COMPILATION AND INTERNAL REPRESENTATION OF DIGITAL SYSTEMS SPECIFICATIONS IN DSL

R. CAMPOSANO*, R. WEBER**

ABSTRACT

This contribution presents the main features of DSL (Digital Systems Specification Language) and its compilation into an internal form. DSL allows imperative and applicative specifications, concurrency at the operation and at the process level and the specification of design constraints such as time, area and power consumption. The internal form includes a first approach to a data-path design, the predecessor-successor relationship of the operations and a timing constraints graph. An example and some performance measures of the current implementation of the compiler conclude the paper.

RESUMEN

Esta contribuciòn presenta las principales caracterìsticas de DSL (Digital Systems Specification Language) y su compilaciòn a una forma interna. DSL permite especificaciones aplicativas e imperativas, paralelismo al nivel de operaciones y al nivel de procesos y la especificaciòn de condiciones de borde como velocidad y àrea. La forma interna incluye una primera aproximaciòn al diseño de la parte de datos, la relaciòn predecesor-sucesor entre las operaciones y un grafo de restricciones de tiempo. El trabajo finaliza con un ejemplo y algunos datos sobre el desempeño de la actual implementaciòn del compilador.

* Forschungszentrum Informatik and ** Institut für Informatik IV (Prof. D. Schmid), Universität Karlsruhe, Haid- und Neustr. 10-14, 75 Karlsruhe 1, F.R.Germany. This work was partially supported by the BMFT (Bundesministerium für Forschung und Technologie) and the SIEMENS AG under grant NT28093.

Digital systems can be described at different levels. We distinguish ideally the behavioural, the structural and the geometrical levels (see also [Shiv83], [CKR84b]). At the behavioural level a system is described in terms of its operations, at the structural level the system is given as the interconnection of modules, and at the geometrical level the exact layout of the masks is specified. Examples are DAISY [John83], ISPS [BBCS79], MIMOLA [Marw84] and DSL [Rose82] at the behavioural level, Zeus [Lieb84b], DDL [DuDi68] and STRUDEL [CaTr84] at the structural level and CIF [MeCo80] at the geometrical level. A bibliography of the numerous existing hardware description languages is given in [Nash84]. Standardization efforts are reviewed in [Waxm84].

The work reported in this paper is part of a larger project which aims at synthesizing IC's from 'pure' behavioural level descriptions. First they are transformed into (also 'pure') structural descriptions which are then converted by a placement and routing system into a geometry (Fig. 1). We deal mainly with the first step, i.e. the transformation of a behavioural description into a structural one. For the transformation of structures into chip layouts we use for example the layout system VENUS [GHHS84] developed by Siemens for automatic placement and routing of CMOS standard cells.

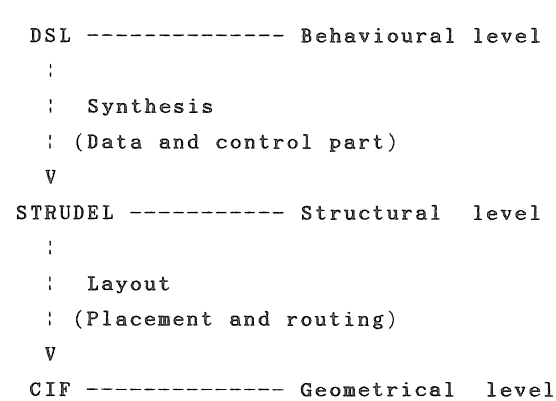

Figure 1. The DSL synthesis approach

There have been reported several approaches to the synthesis of digital systems from behavioural level specifications, mainly 'direct' compilation [DPST81], transformations using graph grammars [GiKn84], data flow analysis [Rose84] and expert systems [KoTh83b]. While some results reported suggest that synthesis (in the above sense) yields reasonable circuits, there is strong evidence that none of these approaches alone can claim to be general, i.e. to be feasable for a large set of designs.

Because of this lack of 'general' techniques, we separate synthesis into different steps, some of them algorithmic and some of them not. The compilation of a behavioural level specification into a first, not at all optimized design of the data path structure and the control sequencing is our main topic in this paper. This occurs in a pure algorithmic way (e.g. no 'artificial intelligence'), leaving perhaps more 'intelligent' activities for later steps.

The language presented in this paper evolved from former versions reported in [CaRo80], [Rose82], [CKR84a], [Rose84]. The additions and changes yielded, even if the name remained the same, a significantly different language. A detailed description of the language and the internal form can be found in [CaWe84], [CaWe85]. In the following we discuss this language and its distinctive features (chapter 2). The internal form into which DSL is compiled defines (only partially as we will see) the semantics (chapter 3). The compiler is presented with some emphasis on the performed consistency checks and some performance measures of our actual implementation (chapter 4). We conclude with a brief summary and some open questions (chapter 5).

2. THE DIGITAL SYSTEMS SPECIFICATION LANGUAGE DSL

To specify digital systems at the behavioural level several languages have been used. The definition of a specific language offers the advantage that the language can be designed according to the special needs of the application. Another approach consists in using an existing programming language such as

PASCAL or ADA [GiKn84] with the obvious advantages. Our main interest in defining DSL was to provide the means for a 'pure' behavioural specification, including timing and electrical behaviour to some extent. The power and comfort of modern programming languages should be given. The main problems we had do deal with were

- what style of specification, i.e. imperative or applicative should be used,
- how to specify concurrency,
- how to allow modularity and hierarchies,
- how to include the description of timing and electrical behaviour and
- which design constraints should be included.

In general DSL shows some similarity with PASCAL. The control structures are the same, i.e. IF-THEN-ELSE, CASE, WHILE-DO, DO-UNTIL and FOR-DO. The data structures are restricted to ARRAYS, while the basic data types are LOGICAL, ONE_COMP, TWO_COMP, BCD, FIXED and FLOAT. The allowed operators include the usual logical, arithmetic and relational operators. It should be noticed that the choice of basic data types and operators is reflected in the basic hardware modules that must be available, e.g. floating point division, two's complement addition, etc. Of course these modules can also be built by a synthesizer from simpler primitives.

A DSL program may include one applicative part and as many imperative processes as necessary. Global actions not constrained to a certain point in time, such as resets, interrupts, etc. are naturally specified in the applicative part. Sequential algorithms, the behaviour of finite automata and sequential behaviour in general are often more comfortably described with imperative procedures.

Concurrency can be specified in DSL at the operation and at the process level. Concurrent operations must be separated by commas inside a FORK-JOIN construct. FORK (JOIN) can be abbreviated by [()). To allow control over concurrent processes, control operations nre provided. They are RESET, START, STOP and CONTINUE and have the following meaning:

OPERATION	DATA PART	CONTROL PART	ACTION
RESET	INITIAL STATE	INITIAL STATE	STOP EXECUTION
STOP	SAME STATE	SAME STATE	STOP EXECUTION
START	SAME STATE	INITIAL STATE	START EXECUTION
CONTINUE	SAME STATE	SAME STATE	START EXECUTION

Figure 2. Imperative Procedure control operations in DSL

The control operations can be applied only to imperative (sequential) procedures and allow, inside a FORK-JOIN, the specification of concurrent processes.

Modularity in DSL is partially achieved by permitting as many imperative processes as necessary. Moreover, a powerful abstraction mechanism permits also hierarchical specifications. The functions performed by a DSL program are (optionally) specified in the PERFORMED FUNCTION statement. Each function is characterized by its name, the inputs (interface, pins) it uses as arguments, the generated outputs (interface, pins) and the control signal sequence that must be applied to the remaining circuit interface to perform exactly this function. The name of the function can be used as an operator in other DSL programs that can also be compiled separately.

The electrical behaviour can be specified only globally and at the interface, e.g. FANIN and FANOUT, VOLTAGE LEVEL, POWER, etc. It is used to select the technology, the interface drivers and as a global constraint for the synthesis process (e.g. power consumption). Timing can also be specified as a global constraint (FREQUENCY). Moreover, it is possible to specify the

delay of any operation or group of operations in absolute time or in clock cycles. This feature is intended to impose timing restrictions to the synthesis, thus allowing to specify the delay in critical paths or other portions of a design.

Besides synthesis the other important application of DSL is automatic test pattern generation and design for testability. We are currently developing two approaches. Roughly the first one consists of the random test pattern generation for the combinational logic using a probabilistic method to estimate the fault coverage and making the storage elements accessible by a scan path or BILBO's [Wund84], [KuWu84]. The second approach attemps to generate test patterns already at the behavioural level by imposing certain restrictions to the allowed operations.

Figure 3 shows an example of a DSL program for calculating out=exp(in) using the Taylor series expansion.

```
   CIRCUIT exponentiation;
(* sequential circuit that calculates output=e**(input) *)
   INTERFACE
           vcc            : 12V;
           gnd            : GND;
           input(15..0)   : INPUT FANIN 1;
           output(15..0)  : OUTPUT FANOUT 10;
           enable         : INPUT FANIN 1;
           clk            : CLOCK FANIN 1;
   POWER         100 mW;
   VOLTAGE       12.00  V;
   HIGH LEVEL    11.00  V;
   LOW  LEVEL     1.00 V;
   TECHNOLOGY    CMOS;
   AREA          30 mm2;
   FREQUENCY 0 TO 500 kHz;

   PERFORMED FUNCTION
           output:=#exp(input)
           CONTROL
              (enable:='0' CYCLES=1);
              (enable:='1' CYCLES=48)
              END;
```

```
VAR     x,y : FIXED (8,8);
        i   : LOGICAL (4..0);
CLOCKBASE clk;

  APPLICATIVE
     output:=y,
     IF enable=0 THEN x:=1,y:=1
                 ELSE START calc
     FI
  END APPLICATIVE;

  IMPERATIVE calc;
     (FOR i:=1 TO 16 DO
         x:=x*input/i;
         y:=y+x
      OD CYCLES=3)
  END IMPERATIVE;

END.
```

Figure 3. Example DSL program

3. SEMANTICS OF DSL: THE INTERNAL FORM

The semantics of DSL can be partially defined by the internal form generated by the compiler. This form consists essentially of 3 graphs that share the same vertices. The vertices in the graphs are the operations in the DSL program.

Let P be a DSL program, V the set of operations that appear (textually) in P and W the set of variables, constants and imperative procedures in P plus some auxiliary variables. Then the internal form S of P is given by

$$S=(G_1,G_2,G_3)$$

$$G_1=(V,E_1)$$

$G_2 = (V\ W, E_2)$

$G_3 = (V, E_3)$

G_1, G_2 and G_3 are directed graphs. The edges E represent

Sequence:
e=(v,v') in E_1 <=> v before v' and v, v' in the procedural part

Data-flow:
e=(v,w) in E_2 <=> w in W is an output of v in V
e=(w,v) in E_2 <=> w in W is an input of v in V

Timing:
e=(v,v') in E_3 <=> there is a timing constraint for a group of operations
1. starting with v and ending with v'
2. ending with v and starting with v'

The vertices and edges have the following attributes:

v in V	operation(v),	e.g. ADD, AND, etc.
	type(v),	e.g. Loop beginning, IF, etc.
	condition(v),	of the form variable=value
w in W	type(w),	e.g. two_complement, fixed
	width(w),	e.g. (15..0)
	array_dim(w),	e.g. [8..1]
e in G_1	condition(e),	of the form variable=value
e in G_2	array_index(e),	the index range if w in e=(v,w) or e=(w,v) is an array, e.g. [2..1]
	width_selection(e),	selects the used bits of w in e=(v,w) or e=(w,v), e.g. (8..1)
e in G_3	timing(e),	i.e. time or cycle delay

The nodes in V (operations) are identified by a unique index, i.e. a natural number. The nodes in W (variables) are identified

by the variable name, the constant name, the procedure name or, in the case of auxiliary variables, by a name of the form SYMB_nmb (nmb a natural number). Notice that for the applicative part, G_1 does not exist, i.e. for a pure applicative specification $S=(G_2,G_3)$.

Besides the internal form S the compiler generates a symbol table and identifies the context of the variables, i.e. in which imperative or applicative parts a variable is used. Also the global constraints are included in the output.

The output of the compiler for the example of fig. 3 is given partially in fig. 4. The first vertex represents the comparison of i with 16. The index is the internal identification, the tag indicates the attribute type of the operation, in this case a 'For Test-Single' which means the single testing operation of a FOR loop. Operation GRT stands for 'GReater Than'. The inputs are the variable i of type LOGICAL and the constant 16; the output is the internal generated variable SYMB_2 of undefined width. Both inputs and outputs define G_2. Predecessor and successor give the edges of G_1, i.e. the index of the corresponding vertices. The succesor edges have the attribute condition, in case of the operation GRT the value of the output. Finally, the timing constraint graph G_3 is given indicating the edge (index of the vertex) and the attribute (EQuaL 3 cycles).

The second operation is the MULtiplication of x and input, its type is 'Expression Begin' and it has no local timing constraints. The complete internal form is given in figure 5 in a graphical representation, with attributes attached to the vertices and edges, and the identification of the nodes inside them. For operations these are natural numbers, for variables their names.

```
Index       : 8       Tag : FTS
Operation   : GRT
Inputs      : i(4..0) : LOG    16
Outputs     : SYMB_2(*..*) : AUX
Predecessor: 11  12
Condition   : 0      Successor : 13
Condition   : 1      Successor : *
Succ.cycle : 3       Index : 12       Relation : EQL

Index       : 13       Tag : EB
Operation   : MUL
Inputs      : x(15..0) : FIX     input(15..0) : IN
Outputs     : SYMB_1(*..*) : AUX
Predecessor: 8
```

Figure 4. Two vertices in the compiler output for exponentiation

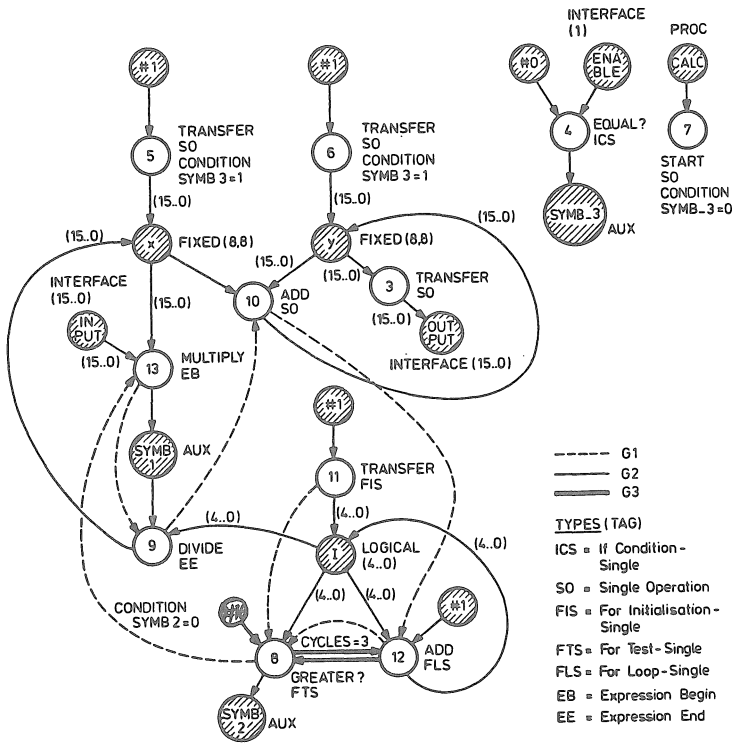

Figure 5. Internal form for exponentiation

The internal form defines an exact predecessor-successor relationship (G_1), the context of each operation (G_2) and the timing constraints (G_3). Global constraints are also included. The output of the compiler is used in subsequent steps for the synthesis of structures and for test pattern generation at the functional level. It is interesting to notice that G_2 already gives a first direct implementation of the data-path by replacing the vertices W by registers and the vertices V by the corresponding operations. G_1 provides the basis for a straight forward control part allocation by activating the operations in the sequence G_1 indicates. Of course the timing constraints given by G_3 and the global constraints must be met in subsequent optimization steps.

The semantics of DSL deliberately do not define the resulting type of many operations, e.g. what should be the type of an (intermediate) result in hardware of the division of a fixed point register by a floating point register? This kind of indefinition must be resolved at a later stage in the synthesis process, either by defining strict rules or, what seems more adequate for hardware synthesis, by interaction with the designer. Another indefinition is the result of the use of common variables in concurrent imperative processes. Also in this case many intended cases may exist, and imposing for example one synchronisation mechanism would be too restrictive. An example might be the concurrent incrementing and decrementing of a register by two processes. In case of doing this in the same cycle, the intended result might not alter it. Of course this could be specified explicitly by merging both processes, or by defining a new process that controls the access to the register, but this tends to produce clumsy specifications. Nevertheless, these situations are identified so that they can be resolved at a later time.

4. THE COMPILER

The transformation of DSL into the internal form is done by compilation. Beyond the usual tasks of a compiler, the DSL compiler achieves a certain kind of consistency of the DSL

specification by checking consistency conditions. This is very important to discover specification errors at the beginning of the design process. The cost of an error increases very fast as the design advances. The conditions implemented in this compiler include:

- Information can not be read from output interfaces or written into input interfaces
- Inside an imperative part no concurrent write or read-write is allowed
- Undefined resulting data types from intermediate operations are marked
- Variables that might generate synchronisation problems because they are used in two or more concurrent imperative parts or in the applicative part are identified
- An imperative part can not control itself

The current version of the compiler was implemented using the compiler-compiler system GAG [KHZ82]. The input to GAG is an attributed grammar in ALADIN. The output is a Pascal program, the complete compiler. Attention was paid to mantain efficiency, even though using such a high level tool. The implementation time was roughly 6 man months. The main characteristics of the compiler are

- 6400 ALADIN code lines for the source
- 27800 Pascal code lines for the generated output
- 467 KB for the object code on a Siemens 7751

The performance for some examples, including the exponentiation given in chapter 2, is shown in figure 6.

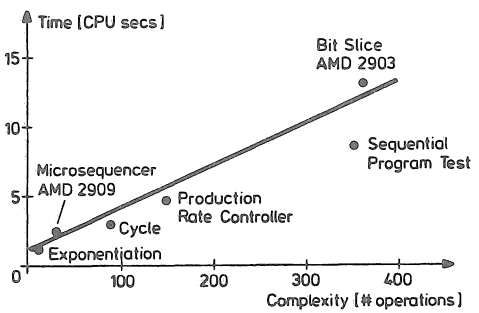

Figure 6. Time performance of the DSL compiler

5. SUMMARY

The language DSL, its internal form S and the DSL compiler were presented. The distinctive features of DSL are both applicative and imperative specification styles, modularity and hierarchies by an abstraction mechanism for functions, specification of concurrency at the operation and at the process level and the inclusion of timing constraints and global electrical constraints. The internal form consists mainly in 3 graphs with common vertices (the operations) which represent the predecessor-successor relationship, the data path and the local timing constraints. A relatively efficient version of the compiler was implemented. Consistency conditions permit the discovery of some specification errors.

The main applications of DSL are the synthesis of structures from behavioural level specifications and the test pattern generation at a functional level. For both applications partial results have already been reported and are currently under development. Interesting problems among others include the properties operations must have to allow a test pattern generation at this level, which synthesis techniques to use to obtain results that meet the specified constraints and how to integrate test and synthesis.

6. REFERENCES

[BBCS79] M.R. Barbacci, G.E. Barnes, R.G. Cattell, D.P. Siewiorek
The ISPS Computer Description Language
Department of Computer Science,
Carnegie-Mellon University, CMU-CS-79-137, 1979

[CaRo80] R. Camposano, W. Rosenstiel
Algorithmische Synthese deterministischer (Petri-) Netze aus Ablaufbeschreibungen digitaler Systeme
Interner Bericht Nr. 22/80, Fakultaet fuer Informatik, Universitaet Karlsruhe, 1980

[CaTr84] R. Camposano, L. Treff
STRUDEL: Eine Sprache zur Spezifikation
der Struktur digitaler Schaltungen
Interner Bericht Nr. 7/84, Fakultaet fuer
Informatik, Universitaet Karlsruhe, 1984

[CaWe84] R. Camposano, R. Weber
DSL - Eine Sprache zur Spezifikation
digitaler Schaltungen
Interner Bericht Nr. 24/84, Fakultaet fuer
Informatik, Universitaet Karlsruhe, 1984

[CaWe85] R. Camposano, R. Weber
Semantik und interne Form von DSL
Interner Bericht 3/85, Fakultaet fuer
Informatik, Universitaet Karlsruhe, 1985

[CKR84a] R. Camposano, A. Kunzmann, W. Rosenstiel
Automatic Data Path Synthesis from
Behavioural Level Descriptions in DSL
Proc. VLSI: Algorithms and Architectures
Edited by P. Bertolazzi, F. Lucio
North Holland, 1984

[CKR84b] R. Camposano, A. Kunzmann, W. Rosenstiel
Automatic Data Path Synthesis from DSL
Specifications
Int. Conference on Computer Design ICCD'84
Port Chester, October 1984

[DuDi68] J.R. Duley, D.L. Dietmeyer
A Digital System Design Language (DDL)
IEEE Transactions on Computers,
Volume C-17, September 1968

[DPST81] S.W. Director, A.C. Parker, D.P. Siewiorek,
D.E. Thomas
A Design Methodology and Computer Aids for
Digital VLSI Systems
IEEE Transactions on Circuits and Systems,
Volume CAS-28, Number 7, July 1981

[GiKn84] Girczyc,E.F., Knight,R.P.
An Ada To Standard Cell Hardware Compiler
based on Graph Grammars and Scheduling
Proc. Internatinal Conf. Computer Design 1984
Port Chester, N.Y., October 1984

[GHHS84] Göttler,E., Haschigh,L., Hörbst,E., Sandweg,G, et al
Entwicklung von kundenspezifischen Schaltungen
Elektronik, Hefte 19,20,21,22, 1984

[John83] Johnson, Steven D.
Synthesis of Digital Designs from Recursion Equations
MIT Press, 1984

[KoTh83b] T.J. Kowalsky, D.E. Thomas
The VLSI Design Automation Assistant: Prototype System
20th Design Automation Conference, IEEE, August 1983

[KuWu84] Kunzmann,Arno, Wunderlich,Hans J.
Steigerung der Effizienz beim Test mit Zufallsmustern
Report 19/84 of the Faculty for Informatics, University of Karlsruhe, October 1984

[KHZ82] U. Kastens, B. Hutt, E. Zimmermann
GAG: A Practical Compiler Generator
Lecture Notes in Computer Science 141
Springer Verlag, 1982

[Lieb84b] Lieberherr, Karl J.
Chip Design in Zeus and Proposal for Standard Benchmark Set for HDL
Proc. International Conf. on Computer Design ICCD'84, Port Chester, N.Y., October 1984

[Marw84] Marwedel, Peter
The MIMOLA Design System: Tools for the Design of Digital Processors
Proc. of the 21st Design Automation Conf., Albuquerque, New Mexico, June 1984

[MeCo80] C. Mead, L. Conway
Introduction to VLSI Systems
Addison-Wesley, 1980

[Nash84] Nash, J.D.
Bibliography of Hardware Description Languages
SIGDA Newsletter, Vol. 14, February 1984

[Rose82] W. Rosenstiel
DSL - Eine Sprache zur Spezifikation der Funktion digitaler Systeme : Konzept und Implementierung
Interner Bericht Nr. 9/82, Fakultaet fuer Informatik, Universitaet Karlsruhe, 1982

[Rose84] Rosenstiel, Wolfgang
Synthese des Datenflusses digitaler Schaltungen aus formalen Spezifikationen
Dissertation at the Faculty for Informatics, University of Karlsruhe, VDI-Verlag, 1984

[Shiv83] S.G. Shiva
Automatic Hardware Synthesis
Proceedings of the IEEE,
Volume 71, Number 1, January 1983

[Waxm84] Waxman, R.
The Many Languages of Electronic Computer Aided Design
Proc. International Conf. on Computer Design ICCD'84, Port Chester, N.Y., October 1984

[Wund84] Wunderlich, Hans J.
Zur statistischen Analyse der Testbarkeit digitaler Schaltungen
Report 18/84 at the Faculty for Informatics, University of Karlsruhe, August 1984