Abstract

In this paper, we focus on test pattern generation from behavioral circuit descriptions written in VHDL. While several approaches were presented in the recent past to solve this problem, none of them can deal with behavioral circuit descriptions containing loop constructs of VHDL. We therefore propose a method to generate test patterns for such circuit descriptions and we give the corresponding algorithms. A step-by-step execution of the algorithms is performed on two pedagogical examples in order to illustrate our approach.

1: Introduction

For the ten last years, research work has been oriented towards automatic Behavioral Test Pattern Generation (BTPG) which is the only way to generate tests from HDL behavioral circuit descriptions when no structural descriptions are available. Today, such a task is done manually and, thus, one of the main interest of BTPG is to automate this.

Modelling the fault hypotheses is an important task in BTPG. [1] introduced a high-level behavioral fault model which is associated with high-level HDL descriptions of digital designs. The derivation of these faults is based on the failure modes of the language constructs of the HDLs. The authors validated their behavioral fault model by performing a fault simulation of behavioral test sequences on a set of circuits and then compared the fault coverages obtained by this model and by the classical gate level stuck-at pin model. The correlation between the two fault coverages was good enough (more than 80 %) to adopt this fault model as a representative behavioral fault model in which some measure of confidence may be given.

Various conceptually different ATPG methods for behavioral descriptions have been proposed in the recent past [2,3,4,5,6,7,8,9,10]. Two different families can be distinguished : test pattern generation using symbolic simulation [4,9] and test pattern generation using path sensitization [2,4,5,6,7,8,10]. The first family leads to the analysis of symbolic expressions and the resolution of equations which can be difficult or even impossible to use for complex descriptions. The second, which is based on a fault-oriented approach and uses the path sensitization technique, deals with three well-known problems : local manifestation of a fault, fault effect propagation to a primary output and constraint justification. Two types of techniques are involved in order to solve these problems. The first, introduced by [3], consists in solving the fault effect propagation and the constraint justification using a "global approach" by considering a system of equalities which represents the complete solution space. This approach is limited because it cannot be easily applied on complex descriptions. The second technique, used in [2,5,6,7,8,10], consists in solving fault effect propagation and justification problems by using a local resolution approach. Choices are made in order to solve a problem locally by restricting the solution space to a tree. These approaches are linked to specific abstraction levels and to the HDLs the authors use.

All these approaches are interesting because they offer a solution to the problem of generating test patterns for behavioral circuit descriptions when no structural description is available. However, symbolic simulation and path sensitization according to a global resolution approach cannot deal with complex behavioral descriptions. Only path sensitization according to a local resolution approach is able to do so. Consequently, in this paper, we are only interested in those BTPG approaches based on this latter approach.

In order to be able to generate test patterns from complex behavioral circuit descriptions, existing BTPG systems must be able to deal with loop constructs which intervene in these circuit descriptions. Yet, except for [9], all the approaches described above deal with sequential and selective constructs but not with loop constructs. The loop type taken into account by [9] are the for-loops the number of iterations of which is fixed in the description. The approach of [9] is interesting but too limited. Since it is based on symbolic simulation, it cannot deal with every type of loops contained in VHDL [11], that is, while-loop and for-loop constructs.

We then became interested in determining a method to allow loop constructs to be handled in existing BTPG systems based on path sensitization according to a local resolution approach. This consists in determining how to solve the three well known problems of fault manifestation, fault effect propagation and constraint justification in the VHDL loop languages constructs.

We then define a formalization of the loops which is independent of the loop constructs. We use this formalization to develop two generation principles in loop constructs:the first one deals with fault effect propagation and the second with constraint justification. These principles are then applied to the two loop
constructs in order to determine algorithms for BTPG in circuit descriptions containing loop constructs. In order to illustrate our approach, we give pedagogical examples which show a step-by-step execution of the algorithms, using behavioral circuit descriptions.

The remainder of this paper is organized as follows. Section 2 presents an overview of our approach which deals with loop constructs in BTPG. Section 3 presents the formalization of the loop constructs we have defined. Section 4 and 5 are respectively devoted to fault effect propagation and constraint justification principles and describes the corresponding algorithms for each loop types. Section 6 gives two pedagogical examples which illustrate our approach. Finally, a conclusion regarding our approach and a presentation of our future work in BTPG are given.

2: Overview of our approach to deal with loop constructs in BTPG

We now present the main concepts on which we have based our method to deal with loop constructs which would be inserted in existing BTPG systems based on path sensitization according to a local resolution approach.

Three types of loop constructs are defined in VHDL: the while-loop construct, the for-loop construct and the "infinite loop". This latter construct is used with the statement "exit loop when condition" and only concerns sensible signals of the VHDL processes. In this paper, we are only interested in while-loop and for-loop constructs.

We first study how loop constructs must be taken into account in fault manifestation, fault effect propagation and constraint justification. Afterwards, we present how each loop type will be dealt with.

2.1: Loop constructs and path sensitisation

Path sensitization allows test patterns to be found by setting up constraints to manifest a fault, by setting up constraints to propagate a fault effect towards primary outputs and by solving the constraints on the primary inputs by justification. The local resolution approach consists in performing locally on the modelling items of the circuit description, the constraint setting up and solving. A loop construct present in a description will not alter the fault manifestation because the constraints set up during this step only concern the “faulty” modelling item, but it will alter the way the Test Pattern Search Procedure (TPSP) deals with propagation and justification. When the generation starts, all the variables of the description are unspecified. The TPSP begins and sets values on variables according to the description statements. At a given step of the search for a test pattern, some variables are still unspecified and other are specified by a value.

Definition

**State of variables:** We call states of variables the combination of values set up on the variables at a given step.

Moreover, the state of variables which is at hand when the TPSP encounters a loop construct of the description, is called the “attack state” and is noted as AS.

Definition

**Attack state AS:** We call attack state the state of variables which have been set up by the TPSP before a loop construct is encountered.

In fault effect propagation, when a loop construct is to be dealt with, two cases must be considered. First, the TPSP must determine whether the loop construct must be bypassed or not, according to the AS. Second, if a fault effect is present on the control of the loop in the AS, the number of iterations corresponding to fault-free values will be different from the number of iterations corresponding to faulty values.

As far as constraint justification is concerned, we must remember that it consists in determining the constraints needed to give the constraint at hand after a hypothetical execution of the item on which the justification is performed. In the case of a loop construct, constraint justification must set up the constraints located “upstream” from the loop needed to give the AS (the constraints “downstream” from the loop) by a hypothetical execution of the loop.

2.2: Taking into account VHDL loops

In order to develop a generic approach which is able to deal with the three types of loop constructs (while-loop and for-loop), we first define a generic formalization of a loop. We then apply this formalization to each loop type. Afterwards, we develop generic principles of fault effect propagation and of constraint justification in loop constructs. We then apply the generic principles to the two loop types. Moreover, the for-loop construct is dealt with as a corresponding while-loop construct. In this paper, we do not deal with "repeat-until" constructs since they are not defined in VHDL. However, our approach concerns this loop construct type too.

The next section is devoted to present the formalization of loop constructs we have developed.

3: Formalization of repetitive language constructs

In this section, we present the formalization we have developed to deal with repetitive constructs.

3.1: Generic formalization of loop constructs

The formalization of a loop construct we propose consists in determining whether a loop construct must be executed or not according to a state of variable. This leads to determine the sets of states of variables corresponding to the execution and to the non-execution of a loop. Since the execution or not of a loop is only determined by the control variable of the loop, we first classify the variables involved in a loop constructs and then define the notions of execution and non-execution states.

Variables are classified into two groups: data variables and control variables. Control variables are those variables which are present in the loop condition. All other variables are considered data variables. Moreover, we distinguish three types of variables: input variables, output variables and states variables. We call input variables those variables which are only read (that is, they are only present either in the condition or in the right parts
of assignments contained in the loop construct). We call output variables those variables which are only written (only present in the left part of the assignments). We call state variables those variables which are read and written (present in both parts of the assignments and even in different assignments). We now define the control state of a loop.

Definition **Control state of a loop L:**
We call control state of a loop L, the values of the control variables of the loop L.

A control state of a loop may correspond either to the execution or to the non-execution of the loop. We now give the characteristic of EXE and N-EXE which are two sets of control states of a loop.

Definition **The set EXE of execution states of a loop L:**
A control state of the loop L is an execution state of L if the values it contains give to the loop condition a value corresponding to its execution.

Definition **The set N-EXE of non-execution states of a loop L:**
A control state of the loop L is a non-execution state of L if the values it contains give to the loop condition a value not corresponding to its execution.

When executing a program containing a loop construct, a “upstream” non-execution state is not transformed but reported “downstream” the loop without any changes. In contrast, an execution state “upstream” the loop is transformed by the execution of the loop into a non-execution state "downstream" the loop. This non-execution state corresponds to the end of execution state. We call N-EXE* the set of these non-execution states corresponding to the end of execution of the loop.

Figure 1 gives the relationship between the sets EXE, N-EXE and N-EXE* of a loop according to a mathematical representation.

![Fig.1, Relationship between the sets EXE, N-EXE and N-EXE*](image)

In this section, we develop a generic principle of fault effect propagation in a loop construct and we then apply this principle to each loop type. This is done in order to describe the propagation algorithms in loop constructs which can be inserted in existing BTPG systems.

4: **Fault effect propagation principles in loop constructs**

In this section, we develop a generic principle of fault effect propagation in a loop construct and we then apply this principle to each loop type. This is done in order to describe the propagation algorithms in loop constructs which can be inserted in existing BTPG systems.

4.1: **Generic propagation principle**

We now develop a generic propagation principle in loop constructs. This consists in determining when the loop must be crossed or bypassed, how to deal with fault effects set up on control variables and how justify the constraints set up during propagation in a loop construct.

- **Propagation principle**

A generic principle of fault effect propagation can be defined by taking the AS characteristics into account. The AS can be either an execution state or a non-execution state. Sometimes, control variables may be still unspecified. In this case, AS cannot be classified. We now describe the processing to be performed in these three cases:

- **AS ∈ EXE and AS ∈ N-EXE**: In this case, constraints must be set up on unspecified control variable so that AS becomes an execution state or a non-execution state. Sometimes, control variables may be still unspecified. In this case, AS cannot be classified. We now describe the processing to be performed in these three cases:

- **AS ∈ EXE**: In this case, fault effects must be propagated on the statements of the loop body. However, in order to perform a determined propagation, every input variables and state variables of the loop must be specified. If some of them are still unspecified, contraints must be set up on them. Afterwards, the propagation can be performed according to the statements of the loop body. The loop is then crossed and the TFTP performs a certain number of loop passes until a non-execution state is obtained. We do not define the principles of fault effect propagation on the loop body statements since these principles are already defined in existing BTPG systems which our approach will be inserted.

- **AS ∈ N-EXE**: In this case, AS must be reported downstream the loop without any further processing.

- **Dealing with fault effects set up on control variables**

Fig. 3: For-loop translation into a while-loop

<table>
<thead>
<tr>
<th>Loop</th>
<th>Control variables</th>
<th>Input variables</th>
<th>Output variables</th>
<th>State variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE a + b &lt; 10 do begin x := a - e; s := s + e; a := a + 1; end;</td>
<td>a, b</td>
<td>b, e</td>
<td>s</td>
<td>a, x</td>
</tr>
<tr>
<td>Example of state of EXE: AS=[a=5, b=0]</td>
<td>Example of state of N-EXE: AS=[a=5, b=6]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
If no fault effect is present on a control variable, the loop is passed an equal number of times by both the faulty and the fault-free values. On the contrary, if a fault effect is present on a control variable, the loop is not passed an equal number of times by both the fault-free values and by the faulty values. Two new states are built, called **FAULTY** and **FAULT-FREE**, which are respectively the state of faulty values and the state of fault-free values. **FAULTY** and **FAULT-FREE** are not necessarily belonging simultaneously to **EXE** or **N-EXE**, but one may belong to **EXE** and the other to **N-EXE**. For both of these states, it is necessary to propagate their values by using the propagation principle described above. After the loop has been crossed (or bypassed) by the two separated states, we recombine the two into one state in order to reconstitute the fault effects. This recombination is performed as follows: if a variable has the same value in **FAULT-FREE** and **FAULTY**, the value is set up on this variable in the recombined state; otherwise, when values differ in **FAULTY** and **FAULT-FREE**, a fault effect is reconstituted in the recombined state.

**Justification of the propagation constraints**

The constraints which have perhaps been set up on control variables, inputs and state variables must be justified on the statements upstream the loop.

The propagation principle described above allows each loop type to be dealt with in fault effect propagation, whatever the loop may be. This generic principle of fault effect propagation can be applied to each loop type. In next sub-section, the fault effect propagation algorithms for the while-loop and for-loop constructs will be given.

### 4.2: Fault effect propagation algorithms for loop constructs

We now describe the propagation algorithms for each loop type from the generic propagation principle. As mentioned previously, an algorithm is developed first for the while-loop and then an algorithm is derived for the for-loop. As said previously, the fault effect propagation on the loop body statements is performed according to the propagation principles of the existing BTPG approaches.

**While-loop**: In a while-loop, the fault effect propagation algorithm is obtained by applying the generic propagation principle without any changes. This algorithm is given in Figure 5.

**Fig. 5**: Fault effect propagation algorithm for a while-loop

```
1. If control variables are unspecified then set up constraints on them
2. If the value of AS correspond to the condition at FALSE then goto 9
   else
   3. If input or state variables are unspecified then set up constraints on them
   4. If no fault effect is on a control variable then
   5. While the values of AS correspond to the condition at TRUE do
      perform one loop pass by propagating on the loop body statements
   else
   6. Separate **FAULTY** and **FAULT-FREE**
      for each of them do
      7. While the values of the state correspond to the condition at TRUE do
         perform one loop pass and propagate fault effect on the loop body statements
      end
      8. Recombine the two states into one
   end
9. End
```

**For-loop**: The propagation algorithm for a for-loop is simplest. The for-loop initializes its control variable itself so that the AS is always an execution state. This algorithm is given in Figure 6.

**Fig. 6**: Propagation algorithm for a for-loop

```
1. If input or state variables are unspecified then set up constraints on them
2. counter = expression_1
3. While counter = expression_2 do
   3.1 perform one loop pass by propagating on the loop body statements
      3.2 counter = counter + 1;
3. End
```

We have described the algorithms which deal with fault effect propagation in each loop type. We now focus on constraint justification in loop constructs.

### 5: Constraint justification principles in loop constructs

In this section, we develop a generic principle of constraint justification in loop constructs and we then apply this principle to each loop type in order to describe the constraint justification algorithms for loop constructs which can be inserted in existing BTPG systems.

#### 5.1: Generic justification principle

As was the case for fault effect propagation, the processing undertaken by the TPSP when it encounters a loop construct in justification depends on the AS characteristics. In order to better follow our approach in justification, it is necessary to keep in mind that the aim of justification is to determine the constraints upstream of the loop needed to give the AS by a hypothetical execution of the loop.

When the TPSP deals with a loop construct in justification, the AS may be an execution state, a non-execution state or some control variables are not specified. We now examine the processing to be performed in these three cases.

- **AS \( \in \) EXE and AS \( \notin \) N-EXE**: In this case, constraint must be set up on unspecified control variables so that AS becomes either an execution state or a non-execution state of the loop.
- **AS \( \in \) EXE**: This case corresponds to a failure in justification. Indeed, it is never possible to obtain an execution state downstream a loop. Therefore, no execution state upstream the loop could be found to obtain the AS downstream the loop. The TPSP will have to invoke its backtracking procedure to manage this failure.
- **AS \( \in \) N-EXE**: In this case, we must consider whether AS belongs to N-EXE* or not. If yes, it is sure that some execution states exist upstream the loop which correspond to the AS downstream the loop.
  - First case : AS \( \notin \) N-EXE*: AS must then be reported upstream the loop without any further processing.
  - Second case : AS \( \in \) N-EXE*: Two processing must be performed. A first solution consists in reporting AS upstream the loop without any further processing (indeed, it belongs to N-EXE).

A second solution consists in justifying the constraints of AS on the statements of the loop body. In order to justify the constraint in a determined way, constraints must be set up on every unspecified output variables and state variables of the loop. A certain number of loop passes can be performed while the values of variables must be in their value domain. Each constraint set obtained by the successive loop passes in justification corresponds to a solution upstream the loop.
The problem of determination of non-execution states which belongs to N-EXE* must be dealt with. Indeed, it is very easy to determine if a state of variables corresponds to an execution or a non-execution state of a loop, but it is not easy to determine whether a non-execution state is also a state of end of execution of the loop. We then propose a method which will have to be inserted in the justification algorithms. By operating one loop pass in justification on the statements of the loop body from AS, two cases can be obtained:

- the state obtained is still a non-execution state,
- the state obtained is an execution state.

In the first case, AS does not belongs to N-EXE*. In the second case, AS is a state of end of execution of the loop and belongs to N-EXE*.

Figure 7 gives an example of determination of a state of end of execution. It shows too the different constraint sets which are solution to the justification in the loop construct.

The justification principle described above allows each loop type to be dealt with in constraint justification, whatever the loop may be. This generic principle of constraint justification can be applied to each loop type.

Table 1: Description of a circuit

<table>
<thead>
<tr>
<th>Description</th>
<th>Circuit1</th>
<th>Example of justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable i,a,e : integer range 0 to 128;</td>
<td>begin</td>
<td>Let AS=i=10, a=24; AS \in N-EXE*</td>
</tr>
<tr>
<td>begin ...</td>
<td>% First pass : [i=9, a=22, e=2] \notin EXE*</td>
<td></td>
</tr>
<tr>
<td>While i &lt; 10 do</td>
<td>Therefore AS \in N-EXE*</td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td>Others passes in justification</td>
<td></td>
</tr>
<tr>
<td>a := e + a;</td>
<td>Second pass : [i=8, a=20, e=2]</td>
<td></td>
</tr>
<tr>
<td>i := i + 1;</td>
<td>Third pass : [i=7, a=18, e=2]</td>
<td></td>
</tr>
<tr>
<td>end;</td>
<td>etc</td>
<td></td>
</tr>
<tr>
<td>Fig. 7, Example of determination of a state of N-EXE* and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>solution constraint sets upstream the loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the following sub-section, we present constraint justification algorithms for the three loop types.

5.2: Constraint justification algorithms for loop constructs

As was the case for propagation, an algorithm for the while-loop is developed first and this algorithm is derived for the repeat-until and for-loops. The justification on the loop body statements is performed according to justification principles defined in the existing BTPG systems.

- **While-loop**: In a while-loop, the generic principle can be applied without any changes. The constraint justification algorithm for a while-loop is given in figure 8.

![Fig. 7. Example of determination of a state of N-EXE* and solution constraint sets upstream the loop](image)

**Fig. 8. Constraint justification algorithm for a while-loop**

- **For-loop**: As was the case in propagation, constraint justification in a for-loop is quite different from the other two loops. Indeed, at the end of a for-loop, after a hypothetical run of the loop, only a compatible and final AS may be found. There is a failure of justification when the AS is either initial or final and incompatible. The constraint justification algorithm for for-loop the number of iterations of which is known is given in Figure 9. When the number of iterations is unknown, the algorithm is exactly the same as for a while-loop to which must be added the counter management.

![Fig. 9. Constraint justification algorithm for a for-loop](image)

Since the for-loop places itself in a initial state in execution, it is not necessary to justify the constraint set up on the counter. This constraint is therefore removed.

These algorithms allow constraint justification to be dealt with in loop constructs.

6: Pedagogical examples

In order to illustrate our approach concerning test pattern generation in behavioral circuit descriptions containing repetitive language constructs, we give two examples concerning two behavioral descriptions containing loop constructs.

6.1: Example of constraint justification in loop constructs

This example uses the description presented in figure10. This descriptions proceeds from [12,13]. A test pattern is generated for a behavioral fault defined by [1] is used (fault of an assignment where the assigned data item assumes the lower or upper extreme of its range).

![Fig. 10. VHDL description containing while-loop constructs](image)

We consider the fault concerns the signal assignment `oup<=x`. So that `oup` is permanently at 0" The three steps of generation are solved as follows:

- **Fault manifestation**: The value 255 allows the fault to be manifested and the constraint `x=255` must be set up. A fault effect must be propagated: `oup=(255 0)`.
- **Fault effect propagation**: Since `oup` is a process output, a fault effect is observable and this step is successful.
- **Constraint justification**: The manifestation constraint is first justified in the then branch of the if-then-else. The main while-loop is to be dealt with (control variable: `y`; state variables: `x, y, h`). Since `x` and `y` are unspecified, we set up a constraint on each on them: `y=0` which is the only constraint which corresponds to a non-execution AS, and `h=0`. We obtain AS=`{x=255, y=0, h=0}`. AS belongs to N-EXE*.

After the three assignments, we have `{x=0, y=255, h=0}` which is the AS at the second loop (in this loop, control variables: `x, y`; state variable: `h`) and belongs to N-EXE*. Therefore the second loop is crossed in justification. Upstream from
the second loop, we obtain \( x=(255,y=255) \). At this point, two cases can be considered: either perform another loop pass in justification, or leave the loop. In this example, we choose to leave the loop. Similarly, we choose to leave the main loop. The constraints obtained are consistent with the then branch of the if-then-else. Therefore, we obtain \( [x=255, y=255] \). This step is successful. Since the three steps are successful, a test sequence has been found.

6.2: Example of fault effect propagation

We now propose to illustrate the propagation principle in loop constructs with the description given in figure 11, which proceeds from \([14]\) and describes the function ADD8.

```
Package SYSTEM is
  subtype WORD is TSV(7 downto 0);
  function ADD8 (A: WORD; B: WORD) return WORD is
    subtype TSV is ARRAY(1 to 3) OF WORD;
    subtype INTEGER range 0 to 3 := 0;
    variable S : TSV(1 to 3) := 0;
    variable SUM : WORD := 0;
    function ADD8 (A: WORD; B: WORD) return WORD is
      subtype WORD is TSV(7 downto 0);
      variable NUM : INTEGER range 0 to 3 := 0;
      variable SUM : WORD := 0;
      variable CARRY : INTEGER range 0 to 1 := 0;
      begin
        for K in 1 to 3 loop
          variable CARRY : INTEGER range 0 to 1 := 0;
          variable SUM : WORD := 0;
          variable NUM : INTEGER range 0 to 3 := 0;
          variable S : TSV(1 to 3) := 0;
          when 0 = SUM(I) := '1'; CARRY := '0';
          when 1 = SUM(I) := '0'; CARRY := '0';
          when 2 = SUM(I) := '1'; CARRY := '0';
          when 3 = SUM(I) := '0'; CARRY := '0';
          case NUM is
            when 0 => SUM(I) := '0';
            when 1 => SUM(I) := '1';
            when 2 => SUM(I) := '0';
            when 3 => SUM(I) := '1';
          end case;
        end loop;
      end loop;
      -- Input variables:
      for K in 1 to 3 loop
        S(K) := ('0' & ('1' & ('0' & ('1' & '0'))));
      end loop;
      -- Output variables:
      for K in 1 to 3 loop
        SUM(K) := ('1' & ('0' & ('1' & '0')));
      end loop;
      -- Constraint justification:
      if S(K)='1' then NUM := NUM + 1;
      end if;
    end loop;
  end loop;
end SYSTEM;
```

Fig. 11, Description containing for-loop statements

We suppose the function ADD8 is invoked during a propagation phase with \( A=111110000 \) and \( B=11111111010000 \). The first statement to be dealt with is the for-loop. We have \( \text{AS}=[A=CARRY='0', \text{NUM}=0, A='111110000', B='11111111010000'] \). In this loop, the classification of the variables is the following: Control variable : \( i \); Input variables : \( A, B \); output variables : \( S, \text{SUM} \); State variables : \( \text{CARRY}, \text{NUM}, i \). Since all variables are specified, the loop body is crossed in propagation for each value of \( i \) between 0 and 7.

With \( i=0 \), we have \( S = 00 \) and \( a = 000 \). The second loop must now be taken into account (Control variable : \( K \); State variables : \( K, \text{NUM} \); Output variable : \( S \) ). The loop body is crossed in propagation for each value of \( K \) between 0 and 3. With \( K=1 \), we keep \( \text{NUM}=0 \); with \( K=2 \), we obtain \( \text{NUM}(1)=0 \) because \( S(2)=1 \); with \( K=3 \), \( \text{NUM} \) is not changed. We come back to the first loop. Since \( \text{NUM}(1)=0 \), the branches 1 et 0 of the case is respectively selected by the fault-free values and by the faulty values. At the end of the case, we have \( \text{SUM}(1)=100 \) and \( \text{CARRY}=00 \).

With \( i=1 \), \( i=2 \) and \( i=3 \), we obtain the same results: \( \text{SUM}(1)=100 \) then \( \text{SUM}(2)=100 \) then \( \text{SUM}(3)=100 \). With \( i=4 \), we have \( S=101 \) and \( \text{CARRY}=100 \). The second for-loop gives \( \text{NUM}(2)=1 \). In the case, we obtain \( \text{SUM}(2)=100 \) and \( \text{CARRY}=100 \). With \( i=5 \), we have \( S=100 \) and \( \text{CARRY}=100 \). The second for-loop gives \( \text{NUM}(3)=1 \). In the case, we obtain \( \text{SUM}(3)=1 \) and \( \text{CARRY}=100 \). With \( i=6 \) and \( i=7 \), the same results that with \( i=5 \) are obtained that is \( \text{SUM}(6)=100 \) and \( \text{CARRY}=100 \). At end, we obtain \( \text{NUM}(3)=1 \) and \( \text{SUM}(3)=1 \). The function ADD8 returns the fault effect set up on \( \text{SUM}(1)=100 \). This fault effect will then be propagated to a primary output in the main description which invoked the function ADD8.

The two pedagogical examples show how loop constructs are be dealt with in fault effect propagation and constraint justification by two step-by-step executions of the algorithms. Other loop types can be dealt with in the same way.

7: Conclusion and future work

We have proposed in this paper an approach to deal with loop language constructs for fault-oriented behavioral test pattern generation approaches stemming from the path sensitization method according to a local resolution approach. Our method involves three steps. We first define a formalization of loop constructs in order to provide an representation appropriate with test pattern generation. We then define a constraint justification principle and a fault effect propagation principle in loop constructs. Propagation and justification algorithms are given for each loop type. Two pedagogical examples of step-by-step execution of the algorithms are proposed to illustrate our approach.

Our future work will be oriented towards the implementation of our approach concerning loop constructs in an existing BTPG system. Afterwards, experiments will be carried out in order to validate this approach.

References