Logic-Level Analysis of High-Level Faults *

Franco Fummi
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona, Italy
franco.fummi@univr.it

Graziano Pravadelli
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona, Italy
pravadelli@sci.univr.it

ABSTRACT

Many high-level fault models have been proposed in the past to perform verification at functional level, however high-level automatic test pattern generators (ATPGs) are still in a prototyping phase, while very efficient logic-level ATPGs are available. This paper proposes a strategy to map high-level faults into logic-level faults. Thus, functional verification, based on a high-level fault model, can be performed by exploiting the capability of state of the art logic-level ATPGs.

Categories and Subject Descriptors: B.8.1 [Hardware]: Testing

General Terms: Verification

Keywords: Fault models, Functional Verification

1. INTRODUCTION

More and more functional verification [5, 7] is adopted to detect design errors exploiting coverage metrics [3, 11] or high-level fault models. In particular, some high-level fault models [4, 2] have been recently proposed to include the characteristics of traditional coverage metrics [9] (e.g., statement, branch, condition coverage) and logic-level fault models [1]. Such a confluence of coverage metrics and fault models mainly depends on the consideration that hard to detect or untestable high-level faults identify corner cases, which can represent design errors [4]. The analysis of the nature of high-level faults allows an effective verification of the expected and unexpected behavior of the design, particularly when faults are directly injected into HDL code which is very familiar to the designer. Whereas, designers would have a very hard work to investigate about the nature of untestable logic-level faults.

In the literature there are some papers [11, 2, 4] that try to correlate high-level with logic-level faults. These works show that test sequences generated to cover high-level faults are also good test cases for detecting logic-level faults. However, high-level automatic test pattern generators are still in a prototyping phase while very efficient logic-level ATPGs have been developed in the past and ported into commercial tools [8, 13]. This paper proposes a strategy to efficiently perform functional verification, based on high-level faults, by exploiting the potentialities of state of the art logic-level ATPGs. To accomplish the goal, four different approaches for mapping high-level faults into logic-level faults are investigated and compared.

The proposed solution for mapping high-level faults into logic-level faults allows, furthermore, to face the interesting verification problem of providing an efficient and accurate mapping technique to establish correspondence between behavioral/RT level signals and logic-level nets. In some cases designers are interested in identifying which portion of the logic-level implementation corresponds to a slice of the high-level description. Possible motivations can be: reuse of parts of the implementation, analysis of a design area, etc. Formal equivalence checkers are used to this purpose, however they require a lot of resources in terms of time and memory. In [10] an interesting fault simulation-based approach is proposed. The authors exploit the fact that circuit diagnosis provides an effective method for identifying a fault location in the circuit. However, the work presents some limitations which are avoided by our mapping strategy.

The paper is organized as follows: Section 2 describes the adopted high-level fault model. Section 3 proposes and compares four different approaches for mapping high-level faults into logic-level faults. Section 4 describes how one of the proposed fault mapping strategies allows to identify the correspondence between RT/behavioral signals and logic-level nets. Finally, experimental results are reported in Section 5.

2. HIGH-LEVEL FAULT MODEL

The high-level fault model adopted in the paper is the *bit coverage*, which simulates under the single fault assumption:
Bit failures. Each occurrence of variables, constants, signals or ports is considered as a vector of bits. Each bit can be stuck-at zero or stuck-at one.
Condition failures. Each condition can be stuck-at true or stuck-at false, thus removing some execution paths in the faulty representation.

Bit coverage is chosen since it is related to design errors [4, 7] and it unifies into a single metrics the well known metrics concerning statements, branches and conditions coverage. In addition, paths needed to activate and propagate faults from inputs to outputs of the DUV are also covered. Finally, bit coverage shows a high correlation between stuck-at faults at different levels of abstraction [4].

Bit coverage faults are automatically injected into an RTL DUV by using AMLETO [5]. It allows to inject faults in VHDL code as well as in SystemC code. For simplicity, in the sequel of this paper we explicitly refer to VHDL examples, but the methodology is actually independent from the

*Research activity partially supported by the European Community IST-2001-34607 project: SYMBAD

GLSVLSI'04, April 26–28, 2004, Boston, Massachusetts, USA.

adopted language. Fault injection is performed by inserting saboteurs into the DUV. Every occurrence of signals, variables, constants and conditions of the high-level description is replaced by an opportune bit coverage saboteur. We define a saboteur for every language type, i.e., `bit`, `integer`, `standard_logic`, `boolean`, etc. They are functions which can supply the correct or the faulty value of the target object depending on the value of a control signal. Faults are enumerated starting from 0, and a `bit_vector`-type port, named `fault`, is added to the DUV. The number of elements of the `fault` port equals the number of faults. The `fault` port drives the control signals of saboteurs. To activate the fault number i, `fault[i]` is fixed to '1'. Figure 1 shows the VHDL saboteur function for the `bit` data type. Saboteurs for other data types are defined by converting the target object to a sequence of bit and referring to the bit case. In this way, changing the definition of the saboteur for bit, the behavior of saboteurs for the other data types changes accordingly. The first parameter of the saboteur function, (`object`), is the target of the fault, while the second (`fault_code`) is the value of the `fault` port. Parameters `start_s0-1` and `end_s0-1` show the valid range for `fault_code` to activate the stuck-at 0-1 on the target object (`end_s0-1` are useless for one-sized data types, e.g., `bit` and `boolean`, since their value equals `start_s0-1`, but they are necessary for multi-sized data types).

The fault injection process generates a unique faulty description of the DUV that includes all bit coverage faults. Figure 2 shows an example of fault-free and faulty VHDL descriptions by using bit coverage saboteurs. It illustrates how the faults are recursively inserted in complex statements as an `if-then-else` statement. For example, to activate the fault stuck-at 0 on the third bit of the integer signal `rmax`, the signal `fault[56]` must be set to '1', since the range for faults stuck-at 0 on `rmax` is from 54 to 61. On the other hand, to activate the fault stuck-at true on the `if-then-else` condition the signal `fault[73]` must be set to '1'.

After fault injection, the fault list is optimized by removing equivalent faults and faults that are untestable without being symptom of design errors (e.g., stuck-at 0 on constants whose value is '0') [6]. Such faults *have not* a corresponding logic-level stuck-at fault, since the synthesis removes the parts of the functional description, where they are injected in, to optimize the design. Remained faults can be all mapped.

3. FAULT MAPPING

In past years, some very efficient logic level ATPGs have been developed. A method to map high-level faults into logic-level stuck-at faults is necessary to exploit the potentialities of these ATPGs for detecting high-level bit coverage faults. In the next subsections four different approaches are proposed and compared.

3.1 Trivial Method

Given the logic-level network of the DUV and the corresponding fault list, the easier way to map a bit coverage fault into a logic-level stuck-at fault consists of the following naive approach:

```
for each bit coverage fault i do
    for each element of fault port do fault[j] := '0' if j ≠ i
    set up logic-level ATPG
    add Stuck-at 1 on fault[i]
    run logic-level ATPG
```

```
function inject_fault_bit(object: bit; fault_code: bit_vector;
  start_s0: integer; end_s0: integer; start_s1: integer;
  end_s1: integer) return bit is
variable res: bit; begin
  if (object='0') then res := fault_code(start_s1);
  else res := NOT(fault_code(start_s0));
  end if;
  return res;
end;
```

Figure 1: Saboteur VHDL function for bit operands.

```
if (data_in > rmax) then ack <= '1';

if (inject_fault_bool(inject_fault_integer(data_in,fault,38,45,46,53)
   > inject_fault_integer(rmax,fault,54,61,62,69),fault,70,70,71,71))
   then ack <= inject_fault_bit('1', fault,72,72,73,73);
```

Figure 2: Fault-free and faulty VHDL code.

The saboteur functions activate or deactivate the related faults accordingly to the value of the `fault` port. During each ATPG session, all the elements but one (element `i`) of the `fault` port are fixed to '0', i.e., they are deactivated. Then, the ATPG is forced to detect the stuck-at 1 on the line `fault[i]`. Thus, for its nature, the ATPG compares two instances of the design: one with a '0' on `fault[i]` and the other with a '1' on the same position. In the first case all bit coverage faults are deactivated, thus the design is fault-free. Instead, in the second case, the bit coverage fault `i` is activated. In this way, if the ATPG detects the logic-level stuck-at 1 on `fault[i]`, actually it detects the high-level bit coverage fault `i`.

Unfortunately, this method is almost infeasible, since the ATPG setup session is very time consuming. For each setup session the ATPG needs to propagate the value of the `fault` port elements fixed to '0' to minimize the circuit logic before starting test pattern generation. To avoid waste of time, a unique setup phase is needed for all bit coverage faults. In the next three methods the goal is obtained by removing the `fault` port and the related saboteurs logic from the synthesized faulty DUV.

3.2 Topological Method

Given a selected technology library, the synthesis process maps each saboteur function call of the DUV into a network of logic gates. By analyzing the synthesized design it is possible to identify the topology of the saboteur corresponding to a bit coverage fault. Thus, the following algorithm allows to use a logic-level ATPG to detect bit coverage faults:

```
for each bit coverage fault i do
    identify the corresponding logic topology according
       to the selected technology library
    remove the topology and directly connect its input
       line (object) and its output line (result)
    if i is Stuck-at 0 or Stuck-at false
       add Stuck-at 0 on the corresponding result line
    else if i is Stuck-at 1 or Stuck-at true
       add Stuck-at 1 on the corresponding result line
set up and run logic-level ATPG
```

In this way only one ATPG set up phase is necessary. Moreover, the time it requires is exactly the same needed for the fault-free design since the saboteur logic and the `fault` port have been removed. The main problem of this approach is represented by the identification of the saboteur topology. For example, by using a simple technology library composed of a NOT, a 2-inputs AND, a 2-inputs OR, a LATCH and a FLIP FLOP, in the majority of cases the

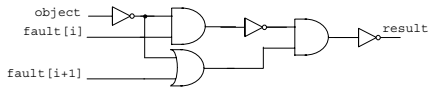

Figure 3: Topology of the bit saboteur.

```
if(inject_fault_bool(inject_fault_bit(line1,fault,890,890,891,891)
  = inject_fault_bit('1',fault,892,892,893,893),...))
```

Figure 4: VHDL faulty code with unrecognizable saboteur topology.

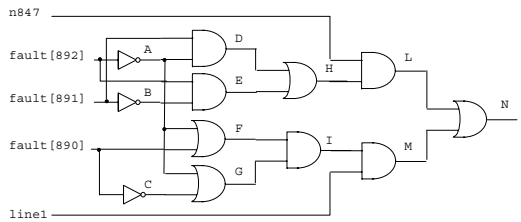

Figure 5: Logic-level representation of the code of Figure 4.

saboteur topology appears as in Figure 3. A program has been written to identify and remove each occurrence of the saboteur topology from the synthesized faulty description of the DUV. Unfortunately, the algorithm fails every time the saboteur logic is minimized and mixed with the functional logic of the DUV. In all such cases the topological mapping between stuck-at faults and bit coverage faults is almost infeasible. Consider for example the VHDL code of Figure 4 and the corresponding logic-level circuit of Figure 5. It is evident that the topology of saboteurs for faults 890, 891 and 892 is not recognizable. Another problem can be observed in Figure 5: the synthesis process can introduce fanouts on the lines of the `fault` port. This can lead the mapping algorithm to map one bit coverage fault into logic-level multiple faults. However, multiple faults are not managed by traditional ATPG tools, thus removing the advantage of the proposed approach.

3.3 Implication Method

The behavior of the saboteur logic, rather than its topology, can be analyzed to map bit-coverage faults into logic-level faults. The activation of a bit coverage fault allows to identify which nets are directly influenced by the fault. Consider the following algorithm:

```
for each bit coverage fault i do
    do not drive input lines of the DUV and propagate,
    through circuit, '0' on fault[j] if j ≠ i and '1' on fault[i]
    until a gate G is reached whose output logic value,
        depending on fault[i], cannot be fully determined
    for every such a gate G do
        for every input L of G depending on fault[i] do
            if the value of L is '0' add stuck-at 0 on L
            else if the value of L is '1' add stuck-at 1 on L
for each element k of fault port do fault[k] := '0'
set up and run logic-level ATPG
```

Faults mapping is performed by computing implications of the `fault` port elements. This approach avoids the necessity of identifying the saboteur topology, however, it does not resolve the problem of fanouts described in Section 3.2. For example, consider the circuit of Figure 5 and the bit coverage fault 890. According to the previous algorithm, '0' is propagated on `fault[891]` and `fault[892]` and '1' is propagated on `fault[890]`, while `line1` and `n847` are not

```
function inject_fault_dummy
  (object: bit; fault_index_s0: bit; fault_index_s1: bit)
   -- pragma map to entity inject_fault_dummy_entity
   -- pragma return_port_name result
   return bit is
  return object;
end;
```

Figure 6: Saboteur VHDL dummy function.

```
function inject_fault_bit(object: bit; fault_code: bit_vector;
  start_s0: integer; end_s0: integer; start_s1: integer;
  end_s1: integer) return bit is begin
  return inject_fault_dummy(object,
          fault_code(start_s0), fault_code(start_s1));
end;
```

Figure 7: Saboteur VHDL function for black box mapping.

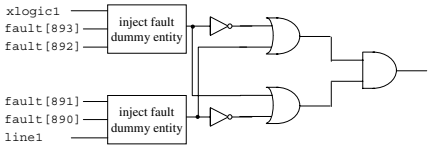

Figure 8: Logic-level representation of the code of Figure 4 by using the `inject_fault_dummy_entity`.

driven. Computing the value for all nets of the circuit, we obtain that: `A`='1', `B`='1', `C`='0', `D`='0', `E`='0', `F`='1', `G`='1', `H`='0', `I`='1', `L`='0', `M`='X', `N`='X'. The values of M and N cannot be determined, since they depend respectively on the value of (`'1' AND line1`) and (`'0' OR M`). Thus, the bit coverage fault 890 corresponds to the multiple logic-level fault (`stuck-at 0 on L, stuck-at 1 on I`). The implications for faults 891 and 892 can be computed in a similar way. The fault 891 is mapped to the pair (`stuck-at 1 on H, stuck-at 1 on I`) and the fault 892 is mapped to the pair (`stuck-at 1 on H, stuck-at 0 on M`).

3.4 Black Box Method

To avoid the disadvantages of the last two methods, the synthesis of the faulty DUV should not minimize the logic of saboteur functions into the functional logic of the DUV. Actually, the saboteur logic is useless for the logic-level ATPG, since we force it to detect logic-level stuck-at faults which are internally modeled by the ATPG. The correspondence between logic-level and bit coverage faults is based only on the position of the logic-level net affected by the saboteur. Thus, saboteur functions can be considered as black boxes with an activation signal for stuck-at 0, an activation signal for stuck-at 1, a fault-free input and a faulty output.

A state of the art synthesis tool [12] allows to map a function into a corresponding design entity, which is considered as a basic component of the selected technology library. Thus, a "dummy" saboteur function is defined (Figure 6); during synthesis it is mapped into a "dummy" entity which acts as a black box. The bit saboteur is modified (Figure 7) in order to use the dummy saboteur. This is a placeholder which simply assigns input to output; the same operation is performed by the corresponding entity. By using the dummy saboteur, the synthesized faulty design behaves exactly how the fault-free design does. However, after synthesis, a stuck-at 0 (1) on the output of an instance of the dummy entity corresponds to the bit coverage fault indicated by the *name* of the signal assigned to the `fault_index_s0 (fault_index_s1)` port. Thus, the sabo-

Design	Gates#	FF#	BC#	Tr.	Top.	Impl.	BB	Synthesis Time	F.C.%	ATPG Time
b01	54	5	201	201	122	175	201	42 s.	100.0	8 s.
b02	26	4	56	56	30	49	56	19 s.	98.2	4 s.
b04	584	66	408	408	214	394	408	38 s.	87.7	42 s.
b06	53	9	133	133	80	118	133	34 s.	100.0	4 s.
am2910	1400	124	3608	3608	1468	3519	3608	655 s.	93.6	3158 s.

Table 1: Experimental results.

teur logic is removed and each bit coverage fault is mapped to a single logic-level stuck-at fault. Consider the example of Figure 4 and the corresponding logic-level circuit obtained by using the black box method (Figure 8). The two instances of `inject_fault_dummy _entity` are not minimized with the functional logic of the DUV and the fanouts on fault port lines disappeared. To use a logic-level ATPG for testing bit coverage fault the following algorithm can be applied:

```
for each bit coverage fault i do
    search inject_fault_dummy_entity connected to fault[i]
    if i is Stuck-at 0 or false add Stuck-at 0 on the output of
       inject_fault_dummy_entity
    elsif i is Stuck-at 1 or true add Stuck-at 1 on the output of
       inject_fault_dummy_entity
set up and run logic-level ATPG
```

4. DESIGN MAPPING

The black box method described in the previous section allows to map signals of a high-level description to the corresponding nets of the logic-level implementation. In particular it supplies a unique net for every signal. In fact, given a signal occurrence of the RT/behavioral level description, the output line of the instance of the `inject_fault_dummy_entity` related to the fault affecting the signal is the desired net.

The strategy proposed in [10] works in a similar way. The authors introduce a stuck-at fault at the signal of interest in the RTL description. Then, they simulate the faulty RTL description to generate the responses for a selected testbench. Lastly, they propose to use a fault diagnosis engine for looking at these faulty responses and the original fault-free logic-level implementation to identify the desired logic-level net. However, this approach presents two main limitations: (1) The mapping succeeds only for signals related to detectable faults. Thus, low testable circuits represent a problem. (2) If a fault on signal S_1 is equivalent to a fault on signal S_2 the mapping could be not able to distinguish between the net corresponding to S_1 from the net corresponding to S_2. This problem is particularly acute when both stuck-at 0 and stuck-at 1 on S_1 are equivalent to stuck-at 0 and stuck-at 1 on S_2. In such cases the technique supplies more than one nets for the selected RTL signal.

The black box method overcomes the previous limitations since: (1) `inject_fault_dummy_entity` can be inserted both for detectable and undetectable faults. No information related to the testability of circuit are required for signal to net mapping. (2) Equivalent faults correspond to well distinguished `inject_fault_dummy_entity` instances. Then, no ambiguities can arise to discriminate different signals affected by equivalent faults.

The time required by the black box method to provide the pair (RT/behavioral signal, logic-level net) is the synthesis time. Given a signal occurrence and the related bit-coverage fault it is extremely easy finding the corresponding instance of `inject_fault_dummy_entity` in the synthesized design: it is the only instance connected to the line `fault[i]`.

5. EXPERIMENTAL RESULTS

Experimental results, performed on a workstation SunFire-280R equipped with 4GB of RAM, are reported in Table 1. Columns 2-4 show respectively the number of gates, memory elements and injected bit coverage faults. Columns 5-8 show how many bit coverage faults are mapped into logic-level faults by using the four methods investigated in Section 3. It is evident that both trivial and black box methods are able to map every bit coverage fault into a logic-level fault. However, as explained in Section 3, the first method is inapplicable in order to use a logic-level ATPG to detect such mapped faults, since the required ATPG setup time is unacceptable. On the contrary, the black box approach allows to efficiently exploit the potentialities of a logic-level ATPG as reported in the last three columns, which report respectively the black box mapping time (indeed, it is the time required by the synthesis process), the achieved fault coverage and the ATPG time by using the algorithm proposed in Section 3.4.

6. ACKNOWLEDGMENTS

We would like to thank Cristina Marconcini for her contribution in performing experimental results.

7. REFERENCES

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testings and Testable Design*. IEEE Press, 1990.

[2] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. An RT-level fault model with high gate level correlation. In *Proc. of HLDVT*, pages 3–8, 2000.

[3] F. Fallah, S. Devadas, and K. Keutzer. OCCOM-efficient computation of observability-based code coverage metrics for functional verification. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):1003–1015, 2001.

[4] F. Ferrandi, F. Fummi, and D. Sciuto. Test generation and testability alternatives exploration of critical algorithms for embedded application. *IEEE Transaction on Computers*, 51(2):200–215, 2002.

[5] A. Fin, F. Fummi, and G. Pravadelli. AMLETO: A multi-language environment for functional test generation. In *Proc. of ITC*, pages 821–829, 2001.

[6] F. Fummi, C. Marconcini, and G. Pravadelli. Redundant functional faults reduction by saboteur synthesis. In *Proc. of HLDVT*, pages 108–113, 2003.

[7] O. Goloubeva, G. Jervan, Z. Peng, and M. S. Reorda. High-level and hierarchical test sequence generation. In *Proc. of HLDVT*, pages 169–174, 2002.

[8] Mentor Graphics. *ATPG Tools Reference Manual*,2002.

[9] G. J. Myers. *The Art of Software Testing*. Wiley - Interscience, 1999.

[10] S. Ravi, I. Ghosh, V. Boppana, and N. K. Jha. Fault-diagnosis-based technique for establishing RTL and gate-level correspondences. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 20(12):1414–1425, 2001.

[11] M. B. Santos, F. M. Gonalves, I. C. Teixeira, and J. P. Teixeira. Implicit functionality and multiple branch coverage (IFMB): a testability metric for RT-level. In *Proc. of ITC*, pages 377–385, 2001.

[12] Synopsys. *Design Compiler User Guide*, 2002.

[13] Synopsys. *TetraMAX ATPG User Guide*,2002.