BEHAVIOURAL VHDL DESCRIPTION BASED SYNTHESIS OF SELF-CHECKING CIRCUITS

András Petri, jr. - András Pataricza - Endre Selényi
Department of Measurement and Information Systems, Technical University of Budapest
E-mail: petri@mit.bme.hu

1 Introduction

This paper demonstrates an experimental implementation of a design method for embedding fault tolerance capabilities into high level digital system models. The method starts with a standardized behavioural level system description and systematically transforms it to an implementation-level circuit design with fault tolerant parts built in. The transformation process aims to keep the changes made in the model transparent from the viewpoint of the designer, in order to maintain compatibility between the original system model and to minimize the manual interaction needed to implement fault tolerance.

The method is intended to be easily incorporated into existing digital system design environments, therefore it applies the de facto industry standard VHDL language both as input and output format. The implantation of the fault tolerant capabilities is performed by *replacing common VDHL data types* with alternate, self-checking capable versions. This way the initial high level model *needs only a minimal modification*, and maintains compatibility with high level simulation and verification tools. The code parts implementing fault tolerance are implemented as a *separate VHDL package* of register transfer level descriptions, and they are included into the result of high level synthesis automatically, or with minimal user intervention (depending on the synthesis tool actually used).

2 Error Detection on Behavioural Level

On the behavioural level of system modelling, the most common data types are *numbers* (usually integers, sometimes floating point numbers). Both the correctness of the result produced by the system and the control flow of the algorithms themselves depend severely on the integrity of the numeric values. However, numbers are transformed to multi-bit data lines at lower levels of abstraction, so usual physical level faults (stuck-at, bridging etc.) are represented as *corrupted values* at the behavioural level. Therefore the numeric values of the high level model need to be protected against unintentional value changes.

There are numerous methods for this purpose [4]. The application of *residue codes* [3] has attractive advantages: relatively low calculation requirements (resulting in low hardware and/or time overhead in the final implementation) and good error detection capability. In our experimental system, the simple *mod 3* residue code was implemented for integer numbers.

Applying the mod 3 residue code involves the extension of all integer values by a separate residue value, that contains its remainder modulo 3. Additionally, the consistency between the numeric value and the residue must be held during the operations on the integers. This task can be solved in a very elegant way in VHDL, due to certain syntactic properties of the language. As all the VHDL operators are treated basically as functions, a feature called *operator overloading* is provided. The working of the standard VHDL operators can also be redefined by the user, simply by writing the appropriate functions that take the operands as arguments and return the result of the operator. This feature is demonstrated on Fig. 1:

```
package integer_rescode is

 subtype res_code is integer range 0 to 3;
constant invalid_rc: res_code := 3;

 type integer_rc is record
   value: integer;
   rescode: res_code;
 end record;

 function “+” (l, r: integer_rc) return integer_rc;
 function “-” (l, r: integer_rc) return integer_rc;
...
end integer_res_code;

package body integer_rescode is

 function rescode (arg: integer) return res_code is
   variable result: res_code;
 begin
   result := arg mod 3;
   return result;
 end res_code;

 function “+” (l, r: integer_rc) return integer_rc is
   variable result: integer_rc;
 begin
   result.value := l.value + r.value;
   if (rescode(l) /= l.rescode)
    or (rescode(r) /= r.rescode) then
     result.rescode := invalid_rc;
   else
     result.rescode := result.value;
   end if;
   return result;
 end “+”;
...
end integer_rescode;
```

Fig. 1: Fragment of the mod 3 residue code in VHDL on high level

In the above example, a data type `integer_rc` is declared as a record: its members are `value` (that carries the numeric value of an integer) and `rescode` (that carries the residue code). Then the function implementing the standard operators “+”, “-”, etc. are defined on `integer_rc` type operands in a way that they return the result of the same operator on integers in `value`, and the residue code of this result in `rescode`.

The constant *invalid_rc* is defined for error detection purposes: as it represents an illegal mod 3 remainder, `rescode` is set to *invalid_rc* if the residue code of any operands was incorrect. The illegal value of `rescode` is propagated in all subsequent operations, thus ensuring that as long as the effect of any value change caused by a fault is detectable by mod 3 residue code, it will be observable on the affected output(s) of the system as well.

This way, embedding residue code-based fault tolerance into behavioural level VHDL circuit descriptions is most simple: it consists of including an alternate VHDL numerical package (that contains the declaration of the residue code protected `integer_rc` type and the definitions of the standard VHDL operators on this type), and replacing all `integer` type definitions with `integer_rc`. This replacement may impose some difficulties, as `integer_rc`, being a record type, has a different set of pre-defined attributes than the `integer` type; however, these attributes are rarely used in behavioural level descriptions.

2.1. Transformation into Register Transfer Level

Modern digital circuit design systems usually offer automatic structural silicon layout generation from low level circuit descriptions. Therefore behavioural level models must be transformed into a lower level. For our experiments, the AMICAL [1] system was chosen, due to its free availability and positive past experiences [2].

The embedded fault tolerant features of the behavioural models must also be transformed. As most high level synthesis systems transform behavioural level operations into pre-defined low level functional units (FUs), this transformation necessitates the re-implementation or modification of FUs representing the overloaded operators. Obviously, the high level synthesis tool must accept the modified `integer_rc` type and convert it appropriately to a lower level data type. As this data type is usually the standard `bit_vector` or `std_logic_vector` types with a globally defined size (which is a user-supplied design parameter), it can be easily extended with the two extra data lines needed by the mod 3 residue code. In the case of AMICAL, a set of FUs is supplied in low level VHDL description format, so their modification is just as simple as the modification of the behavioural model. The register transfer level equivalent of the `integer_rescode` package was implemented as a straightforward modification of the IEEE_1164 `std_logic` numeric operation package.

2.2. The GCD example circuit

The sample circuit used for demonstration of the embedding of fault tolerance was the GCD (Greatest Common Divisor) example supplied with AMICAL. The circuit realizes the traditional Euclidean algorithm for calculating the greatest common divisor of two integer numbers as a sequential digital circuit. The behavioural level VHDL description is shown below. Comments starting with `--!!` denote the changes necessary for embedding fault tolerance.

```
package synchro is
 function rising_edge(signal horloge:bit) return boolean;
 procedure delay(temps:time);
end synchro;
package body synchro is
 function rising_edge(signal horloge:bit) return boolean is
 begin
 return(horloge'event and horloge='1' and horloge'last_value='0');
 end rising_edge;
 procedure delay(temps:time) is
 begin
   wait for temps;
 end delay;
end synchro;
use work.synchro.all;
use work.integer_rc.all;                --!! new line inserted

entity gcd is
 port(  clk : in bit;
        reset : in bit;
        start : in bit;
        din : in bit;
        xi, yi : in integer_rc;         --!! instead of integer
        dout : out bit;
        ou : out integer_rc);           --!! instead of integer
end gcd;

architecture behavior of gcd is
begin
 process
   variable x,y: integer_rc;            --!! instead of integer
 begin
   wait until (start = '1' and rising_edge(clk));
   dout <= '0';
   calculation : loop
     wait until (din = '1' and rising_edge(clk));
     x := xi;
     y := yi;
     while (x /= y ) loop
       if (x < y) then y := y - x;
       else x := x - y;
     end if;
   end loop;
   delay(250 ns);
   ou <= x;
   dout <= '1';
   wait until (din = '0' and rising_edge(clk));
   dout <= '0';
 end loop;
 end process;
end behavior;
```

3 Conclusions

During the experiments, the embedding of mod 3 residue code protected data values into the behavioural level circuit description has been successfully solved. The modified description was still accepted both by the VHDL analyzer subsystem of the AMICAL synthesis tool and

the ModelTech V-System VHDL simulator that was used for validation. The functional equivalence of the original and the modified description was ensured due to the simple and systematic changes.

After porting the mod 3 residue code implementation into register transfer level, and modifying the AMICAL RTL functional unit library appropriately, AMICAL was able to produce an RT level VHDL description that proved to be functionally equivalent with the behavioural level description. Minimal user intervention was required to circumvent certain problems with the AMICAL version used (it involved trivial modifications of the output VHDL files).

Attempts were made to generate a gate level VHDL circuit description from the RT level description. According to the targeted implementation architecture, a Xilinx FPGA development tool was used to convert the RT level VHDL model into a Xilinx FPGA wiring. The overhead in the number of equivalent gate inputs was 48% with 8 bit wide integers and 42% with 32 bit wide integers. These values are significantly higher than the expected values according to the literature [5]. The actual overhead value, however, is severely affected by the following factors:

- the sub-optimal implementation of the residue code checker (it was necessary to re-implement it almost completely due to the different VHDL subset used by the FPGA design environment);
- the sample circuit was mostly data dominant, with a very simple control sequence;
- the FPGA synthesis tool generated the checker as a combinational circuit, optimized for speed. A slightly slower, sequential implementation would have resulted in a smaller hardware overhead;
- the target architecture applied in AMICAL resulted in the gate level implementation of the residue code checker in each functional unit. If the residue code checker is implemented as a separate unit, and is shared between the FUs of the circuit, the overhead with respect to the whole circuit is obviously smaller.

The hardware overhead, based on these assumptions, can be estimated as about a half magnitude smaller with a better fitting low level synthesis technique.

References

[1] JERRAYA, A. A. et al: AMICAL: Interactive Architectural Synthesis Based on VHDL. Internal Report, INPG/TIMA System Level Synthesis Group, Grenoble, April 1994.

[2] SALLAY, B., PETRI, A., TILLY, K., PATARICZA, A., SZIRAY, J.: High Level Test Generation for VHDL Circuits. Proceedings of the IEEE European Test Workshop ‘96, June 1996, Montpellier, pp. 201-205.

[3] SAYERS, I. L., KINNIMENT, D. J.: Low-cost residue codes and their application to self-checking VLSI systems. IEEE Proc. E, 132(4): pp. 197-202, 1985.

[4] THOMA, Y.: Coding techniques in fault-tolerant, self-checking and fail-save circuits. In: D. K. Pradhan (ed.): Fault Tolerant Computing, Theory and Techniques. Vol. 1, pp. 336-416. Prentice-Hall, 1986.

[5] WATTERSON, J. W., HALLENBECK, J. J.: Modulo 3 residue checker: New results on performance and cost. IEEE Trans. on Comp., C-37, pp. 608-612, May 1988.