\}$
S_k is a statement and $i \leq k \leq j$
I is the input set
O is the output set

C.2 Global Data Flow Analysis

The inter-block data dependency is much more complex than the intra-block.

Our approach for this global data flow analysis is explained in the following steps:

1. Group basic blocks within the same structure called meta block. This can be done in *if-then-else* or *while-loop* structures. An *if-then-else* meta block consists of three subblocks, a condition, a *then* body and an *else* body. A *while-loop* meta block consists of two subblocks, a condition and a loop body.
2. Calculate the input and output sets of those meta blocks according to table I [CHE90].

TABLE I

INPUT AND OUTPUT SETS

Type of meta block	Input and output sets
if C then T else E	$I(IF) = I(C) \cup I(T) \cup I(E) \cup \{O(IF) - (O(T) \cap O(E))\}$ $O(IF) = O(T) \cup O(E)$
while C loop L	$I(WL) = I(C) \cup I(L) \cup O(L)$ $O(WL) = O(L)$

3. Analyze the *definition* and *use* relationships among the meta block hierarchically

3.1 For the *if-then-else* meta block we have the following rules:

- The *use* values of each variable in the input set is every subblock number whose input set contains it.
- The *definition* value of each variable in the output set is a 2-tuple. The first value is the number of subblock *then* if the variable is found in this structure; otherwise, it is defined outside the meta block. The second value is found in subblock *else* in similar way.

3.2 For the *while-loop* meta block we have the following rules:

- The *use* values of each variable in the input set is every subblock number whose input set contains it.
- The *definition* value of each variable in the output set is a 2-tuple. The first value is the loop body subblock number and the second value is the subblock number defined outside the meta block, which contain the initial value of the variable.
- If a variable is defined in the input set of the loop condition or in the loop body, and in the output set of the loop body subblock, its input *definition* becomes a 2-tuple. The first value is the loop body subblock number and the second is the subblock number defined outside the meta block, which contain the initial value of the variable. If one of the conditions is not satisfied, the value of the tuple is set to zero.

D. Graph Generation

After the global data flow analysis, all the inter-block data dependencies are found and the global DFG and CTG can be generated.

E. Graph Optimization

Like traditional compilers for programming languages, there are plenty of opportunities to optimize the graphs during the VHDL to SDS translation. Here, we focus on those transformations that are guaranteed to improve the final design. The flow graphs and the definition/use information are particularly useful in performing these transformations. We use the following rules:

- For any basic block B_k in a flow graph where $k > 1$, if B_k does not have any incoming edges, then it is dead code.
- For any two operations performing the same function, they should produce equivalent results under the same input condition. Furthermore, if there is a commutative property, the order of inputs may not be important as long as their correspondence can be established. Therefore, one of them can be removed from the graph and the definition and use information of its outputs can be redirected to the other.

F. Scheduling

Scheduling is a very important problem in architectural synthesis. Whereas sequencing graph prescribes only dependencies among the operations, the scheduling of a sequencing graph determines the precise start time of each task.

The number of resources and timing may be bounded or not. We use scheduling without resource constraints, which is applied when dedicated resources are used.

Practical cases leading to dedicated resources are those when operations differ in their types or when their cost is marginal, when compared to that of steering logic, register, wiring and control. Eventually, unconstrained scheduling can be used to derive bounds on latency for constrained problems. The two algorithms used are ASAP (as soon as possible) and ALAP (as late as possible). The ASAP scheduling algorithm yields the minimum values of the start times. A complementary algorithm, ALAP, provides the corresponding maximum values.

III. TRANSLATION EXAMPLE

Figure 4 shows a sample VHDL description that consists of one process. This program counts the number of elements grater than "10" in the vector.

In this description, there are nine basic blocks in the architecture body. Since each basic block consists of single statements only inter-statement data dependencies are necessary. After collecting the input and output sets of each block, we have the use/definition table shown in table II.

```
-- An example VHDL description
package VTYPE is
  type VECTOR is array(0 to 7) of
INTEGER;
end VTYPE;

use work.VTYPE.all;

entity dct is
  port( V: in VECTOR; O: out INTEGER);
end dct;

architecture behavior of dct is
  begin process
  variable  I, A: INTEGER;
  begin
          I := 0;
          A := 0;
         while I < 8  loop
                if  V(I) > 10 then
                          A := A + 1;
                end if;
                 I := I + 1;
        end loop;
       O <= A;
    end process;
  end behavior;
```

Fig.4 A sample VHDL description

TABLE II

GLOBAL DATA FLOW ANALYSIS

Block	Input set			Output set		
	val	def	use	val	def	use
1				I		8
				A		8
2	I	8				
3	V	5				
	I	5				
4	A	5		A		5
5	V	7	3	A	4,7	7
	I	7	3			
	A	7	4			
6	I	7		I		7
7	V	8	5	I	6	8
	I	8	5,6	A	5	8
	A	8	5			
8	V	0	7	I	7,1	
	I	1,7	2,7	A	7,1	9
	A	1,7	7			
9	A	8		O		0

Since there are conditional and loop statements, the CTG generated by VHDL2SDS is not a simple sequence of ranges. The DFG graphs are shown in figures 5 and 6, and the CTG graph is shown in figure 7.

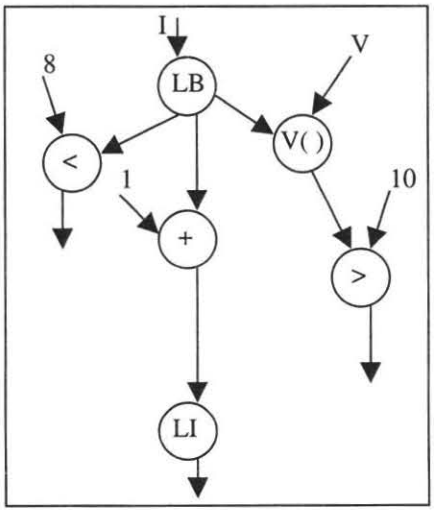

Fig.5 The DFG of variable I

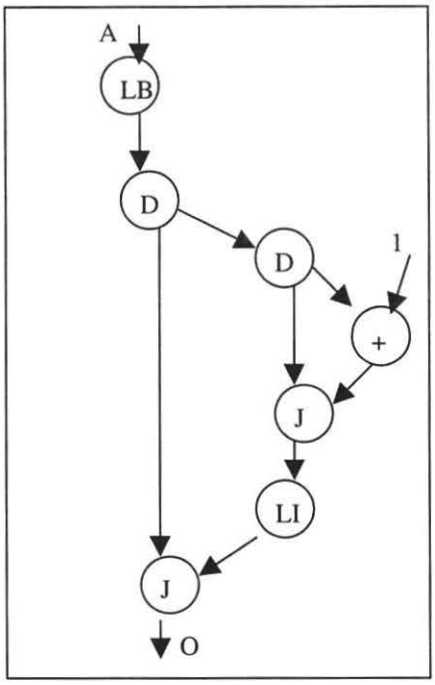

Fig.6 The DFG of variable A

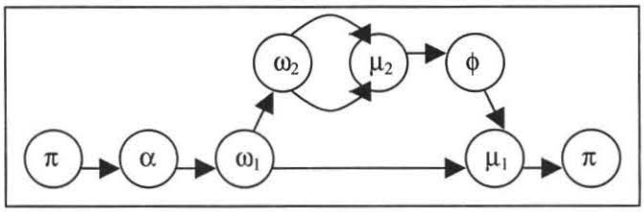

Fig.7 The CTG

Assuming the restriction that each operation take exactly one control step to execute, we have the following ASAP scheduling for the program, shown in figure 8.

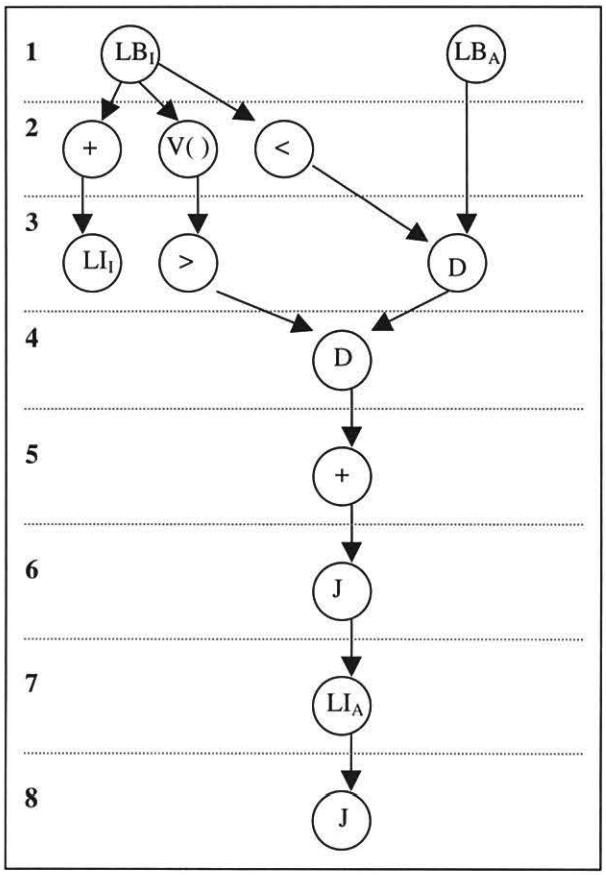

Fig.8 ASAP schedule

Given a final schedule, we can easily compute the number of functional units that are required to implement the design.

IV. Conclusion

The main objective of the translator VHDL2SDS is to obtain an optimized structured representation of a digital system, captured from a behavioral specification, and suitable for validation/verification and synthesis. The translator program is implemented and under tests. The examples submitted so far indicate that the approach taken is feasible. The structured representation of SDS is suitable for validation/verification, circuit optimization, and synthesis.

Future work includes integration of VHDL2SDS to a synthesis system, incorporation of circuit optimization techniques, and implementation of validation/verification tools using the SDS representation. A much larger subset of VHDL may be used, at least incorporating structural constructs, to provide better expressive power for design specification. Other hardware description languages could also be used for the behavioral specification of a digital system.

Acknowledgments

We would like to thank the financial support of Lab. Engenharia de Computadores - LECOM/DCC/UFMG and CNPq under the grant PIBIC/CNPq.

References

[LIP89] LIPSETT, R.; SCHAEFER, C.; USSERY, C. *VHDL: Hardware Description Language*. Kluwer Academic Press, 1989.

[SIL97] SILVA Jr., Diógenes; PARKER, Alice C. *SDS: a System Level Data Structure for Design Representation*. IWLAS 97 - Intl. Workshop on Logic and Architecture Synthesis, Grenoble, France, December 1997.

[VAU94] VAUL. *A VHDL Analyzer and Utility Library*. http://www-dt.e-technick.uin-dortmund.de/~mvo/vaul/ University of Dortmund - Department of Electrical Engineering, AG SIV, 1994.

[KNA85] KNAPP, David; PARKER, Alice C. *A Unified Representation for Design Information*. Proc. of the IFIP Conf. on Hardware Description Languages, Aug. 1985.

[WAL87] WALKER, R. A.; THOMAS, Don E. *Design Representation and Transformation in the System Architect's Workbench*, Proc. ICCAD-87, 1987.

[DUT89] DUTT, Nikil D.; HADLEY, T.; GAJSKI, D. .D. *An Intermediate Representation for Behavioral Synthesis*, Proc. of DAC, 1990.

[CHE90] CHEN, Chi-Tung. *A VHDL Language to DDS Data Structure Translator*. Technical Report, Department of Electrical Engineering - System, University of Southern California, 1990.

[KU90] KU, David; DE MICHELI, Giovanni. *HardwareC- a Language for Hardware design (version 2.0)*. Technical Report CSL-TR-90-419, Stanford University, April 1990.

[DEM90] DE MICHELI, Giovanni; KU, David; MAILHOT, F.; TRUONG, T. The Olympus Synthesis System for Digital Design. *IEEE Design and Test of Computer*, October 1990.

[THO91] THOMAS, D. E.; MOORBY, P. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers,1991.

[IEE87] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE Computer Society Press, March 1997.