PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 35, NO. 1, PP. 263-272 (1991)

THE USE OF A FUNCTIONAL DESCRIPTION LANGUAGE FOR TEST GENERATION

J. SZIRAY and ZS. NAGY

Computer Research and Innovation Center, SZKI
H-1251 Budapest, Hungary

Received: Febr. 1, 1991.

Abstract

The paper presents a new hardware-description language (OPART) which serves for specifying the functional models of logic elements for an automatic test-generation program. The significance and novelty of OPART rely on the fact that it enables in a user-oriented way to define models not only for fault simulation, but also for algorithmic test calculation. In the paper the basic syntactic features of OPART are first discussed. After that examples and the computer implementation will be overviewed.

Keywords: : Test pattern generation, fault simulation, hardware-description languages, logic modelling.

Introduction

The automatic test-generation programs (ATGP's) used in digital electronics model the circuit elements (modules) by means of built-in subroutines. As a consequence of this fact, the scope of logic diagrams accepted by an ATGP greatly depends on the set of the modelling routines. Whenever the extension of this set is required, the task is solved by way of additional software development on the original program. From the user's point of view, the software development is infeasible. Instead, the user is interested in a system that enables the logic behaviour of a module to be defined by a special hardware-description language (HDL). In this case, the compiler of the HDL produces an object code in a library which can then be processed by the original ATGP.

As seen, the approach outlined is similar to that used in modern logic simulators which serve for design verification. On the other hand, in the case of an ATGP the task of modelling is usually not the same. As a rule, an ATGP involves a test-calculation component and a fault simulator, which require different functional models, whereas a logic simulator has only one model.

The aim of our paper is to present an HDL that is meant for the purposes described above. The language (called OPART) has been developed for the connecting test-generation program DIAS (SZIRAY, 1987). DIAS handles the logic elements of a circuit on the functional level (BREUER – FRIEDMAN, 1980). It means that the logic values of an element are evaluated with the knowledge of its external functional behaviour. The built-in modules in the system DIAS are various logic gates and flip-flops, bus element, and transfer gate.

It should be added that, as known, the hierarchical level of logic modelling influences the feasibility and computational efficiency of test calculation algorithms to a great extent. The importance of functional level models versus gate-level ones is increasing rapidly with the growing complexity in the VLSI area. Here the Application-Specific Integrated Circuits (ASIC's) are also to be mentioned, which have customized functional cells as fundamental elements.

Requirements to the Language

DIAS includes a fault simulator and three components for automatic test generation. Fault simulation is performed on the ground of the so-called *concurrent approach* (ULRICH – BAKER, 1974), (BREUER – FRIEDMAN, 1976). The test generator part applies a random, an adaptive random, as well as a deterministic method.

The random and adaptive random processes are influenced and controlled by the circuit behaviour upon the generated tests. Here the behaviour is monitored by means of fault simulation. The deterministic test calculation applies the so-called *composite justification algorithm* (SZIRAY, 1979), (SZIRAY, 1982). The algorithm is based on calculations performed simultaneously in the faulty and fault-free circuit versions, where the only computational means is iterative line-value justification (BREUER – FRIEDMAN, 1976). Line-value justification is a procedure with the aim of successively assigning input values to the logic elements in such a way that they are consistent with each previously assigned value. The salient advantage of composite justification is the total absence of the fault propagation phase. This feature has greatly facilitated the use of an HDL.

As known, concurrent fault simulation and composite justification use different models for a logic element. The simulation model defines the output and next state values as a function of the input and present state values. It will be referred to as the *normal model.* The other model is called *inverse* and it serves for line-value justification. An inverse model defines the set of possible input patterns which result in a specific state or output pattern. Since the evaluation of a logic element within DIAS is done in the unit-delay mode, the normal and inverse models do not involve timing information.

DIAS handles a five-valued logic system. This system incorporates the following logic values: logic 0, logic 1, unknown (u), high impedance (Z), and don't care (d). In order to be compatible with DIAS, the OPART language must handle the same multi-valued logic.

The complexity of a logic module is not limited in principle. Only practical trade-offs, such as processing time, and user efforts in terms of modelling accuracy are to be considered. It should also be added that the existence of an inverse model is not a prerequisite, since in case of its absence only the deterministic test calculation will be omitted in a DIAS run.

Finally, considering the wide-spread D-algorithm as a test-calculation tool (BREUER – FRIEDMAN, 1976), (ROTH, 1966), it is known that the algorithm needs two functional models: a model for the D-propagation through an element, and another one for line-value justification. In principle, it is feasible to elaborate an HDL to meet these modelling requirements. However, such a language would be more complicated than that which does not handle D-propagation, as in the case of OPART. Moreover, the inevitable design of the propagation D-cubes for a module is a considerable additional burden for the user (BREUER – FRIEDMAN, 1980).

In the following the basic syntactic features of OPART are discussed. After that examples and the computer implementation will be overviewed.

Features of the Syntax

Declarations:

The declarations statements are for prescribing the program variables, constants, I/O, state and bus variables of the logic module. The use of vectors and arrays is also allowed. Examples of declarations:

```
INTEGER     alfa, beta [10], ro [3,5,8];       {integer variable,
                                               vector, and array}
CONST     six = 6, bin_ four = B'0100', ali = X'BABA';
                          {integer, binary, and hexadecimal constants}
BIN     seq [16] (3), bit;       {binary vetor and variable}
IN     select_ vector [0:7], contr [4], strobe;      {inputs}
OUT     count [0:15], sum [4], carry_ bit;      {outputs}
STATE     mem [128], store [2];      {internal states}
```

Here the brackets denote vectors and arrays, while the parentheses include the length value. A from-to index range can be designated by a colon as a separator.

Subroutines:

The number of subroutines within a functional program is arbitrary. The subroutines are in close relationship with the main program, so they must be included within the frame of it. A subroutine can have formal parameters of any type. Example for a subroutine head where integer, hexadecimal, input, and output type parameters are declared:

```
SUBR bus_ check ( INTEGER alfa, omega [12];
                  HEX grand [6] (3),  can[2], yon;
                  IN der, die, das;    OUT des [4] )
```

Arithmetic and Logic Operations, Relations:

OPART has two arithmetic operations: addition and subtraction. Furthermore, the following logic operations are allowed: AND, OR, NAND, NOR, INVERSION (NOT), EXCLUSIVE OR (XOR), and EQUIVALENCE (EQV).

These operations can be executed either with two operands, or with a single operand. For example,

NOT (stan AND pan) EQV ben

yields a bit string of the same length as the operands have, while

XOR asterisk

is a one-bit result of XOR-ing the components of the vector *asterisk*.

Concatenation of strings and designation of subranges in a string are also permissible. The language handles the usual comparative relations, true/false logic conditions, and conditional expressions. It should be mentioned that the consistency relation between two logic values is also processed. For example, 0 and 1 are inconsistent, while *d* and any other value are consistent with each other. The relations are defined for string variables too, where multivalued logic is processed. If, for instance,

art = B'101',   bart = B'0d1',   cart = B'0Zd',

then

art ˜ bart   and   bart ˜ cart

are false and true consistency relations, respectively.

Software functions:

Another important feature of the language is the use of software generators which are standard functions. These functions serve for generating regular bit sequences, where the argument is an arbitrary logic or arithmetic expression. The standard functions are as follows: incrementation, decrementation, various shift operations, inversion, computation of odd or even parity bits. As an example consider the following statement:

eta = INCR (kappa AND lambda XOR SHL (mu)).

Here INCR increments the actual value of its argument by one. This argument is computed in the following way: the bit by bit logic multiplication of *kappa* and *lambda* is added by the exclusive or operation to the bit vector of *mu* that had been shifted to the left by one position.

Structuring statements:

The flow control of an OPART program is organized by means of the following statements: GO TO, IF-THEN-ELSE, CASE for unconditional and conditional branching, and FOR-DO, WHILE-DO for looping. The structuring statements can arbitrarily be nested with each other.

The activation of a subroutine is carried out by the CALL statement where the actual values of the parameters are also given.

The counter variable:

In case of an inverse program, usually more than one input patterns belong to an output or state pattern. To keep account of the different solutions OPART maintains a dedicated *counter variable* with the fixed keyword #TURN.

Examples

In this section three brief examples will be shown to illustrate the use of the language.

1) The first example is the normal functional description of the parity generator circuit of type SN74180.

```
PROGRAM   /NORMAL/   pargen
IN        in_vect [8],    {Input byte to be checked for parity}
```

```
        in_even,       {Logic level of even parity}
        in_odd;        {Logic level of odd parity}

OUT     sum_even,      {Signal for even parity}
        sum_odd;       {Signal for odd parity}

sum_even =  NOT in_even
            AND   (NOT in_odd OR    PARODD (in_vect))
            OR   NOT in_odd AND     PAREVEN (in_vect);

sum_odd =   NOT in_even
            AND   (NOT in_odd OR    PAREVEN (in_vect))
            OR   NOT in_odd AND     PARODD (in_vect)

 END PROGRAM    {pargen}
```

The only comment to the above program is that PARODD and PAREVEN are the standard functions for generating a logic 1 for an odd parity and even parity vector, respectively.

2) The second example is an OPART description of the 8-bit shift register of type SN74198. Here *clear* is the clear input, *clock* is the clock signal, *data* is the vector of data inputs, $s0$ and $s1$ are control inputs, sin l and sin r are the serial inputs of shift left and shift right, respectively, while q denotes the 8-bit output. The register is edge-triggered. In order to handle this control mode, an extra state bit is introduced in addition to the eight storage bits. For the sake of brevity, only the logic values 0 and 1 are taken into account in this functional specification.

```
PROGRAM    /NORMAL/    shift_8

IN    s0, s1, sinl, sinr, data [8], clear, clock;

OUT    q [8];

STATE    clock_st, q_st [8];

IF    clear == 0    THEN    q_st = B'0' * 8 {Each bit is set to zero}

ELSE

    IF    (clock_st == 0)    AND    (clock == 1)    THEN

                       {Rising edge of clock occurred}

    CASE    s0 | s1
    OF   B'11':   q_st = data                 {Parallel load}
    OF   B'10':   IF   sinl == 1              {Shift left}
                  THEN    q_st = SHLONE (q_st)
                  ELSE    q_st = SHL (q_st)
                  END IF

    OF   B'01':   IF   sinr == 1              {Shift right}
                  THEN    q_st = SHRONE (q_st)
                  ELSE    q_st = SHR (q_st)
                  END IF

    END   CASE

    END   IF

END   IF

q = q_st;           {Setting of the output values}

clock_st = clock          {Storing of the clock-signal value}

END   PROGRAM    {shift_8}
```

It can be seen that the compound statements are terminated by their keywords followed by END, e. g., IF – END IF. A CASE section is designated by the form 'OF actual constant:'. The software generator SHLONE shifts the bits of *q* to the left by one position where the entering bit is 1. SHL does the same, with the entering bit 0. In a similar way, SHRONE and SHR result in shifting to the right, with the entering bit 1 and 0, respectively.

3) The third example is the inverse functional program of an 8-bit multiplexer.

```
PROGRAM /INVERSE/ multiplex

IN   addr [3],          {3 address bits}
     data [0:7],            {8 data bits}
     enable;            {Enable input}

OUT   ex;      {Output}

data = B'd' * 8;      {Each data bit is set to don't care}

IF       (ex == 0)    OR    (ex == 1)    THEN

         enable = 1;
```

```
        IF      #TURN <= 8   THEN

                addr = #TURN - 1;
                data [#TURN] = ex

        END     IF

        IF      (#TURN > 8)   AND   (ex == 1)   THEN

                #TURN = -1

        END     IF

        IF      (#TURN > 8)   AND   (ex == 0)   THEN

                enable = 0;
                addr = B'd' * 3;
                #TURN = 0

        END     IF

ELSE

        IF      ex == B'd' THEN            {The output value
                                            is don't care}
                addr = B'd' * 3;
                enable = B'd';
                #TURN = 0

        ELSE              {The output value is unknown}

                #TURN = -1

        END     IF

END     IF

END     PROGRAM    {multiplex}
```

As for the use of #TURN the following rules are to be considered:

— The iterative activization of the program *multiplex* is done from DIAS, where the actual value of *ex* is an input to the inverse program.

— Starting with #TURN = 0, the incrementation of #TURN by one before each activization of *multiplex* is executed in DIAS.

— The inverse program can also set the counter value, with the following meanings:

#TURN = 0:	the last solution (input pattern) has been produced for DIAS;
#TURN = −1:	there is no existing solution.

The Compiler and the Processing Environment

The OPART compiler operates in two passes. In the first pass the syntactic and semantic analysis of the source text is accomplished. The second pass serves for the translation when the object code is generated. The compiler is capable of detecting more than 100 user errors.

The code is a sequence of various command records in a concise form. Each record contains a command code and the necessary parameters. These records are processed in DIAS by a so-called *run-time system* (RTS).

The RTS has the following main functions:

— It maintains a permanent communication link between the object code and the fault-simulation/test-calculation component of DIAS. This is achieved through a direct access to the input, output and state variables in the code.

— It interprets the command records of the code in the required order, thus performing all the necessary calculations in the actual run-time situation.

The test-generation system DIAS runs on DEC-VAX computers, in VMS operating system. The OPART compiler has been developed in the same environment, and also on IBM-PC-XT/AT in MS-DOS operating system. Both versions and the run-time system are programmed in PASCAL language.

The compilation speed experienced so far on a MicroVAX II has ranged from 500 to 2000 lines/minute, with the inclusion of cross-reference lists.

Thus far the normal and inverse functional descriptions of the following modules have been completed: multiplexer, decoder, shift register, counter, comparator, parity generator and various combinational elements (cells) of gate-array circuits. These models have also been involved in several test generation runs with satisfactory efficiency.

Finally, it should be noted that the normal and inverse codes of a logic module must be consistent with each other. To facilitate for the user to achieve this goal a code verification program has been developed. In our solution the two codes are passed to the verifier program which processes them alternately and compares the corresponding input/output values computed in this way. Within a computational phase, first an inverse code is taken. Here the input patterns obtained are sent one-by-one to the normal code where the outputs are calculated. Thus the verifier program is able to detect any inconsistency, both at an input and at an ouput line. It can be seen that the process outlined is essentially a two-direction simulation of a logic element.

Acknowledgements

The authors wish to thank G. TEMESVÁRI and A. DÉRI for their valuable advice in developing the OPART language.

References

BREUER, M. A. - FRIEDMAN, A. D. (1976): Diagnosis and Reliable Design of Digital Systems, Computer Science Press, USA,

BREUER, M. A. - FRIEDMAN, A. D. (1980): Functional Level Primitives in Test Generation, *IEEE Trans. on Computers,* Vol. C-29, pp. 223-235, March 1980.

ROTH, J. P. (1966): Diagnosis of Automata Failures: a Calculus and a Method, *IBM Journal of Research and Development,* Vol. 10, pp. 278-291, July 1966.

SZIRAY, J. (1979): Test Calculation for Logic Networks by Composite Justification, *Digital Processes,* Vol. 5, No. 1-2, pp. 3-15.

SZIRAY, J. (1982): Functional Level Test Calculation and Fault Simulation for Logic Networks, *Discrete Simulation and Related Fields* (Edited by A. Jávor), pp. 223-234, North-Holland Publishing Company, Amsterdam.

SZIRAY, J. (1987): The Test-design Program System DIAS, *The First Hungarian Custom Circuits Conference, Proceedings,* pp. 303-309, Gyöngyös, May 1987.

ULRICH, E. G. - BAKER, T. (1974): Concurrent Simulation of Nearly Identical Digital Networks, *Computer,* Vol. 7, pp. 39-44, April 1974.

Address:

József SZIRAY
Zsolt NAGY
Computer Research and Innovation Center
P.O.Box 19, H-1251, Budapest, Hungary