Automatic Modifications of High Level VHDL Descriptions for Fault Detection or Tolerance

R. Leveugle
TIMA Laboratory
46, Avenue Félix Viallet - 38031 Grenoble Cedex – FRANCE - E-mail: Regis.Leveugle@imag.fr

Abstract

The need for integrated mechanisms providing on-line error detection or fault tolerance is becoming a major concern due to the increasing sensitivity of the circuits to their environment. This paper reports on a tool automating the implementation of such mechanisms by modifying high-level VHDL descriptions. The modifications are compatible with industrial design flows based on commercial synthesis and simulation tools. The results demonstrate the feasibility and the efficiency of the approach.

1. Introduction

Mechanisms aiming at on-line detection or tolerance of faults have been integrated for years in circuits dedicated to critical applications such as spacecrafts, avionics or more recently airbags for cars. Tools were proposed to automate some implementations (e. g. [1, 2]) but were not widely used. Commercially available solutions, such as the specific state assignment option provided in the Autologic synthesis tool [3], were also not flexible enough to attract a large number of users. In consequence, the circuits implemented with on-line detection or tolerance mechanisms remain mostly implemented manually [4] or with proprietary tools [5].

The current evolution of the CMOS technologies increases the sensitivity of the circuits to their environment and consequently the probability of transient faults. Phenomena reported for years in the space environment, such as bit-flips due to the impact of particles (called Single Event Upsets, or SEUs), are becoming now observable even at the sea level, especially due to the impact of atmospheric neutrons. The need for integrated on-line detection or tolerance is therefore pervading everyday life applications such as mobile telecommunications or computing. However, the noticeable development time required to manually insert fault detection or tolerance is not compatible with the time-to-market pressure in these application areas. The need for flexible tools allowing the designer to increase the dependability of a circuit is therefore growing. Such a tool must be completely compatible with industrial design flows based on commercial synthesis and simulation tools, and must provide various trade-offs between dependability characteristics and induced overheads, so that the designer can tune the mechanisms implemented in the different blocks of a circuit according to the constraints on each block. Such a tool could also be used to harden reusable components (IPs).

The goal of the study reported here is to evaluate the feasibility and interest of an approach based on automatic modifications of high level specifications. The major advantage with respect to modifications performed at the gate level would be to allow an early insertion of the mechanisms and an early validation of the circuit or system-level dependability. The drawback is the difficulty to precisely control the structure of the implemented mechanisms and the potential increase of the overheads.

A flow based on VHDL and commercial CAD tools was reported in [6] to implement totally self-checking (TSC) circuits. The authors focused on the state assignment and signal encoding aspects. The approach studied here goes one step further, as it consists in generating a modified high-level description of the circuit, providing the required dependability characteristics and used in place of the initial description in any design flow.

A similar approach had been recently proposed to implement massively redundant architectures such as Triple Modular Redundancy (TMR) or error detecting/correcting codes to locally protect registers [7, 8]. The architectures considered in this paper are more specific and impose more constraints on both the high-level modifications and the global design flow, including the control of the synthesis process. These architectures, when implemented at the gate level during the logic synthesis process, proved to provide interesting trade-offs [2]. They have therefore been revisited to evaluate the feasibility of their implementation earlier in the design flow. In the current implementation of the tool, the modifications are performed on circuit blocks described either as a Finite State Machine (FSM), or as a Register-Transfer Level (RTL) or behavioral control flowchart.

The targeted architectures are briefly presented in section 2. Details on the architectures can be found in [2, 9]. This paper will rather focus on the high-level modifications, summarized in section 3, and on the discussion of extensive experimental results shown in section 4 for various implementation targets.

2. Targeted architectures

Figure 1 illustrates the selected architecture for fault tolerance, named SID (Single Independent Decoder) [2]. This architecture aims at tolerating single bit-flips in a state register as well as single or multiple faults in the next state computation logic. The combinatorial logic computing the next state is therefore implemented with independent logic cones, that requires a precise control of the synthesis process. The states are encoded using a Hamming single error correcting (SEC) code and the fault tolerance is achieved by a decoding logic connected onto the outputs of the state register. This decoder corrects an erroneous bit in the current state code, before this code is used to compute the outputs and the next-state code.

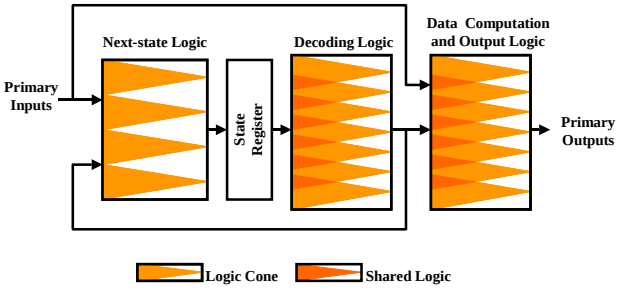

Figure 1: The SID architecture.

The architectures considered for on-line testing aim at detecting erroneous transitions and are based on control flow checking (CFC) using signature analysis. Figure 2 illustrates, on the example of a FSM implementation, one of these architectures, called CFC-AJS because explicit signature adjustments (AJS) are used to ensure an easy-to-verify invariant property on the signature [9]. This architecture requires to add a monitor block to the initial circuit and to generate combinatorial logic to compute the adjustments. This implementation is not limited to the detection of single bit-flips in the state register ; multiple bit-flips due for example to a particle impact in the combinatorial logic can also be detected.

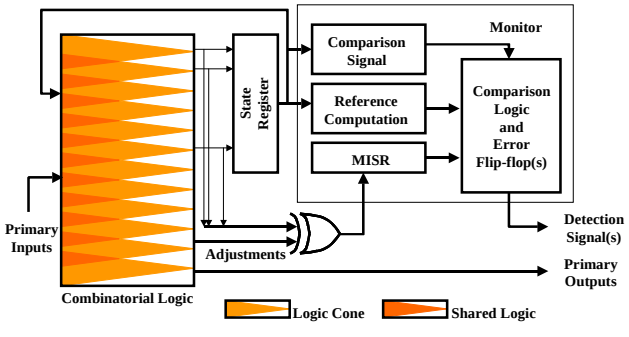

Figure 2: The CFC-AJS architecture.

3. Automatic high-level modifications

There is a strong relationship between the modifications in the high-level descriptions and the synthesis process, since it must be ensured that the synthesis tool will not partially or totally suppress the redundancy inserted in the description. Also, some information may be necessary from the synthesis (e.g. the state assignment result) and some control may be necessary onto the synthesis process (e.g. to generate independent logic cones in the SID architecture). The high-level modifications, illustrated in this paper by modifications in VHDL descriptions, must therefore take into account the constraints of the global implementation flow. The proposed modifications have been defined so that they are independent from the synthesis tool ; any commercial tool can thus be selected by the designer.

One of the basic templates for the initial circuit descriptions is illustrated by the example in Figure 3. The modification flow will then be illustrated on this example. The states are defined in Figure 3 using symbolic names and an enumerated type. This is the most efficient description, since it allows the synthesis tool to optimize the state assignment. However, the proposed approach can also be applied to descriptions in which the state assignment is directly specified by the designer, using for example integer constants for the states. The circuit can be of either Moore or Mealy type. Also, the tool we have developed is not limited to finite state machine descriptions, such as the one used as example ; control flowcharts including operations on data can also be automatically modified. Finally, the VHDL architecture can be described using any number of processes.

```
entity FSM is
port (
  clk, reset : in std_logic;
  in1, in2   : in std_logic;
  out1, out2 : out std_logic
);
end FSM;

architecture archi of FSM is
  type STATE_TYPE is (state_0, state_1,
                      state_2, state_3, state_4);
  signal state, next_state : STATE_TYPE;
begin
  process (clk, reset)
  begin
    if (clk='1' and clk'event) then
      if (reset='0') then
        state <= state_0;
      else
        state <= next_state;
      end if;
    end if;
  end process;

process (state, in1, in2)
  begin
    case state is
      when state_0 =>
        if (in1='0') then
          next_state <= state_0;
        else
          next_state <= state_2;
        end if;
      when state_1 =>
      …
    end case;
  end process;

  process (state)
  begin
    case state is
      when state_0 =>
        out1 <= '0';
      when state_1 =>
      …
    end case;
  end process;
end arch;
```

Figure 3: Simple initial description example.

3.1. The SID architecture

The VHDL modifications for the SID architecture must be particularly careful to avoid unwanted shared logic during synthesis. The different phases are as follows.

The initial description is first loaded into the synthesis tool and a state assignment on a minimal number of bits is performed. Of course, this phase is necessary only if the states are specified as an enumerated type, without any user encoding. Any user-defined state assignment can also be used. Once the state assignment is known, the state codes are completed by adding the Hamming check bits. A VHDL package is then created, including the definition of all the redundant and irredundant state codes. The result is illustrated in Figure 4 for the example in Figure 3.

The logic computing the next state must be implemented with independent logic cones ; this is achieved by making a hierarchical decomposition of the circuit, with separated blocks for each state register input. This approach avoids a complex control of the synthesis process, which may furthermore have to be adapted to each synthesis tool, depending on the available options.

The initial description is therefore analyzed to separate the next state computations from the other statements, related to data or output computations. The next state computations are further decomposed into computations of next state code bits, taking into account the Hamming state encoding obtained in the previous phase. The modified VHDL description is then generated hierarchically with a sub-block (or VHDL component) for data and output computations, a sub-block corresponding to the Hamming decoding logic, a sub-block for the state register and a sub-block for the computation of the next-state bits (each bit being associated to a different VHDL architecture). The description of some blocks is illustrated in Figure 5. In that way, the description can be correctly synthesized with any tool, without requiring any particular option but respecting the hierarchy during the synthesis, i.e. synthesizing and optimizing separately each block. The different blocks may also be synthesized with different options, depending on the designer goals. For example, the critical decoding logic can be optimized for area while some other blocks can be optimized for speed.

```
package SID_state is
  subtype STATE_TYPE is std_logic_vector (2 downto 0);
  subtype enc_STATE_TYPE is std_logic_vector (5 downto 0);
  constant state_0: STATE_TYPE := "000";
  constant enc_state_0 : enc_STATE_TYPE := "000000";
…
end SID_state;
```

Figure 4: Package created for the architecture SID.

```
entity state_register is
port (
  clk : in STD_LOGIC;
  reset : in STD_LOGIC;
  next_state : in enc_STATE_TYPE;
  cur_state : out enc_STATE_TYPE
);
end state_register;
architecture arch of state_register is
…
```

```
entity next_state_logic is
…
architecture arch0 of next_state_logic is
…
   when state_0 =>
        if (in1='0') then
          next_state_b<=enc_state_0(0);
        else
          next_state_b<=enc_state_2(0);
        end if;
…
```

Figure 5: Partial description for the architecture SID, illustrating the extraction of the state register and the bit-level computation of the next state code.

3.2. The CFC-AJS architecture

The generation of the modified circuit description for the CFC-AJS architecture is also performed in several phases:

In the case of the CFC-AJS architecture, the first and most complex task is the analysis of the control flow graph to determine the locations of the adjustments. The algorithm used is the one previously implemented in the ASYL-SdF tool [2] and it will not be detailed here since it is not directly related to the VHDL modifications.

A state assignment is then performed on the initial description, using the synthesis tool, if the designer has not pre-defined any encoding. As in the SID case, a VHDL package is then created, defining the state names and codes. This package is used in the description of all the blocks of the architecture and is illustrated in Figure 6.

Once the state assignment is known, the signatures associated with each state and the adjustment values are computed with respect to a MISR either specified by the user or automatically selected with a primitive divisor polynomial. The initial circuit description is then modified to insert the adjustment computations, as illustrated in Figure 7. Let us notice on this example that a "when others" clause has to be added in the case statements related to the current state. The reason is that after state assignment, some binary codes are in general unused. These codes can be used as Boolean "don't cares" to minimize the logic, as illustrated in Figure 7. It is also possible to use this clause to further harden the circuit, by generating an error signal allowing the user to detect immediately an illegal state code. A RTL description of the monitor is then generated. Finally, the complete hierarchical description of the circuit with on-line test is obtained by connecting the modified circuit description and the monitor description.

```
package CFC_state is
  subtype STATE_TYPE is std_logic_vector(2 downto 0);
  constant state_0 : STATE_TYPE := "000";
  constant state_1 : STATE_TYPE := "001";
  …
end CFC_state;
package body CFC_state is
end CFC_state;
```

Figure 6: Package created for the architecture CFC-AJS.

```
process (state, in1, in2)
begin
  case state is
    when state_0 =>
      if (in1='0') then
        next_state <= state_0;
        adjust <= "000";
      else
        next_state <= state_2;
        adjust <= "011";
      end if;
```

```
        when state_1 =>
…
      when others =>
        next_state <= (others=>'-');
        adjust <= (others=>'-');
  end case;
end process;
```

Figure 7: Partial description for the architecture CFC-AJS, illustrating the insertion of the adjustment computations and the addition of the "when others" clause in the case statement.

4. Experimental results

It was proved that the architectures CFC-AJS and SID can lead to lower overheads (in terms of area and power consumption) than more classical architectures and also to very interesting dependability characteristics (in terms of fault coverage, mission time or mean time to first failure). These results were obtained when inserting the extra logic during the synthesis and carefully optimizing each added element. The drawback of this approach was to require a specific synthesis tool, namely the Asyl-SdF tool [2]. The experiments reported here aimed therefore at comparing the overheads previously obtained with Asyl-SdF and those obtained with a commercial synthesis tool after the proposed high-level modifications.

Using the tool presented in section 3, the approach has been extensively evaluated on examples from the classical benchmark suits and on various other circuit examples coming from university and industry. All the results reported for the proposed approach have been obtained with the synthesis tool Leonardo from Mentor Graphics, using the "binary" state assignment option that generates state codes on a minimal number of bits. The results shown in tables 1 and 2 correspond to implementations with the library cub in the CMOS 0.6 micron technology from AMS. The results of the previous work had been obtained with the specific synthesis tool Asyl-SdF and the CMOS 1.2 micron standard cell library from VLSI Technology [2]. For the approach proposed in this paper, the figures have been obtained after synthesis, using estimated interconnection characteristics. The previous results were obtained after placement and routing in the Compass environment. These comparisons may therefore be refined, taking into account for the two approaches the actual interconnection characteristics after placement and routing. However, the experiments made on some circuit examples showed that the information available after synthesis are either pessimistic or quite close to the results obtained after placement and routing. As examples, the area overhead after placement and routing for the benchmarks called "jay" and "dstate" and the architecture SID are 50% and 68% respectively, instead of 79.01% and 103.33% estimated after synthesis. The main conclusions of the study would therefore be similar. Furthermore, a strict comparison of the overheads cannot be meaningful, since the technologies, tools and libraries are different. We can only compare the order of magnitude of the overheads.

The results obtained clearly show that the developed methodology is, on an average, as efficient as the approach in [2] for the SID architecture. 66 benchmarks have been modified and synthesized. 30 out of the 61 benchmarks previously synthesized using Asyl-SdF had higher area overheads using high-level modifications. However, for the 66 examples, the average area overhead is equal to 136% with the proposed approach and a synthesis optimizing the circuit area, to be compared with a 149% average overhead in the previous approach. When performing a synthesis optimizing the critical path of the circuit, the average area overhead is 150% with the proposed approach. In terms of critical path, the average overhead estimated by the synthesis tool is 55% (61% when the synthesis aims at optimizing the delay), to be compared with a 198% overhead (after place and route) in the previous approach.

Table 1: Comparison of overheads for the SID architecture, after a synthesis optimizing the area.

Overhead	Area		Critical path	
Benchmark	Current work	Previous work	Current work	Previous work
dstate	103.33%	145.34%	54.64%	172.83%
imec1	67.11%	47.58%	50.00%	77.15%
jay	79.01%	94.98%	24.17%	152.67%
protocole	115.52%	130.76%	40.64%	334.48%
s1	115.44%	95.29%	71.04%	200.00%
s1488	45.61%	69.06%	53.81%	131.15%

Table 2: Comparison of overheads for the CFC-AJS architecture, after a synthesis optimizing the area.

Benchmark	Area overhead (Current work)	Area overhead (Previous work)
imec1	33.22%	31.33%
jay	58.56%	131.98%
protocole	81.03%	138.19%
s1	63.97%	127.85%

Table 3: Average resource overheads (SID).

Technology	cub	Virtex	A40
Optimized area	135.53%	96.98%	117.40%
Optimized delay	149.79%	100.76%	106.48%

Table 4: Average critical path overheads (SID).

Technology	cub	Virtex	A40
Optimized area	55.05%	35.84%	84.88%
Optimized delay	61.00%	33.66%	109.67%

Table 5: Average resource overheads (CFC-AJS).

Technology	cub	Virtex	A40
Optimized area	115.38%	119.47%	113.62%
Optimized delay	108.00%	123.91%	116.83%

Table 6: Average critical path overheads (CFC-AJS).

Technology	cub	Virtex	A40
Optimized area	48.37%	14.27%	64.32%
Optimized delay	55.36%	10.90%	71.45%

The results for the CFC-AJS architecture confirm the efficiency of the proposed approach. The average area overhead on the 66 benchmarks is 115.38% when the synthesis optimizes the area and 108% when optimizing the critical path (but the overhead is for example as low as 7.82% for the benchmark imec7). This can be compared with an average overhead equal to 130.82% (after placement and routing) in the previous approach. The average critical path overhead is respectively 48.37% and 55.36% (only 0.1% for imec7) for the proposed approach.

The characteristics of some of the benchmarks and the related overheads can be found in [9]. Tables 3 to 6 summarize the average overheads obtained for three technologies, including two programmable devices, on the 66 modified benchmarks. The synthesis process has not been carefully tuned to minimize these overheads, which could therefore be further reduced. The interesting point is that the resource overheads (in terms of area for the standard cell implementation, number of CLBs for the Xilinx implementation and number of basic cells for the Actel implementation) have the same order of magnitudes for the different technologies. The critical path overheads tend to be smaller for implementations on the Virtex device. However, very careful placement and routing would be required to keep this advantage. Let us also recall here that the presented architectures are not intended for very high speed blocks, but for blocks which are not on the circuit-level critical path and which have to be optimized for the best trade-off between coverage and the various overheads. The negative critical path overheads obtained in several cases demonstrate furthermore that the results presented here are very dependent on the optimization heuristics of the synthesis tool ; slightly different results may be obtained using another synthesis tool and/or specific optimization options.

5. Conclusion

The feasibility of dependability increase by automatic modifications of high-level VHDL descriptions has been demonstrated and a tool has been developed. The tool can be used to modify circuit blocks described either as finite state machines or RTL/behavioral control flowcharts. Results have been obtained on a large set of benchmarks coming from various sources and demonstrate the interest of the proposed approach to implement circuits with the SID and CFC-AJS architectures.

This is the first attempt at automating the implementation of such specific dependable architectures by modifying high-level descriptions. Results are interesting since they are, on an average, very close to those obtained when the modifications are performed by a specific synthesis tool as reported in [2], although in that later case the insertion of the additional elements can be controlled more precisely and therefore can be more optimized. The slight loss in optimization observed in some cases is compensated by the advantageous compatibility with industrial design flows based on any commercial synthesis and simulation tools. Furthermore, modifying the high-level description allows the designer to evaluate earlier in the design process the global dependability of his circuit. Last but not least, lower overheads have been achieved for a noticeable number of benchmarks, compared with the previous approach, and similar results have been obtained for several implementation technologies.

This study provides motivations to automate other types of modifications, leading to different trade-offs between dependability characteristics and implementation overheads, in order to provide the designer with a complete toolbox adapted to various design constraints and various types of blocks.

Acknowledgment

The author wants to thank Rémi Cercueil for his work on the methodology and the development of the tool automating the modifications.

References

[1] I. Alzaher-Noufal, M. Nicolaidis, "A CAD framework for generating self-checking multipliers based on residue codes", Design, Automation and Test in Europe Conference (DATE), March 9-12, 1999, pp. 122-129

[2] R. Rochet, R. Leveugle, G. Saucier, "ASYL-SdF: a synthesis tool for dependability in controllers", IEICE Transactions on Information and Systems, Vol. E79-D, no. 10, October 1996, pp. 1382-1388

[3] Autologic User Manual, Mentor Graphics

[4] I. Gonzalez, L. Berrojo, "Supporting fault tolerance in an industrial environment: the AMATISTA approach", 7th IEEE Int. On-Line Testing workshop, July 2001, pp. 178-183

[5] E. Böhl, R. Stephan, W. Glauert, "The architecture of the fail-stop controller AE11",3rd IEEE Int. On-Line Testing workshop, July 1997, pp. 47-52

[6] C. Bolchini, R. Montandon, F. Salice, D. Sciuto, "Design of VHDL-based totally self-checking finite-state machine and data-path descriptions", IEEE trans. on VLSI Systems, vol. 8, no. 1, February 2000, pp. 98-103

[7] F. Vargas, A. Amory, "Recent improvements on the specification of transient-fault tolerant VHDL descriptions: a case-study for area overhead analysis", SBCCI, September 2000, pp. 249-254

[8] L. Entrena, C. Lopez, E. Olias, "Automatic insertion of fault-tolerant structures at the RT level", 7th IEEE Int. On-Line Testing workshop, July 2001, pp. 48-50

[9] R. Leveugle, R. Cercueil, "High-level modifications of VHDL descriptions for on-line test or fault tolerance", The IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems, October 2001