A comparative analysis of the functional and the structural level fault detection for sequential machines

Mijland, H.P.J.

Award date:
1992

Eindhoven University of Technology
Department Electrical Engineering
Digital Systems Group (EB)

A COMPARATIVE ANALYSIS OF THE FUNCTIONAL AND THE STRUCTURAL LEVEL FAULT DETECTION FOR SEQUENTIAL MACHINES

By H.P.J. Mijland

Master Thesis
Eindhoven, the Netherlands
September 1989 - June 1990

Supervisor: Prof. Ir. M.P.J. Stevens
Coach: Dr. Ir. L. Jóźwiak

ABSTRACT

In this report a new concept to generate test vectors for sequential machines is checked. Not one level of abstraction is considered but two levels. The test generation occurs at the functional level, while information is retrieved from the structural level.

Four methods for test generation, using structural level information at the functional level were used to check the concept. Test generation was done for a few examples. A comparison was made with a conventional functional level test generation algorithm and a conventional structural level test generation algorithm.

The concept of using structural level information on functional level test generation gave for all the examples better results than the conventional test generation algorithms.

CONTENTS

LIST OF USED ABBREVIATIONS

cov	fault coverage
MEM	memory elements
noib	number of input bits
nomult	$\log_2$(multiplicity)
NSL	next state logic
OL	output logic
PI	primary inputs
PO	primary outputs
rntv	relative number of test vectors
sa0	stuck-at-zero
sa1	stuck-at-one
SI	secondary inputs
SO	secondary outputs

1. INTRODUCTION

The subject of this report is the testing of sequential machines. A lot of work has been done in recent years, to facilitate the algorithms used for test generation. So far a lot of this work is done to convert the test generation algorithms for combinatoric circuits to be used for sequential machines. There are two important disadvantages of this conversion.

The first disadvantage is that the sequential machine is considered as a combinatoric circuit, not taking into account the sequential character of the machine. Most of the times these methods cut the loops, caused by the memory elements, which leaves a combinatoric circuit.

The second disadvantage is that the machine is described at one level of abstraction only. By looking at only one level a great deal of information is lost about the machine. This results in too many test vectors.

The aim of this work is to check the following concept: use information from more levels to generate test vectors. This has two advantages. First, that a machine can give different information on several levels, so maybe test generation becomes easier and less test vectors must be generated to guarantee a given fault coverage. Second, that the sequential character of the machine does not get lost, so the machine is tested as a sequential machine.

In this report two levels of abstraction are considered: the functional and the structural level. First attention is paid to the conventional test generation algorithms at the two levels and the use of these for sequential machines. Then, four test generation methods are considered. These methods generate test vectors at the functional (state diagram) level making use of information from both levels.

To check the concept described above, a comparative analysis of the methods has been made using several sequential machines from the international banchmark set [1].

2. FUNCTIONAL AND STRUCTURAL LEVEL TESTS

2.1. Introduction

In the last decades very much work has been done in developing test algorithms. The starting point in these algorithms is always one level of abstraction of the digital circuit description. In this chapter some of these algorithms, which are known from literature (e.g. [5]), will be described. The concept which will be checked in this work: the use of two levels of abstraction for test generation, will also be explained.

2.2. Structural level testing

A digital circuit described on the structural level is composed of logical gates. At their turn logical gates are composed of transistors. Depending on the technology used, these transistors can be bipolar transistors or MOSFETS. After fabrication these transistor circuits may contain faults. These faults can have in two ways effect on the performance of the circuit. One way is that the faults change the parameters of the circuit, like longer delay times. The other is that the circuit faults cause logical faults in the gates. All these kinds of faults can cause improper operation of the circuit.

Testing a digital circuit, the definition of a good fault model is necessary. A digital circuit can be tested on timing, on power consumption or other parameters. Also the circuit can be tested for logical faults. Two very important types of logical faults are stuck-at faults and bridging faults. Stuck-at faults are faults which cause a signal line to be always logic "1" or logic "0", the first named stuck-at-one (sa1), the second named stuck-at-zero (sa0). Bridging faults cause two signal lines which are next to each other to be connected. Depending on the technology used, the resulting signal can be the logic AND or the logic OR function for these two signals.

The logical fault model does not take into account the physical faults which can occur in the transistor implementation of the machine. Still most structural level fault models used are based on the logical fault model. In testing PAL's or PLA's bridging faults are very important. For digital circuits stuck-at faults are considered most important. Most test algorithms are based on the occurrence of only a

single stuck-at fault. This model is therefore known as the single-stuck-at fault model. It is widely used and also widely accepted.

After defining a fault model, test generation can begin. Many algorithms have been proposed in the past decades, of which two are the best known, the D-algorithm (D=discrepancy) and the PODEM (path oriented decision making) algorithm. In the circuit under test a fault is injected, and the algorithms try to excite the fault and propagate its effect to an output. The disadvantage of these algorithms is that they use backtracking to find a path to propagate the effect of a fault. This means that generation time can be very long.

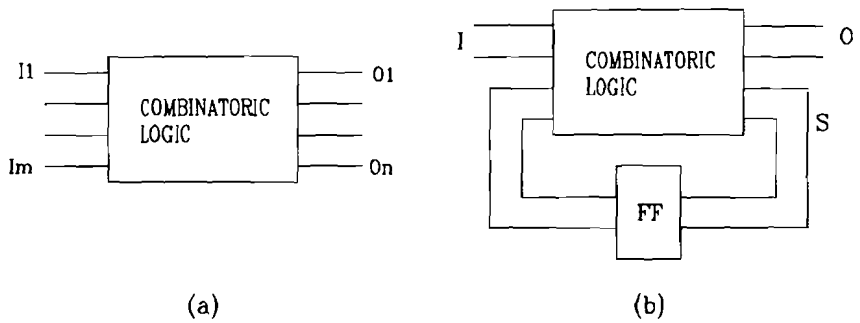

Figure 1. (a) a combinatoric circuit and (b) a sequential machine.

Digital circuits can be divided into two classes: combinatorial and sequential. In figure 1.(a) a combinatoric circuit is shown. The outputs are a function of the inputs only. In a formula this is

$$O_i = f_i (I_1, .. , I_m) \qquad i=1,..,n \qquad (1).$$

For a sequential machine the outputs are a function of the inputs and of the present state of the machine. This state is held by a set of memory elements, outlined in figure 1.(b). The general formula for a sequential machine is normally given as the 5-tuple

$$M = (I, S, O, \text{delta}, \text{lambda}) \qquad (2),$$

in which I is the set of inputs, S is the set of states, O the set of outputs, delta the function which relates S to I and S, and lambda the function which relates O to S in case of a Moore machine and to S and I in case of a Mealy machine.

For combinatorial circuits it is quite easy to generate test patterns, because a fault can be excited and propagated to an output by only one test vector. For sequential machines the effect of a fault can be propagated through the memory elements, so it needs a sequence of test vectors to detect a fault. Most of the

structural level test generation algorithms are made for combinatorial circuits. With a few changes it is possible to use them for sequential machines.

A very popular way the combinatorial test generation algorithms are used for sequential machines is the scan technique. The goal of this technique is to make the state of the sequential machine (the bit pattern held by the memory elements) visible. This is only possible when synchronous sequential machines are considered. The memory elements are coupled into a shift register. Testing is done by shifting in and out test vectors. After generating a test vector, it is shifted in, an input is applied after which the resulting state is shifted out and compared to the expected next state.

There are two disadvantages of this method. One is that it takes a long time to shift in and out test vectors. The second is that the sequential machine is degraded into a combinatorial circuit, not taking into account the sequential character of the machine.

Also the other test generation algorithms for combinatoric circuits have the disadvantage like in previous example. The way in which they are used for sequential machines is kind of artificial and the sequential character of the machine is neglected.

2.3. Functional level testing

In this report attention is paid to the functional level testing of sequential machines. The functional level description of a sequential machine is done by a state transition table or a state transition graph. An entry in a state transition table consists of a present state, an input and a next state. These three combined is called a transition.

For functional level testing the definition of a good fault model for this level is necessary. Gupta and Armstrong [2] have defined a functional level fault model consisting of two types of faults. First they map physical faults to the functional level. Secondly they scatter the functional description which causes functional faults. The first type of faults is thus extracted from a different level of description then the functional level.

The disadvantage for functional level testing is that normally little or nothing is known about the implementation of the machine on a lower level of abstraction. This results in a fault model that contains only the faults originated in the disturbance of the functional level description of the machine.

The advantage in functional level testing is that generation time is smaller than for structural level testing. Disadvantage is that there is no connection between the functional level description and the implementation on a lower level. Therefore functional level testing most of the time comes down to exhaustive testing. This means that for larger sequential machines the testing time grows exponentially with the growth of the number of input bits and the number of memory elements.

2.4. Combining structural and functional level testing

For functional level testing the generation time can be very short but the absence of a good fault model results in a very long testing time. For the structural level algorithms the test generation time can be very long, but the testing time is short. Why not combine these two levels, so profit is taken from the advantages, while maybe the disadvantages disappear. Generating test vectors at the functional level, looking at the structural level, means that only those faults which influence the functionality of the machine are considered. The concept proposed here is to use the information about the faults on the structural level in the functional level test generation. The purpose for doing so is to shorten the generation time but also shorten the testing time.

If this concept works, further work can be done to use more levels of abstraction. It could be possible to find a fault model on the lowest level of abstraction, the level describing the silicon lay out, from which information is derived which in its turn can be used on a higher level of abstraction. This method shows an analogy with the design of digital circuits, which tends to be done hierarchically.

One of the goals in the design of a circuit at the structural level can be to keep the circuit irredundant. Sometimes redundancy is added to prevent the appearance of timing problems. An other form of redundancy is important in this report. When a sequential machine is defined, it consists of a set of states, and a set of inputs which can be combined to form a sequential machine. This sequential machine is able to generate a sequence of outputs, which are a function of the sequence of inputs. The states have to be implemented in a circuit at the structural level. These states are normally defined as a set of bits in a register. This register can hold as many states as two to the power of the number of state bits. This means that when the sequential machine has e.g. twelve states defined, and the implemented register has four bits, sixteen states are realized.

Another form of redundancy occurs when a certain input is forbidden from a given state. This is possible when the surrounding logic never generates this

input. In the definition of the machine at functional level this input is not possible, but in the implementation of the machine at the structural level this input is possible and will cause a transition which is not defined. This shows the difference between the definition and the realisation of a sequential machine.

Generating test vectors at the functional level, using information of the implementation at the structural level, causes that it is not necessary to test on faults which have no effect on the functional behaviour of the machine. When the test generation occurs on the functional level it is even impossible to test these faults, because a certain input which could detect such a fault, is not allowed to occur. Also it is impossible to reach a state which could detect the fault, because the state is not defined. Therefore it is impossible to generate test vectors which detect faults that only effect the redundancies in the implementation at the structural level. But this is not necessary because the faults don't effect the functional behaviour of the machine, not during testing and also not during operation.

2.5. Structural level information

It is not necessary to extract all the information from the structural level description of the machine under test. Some restrictions are made to make comparison possible.

A sequential machine can be build out of three blocks. In the next figure (Figure 2.) these three blocks are shown. The signal lines in this figure have the following meaning: PI are the primary inputs, the real inputs to the machine, SI are the

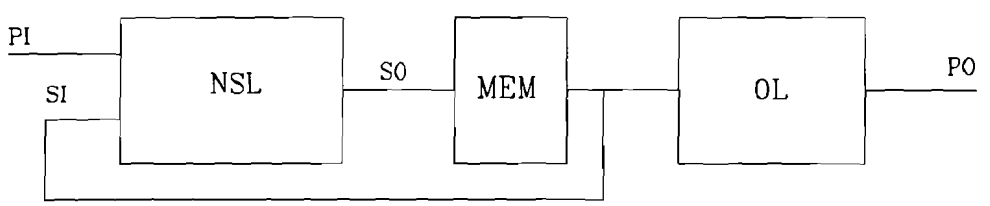

Figure 2. Logic blocks of a sequential machine

secondary inputs or the outputs from the memory elements, PO are the primary outputs, the real outputs for the machine and SO are the secondary outputs or the inputs for the memory elements. The logic blocks are called NSL, MEM and OL. NSL means next state logic, which is a combinatorial circuit which computes the next state. MEM is the memory part which contains the memory elements. This can be a register, a counter or flip flops. The block OL computes the outputs and contains the output logic which is also a combinatorial circuit. In this figure the OL is a function of the SI only. This means that the machine is a Moore type. When the machine is of the Mealy type, the output also is a function of PI. This means that there has to be a signal line from PI to OL. For this report it makes no difference if the machine is a Moore- or Mealy-type.

In the previous subsection it is explained that it is not necessary to exhaustively test the sequential machine, looking at the structural level. This is also the case for the OL. When there are in the Moore type machine four memory bits, but only twelve states defined, then it is not necessary to test all sixteen possibilities, which the four memory elements can contain. The reason for this is the same as in the previous subsection, that faults which only influence transitions that do not occur, do not influence the functional behaviour.

The methods which will be explained in the next chapter, take into account only that part of the sequential machine, that is composed by the NSL. The reason for this is twofold. The first is that when the NSL is fault free, then the other two parts can be tested as well. Secondly, all the faults which can occur in the MEM-block and the OL-block can be processed as well, together with the NSL-block, in the methods following in the next section. However for comparison with the conventional test generation algorithms, only the NSL-block will be dealt with here.

The four methods assume that the state denoted in SO, can be made visible to detect a faulty next state. There are a few methods in which this is possible.

1: This method is mentioned earlier in subsection 2.2. and is called scan technique.

2: In an article published by Devadas c.s. [3], he proposes a synthesis procedure in which states are assigned in a sequential machines, such that a single stuck-at fault is easily detected. The state assignment makes sure that a single stuck-at fault in the NSL has such an effect on the next state, that the faulty next state does not have the same output as the correct next state. This is possible if for example the stuck-at fault only influences one state bit. In the definition of the machine, two states which have the same output, get assigned two patterns which differ in more than one bit.

3: When all states in the sequential machine have different outputs, it is possible to define the inverse function of the function which is defined in the OL. This inverse function can be added to the PO, and thus computing the present state of the machine. When different states have the same output,

then an extra output bit has to be added in the OL to determine which of the states the machine is in.

Stuck-at faults in the structural level description of the sequential machine have or have no effect on the functional level description. When a stuck-at fault in the NSL occurs, the only effect it can have will occur in the next state. If the effect of the stuck-at fault propagates to one or more next state lines, this will result in a faulty next state. This means that a stuck-at fault which is not redundant, will always have an effect on a transition from one state to another. When the state of the sequential machine is made visible by one of the three methods, mentioned above, all the irredundant stuck-at faults can be detected by a transition. This means that it is possible to assign stuck-at faults to transitions. This is what the concept of this work is based upon.

2.6. Conclusion

In this section structural and functional level test methods are treated. The disadvantages for structural level testing is the large generation time and the extra work which is done for faults that do not effect the proper operation of the machine. For functional level test generation, there is no well defined fault model, so it results in exhaustive testing. The generation time is small, but the testing time is much too long for large machine.

That is why a new concept is introduced: use information from the structural level to generate test vectors at the functional level. The goal is that this might result in less complex algorithms than the structural level test generation algorithms, while they also generate less test vectors than the functional level test generation algorithms. An advantage is that the new concept does not consider redundancies.

3. TEST METHODS BY COUPLING LEVELS

3.1. Introduction

A transition at the functional level is realised by logic at the structural level. Some transitions use more logic than others. Therefore these transitions can be disturbed by more logical faults than others. The assumption made here is that if a transition has a higher fault measure, it has to be tested more than other transitions. Therefore the next methods all use a measure for transitions telling something about the functional fault probability and the difficulty to test a transition. This is the information which is derived from the structural level.

In the following subsections four methods will be introduced. There will be an increase in the use of lower level information. After that the expected performance of each method will be dealt with.

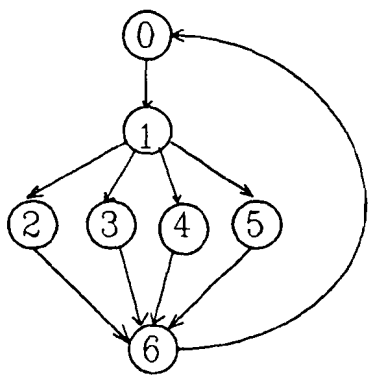

Figure 3. Example of a state transition graph of a sequential machine.

All of the methods use the functional level description of the machine for test generation. An example of the functional level description of a sequential machine consists of a state transition graph, shown in figure 3. (the edges represent inputs, while the nodes represent states). It can be considered as a set of transitions. One transition consists of three parts, a present state, an input and a next state. As mentioned in subsection 2.5. the output is not important in these methods. Both the present state and the next state are completely specified at the structural level. An example for a transition is

IOWAIT reset START (3),

in which IOWAIT and START are the present state and the next state respectively, and reset is an input. After state assignment and specification of the inputs, this transition can be written as

```
11010      0---      00000                    (4).
```

Here the number of state bits is five and the number of input bits is four. In the input '-' means that this bit is don't care.

The input has the possibility of containing don't care bits. An example is given in (4), where the first bit controls the reset of the machine. This means that for a transition with don't care input bits there are more possibilities to transit from the present state to the next state. This is called multiplicity.

Exhaustive testing of a sequential machine means that multiple transitions have to be tested. The following methods try to restrict the number of test vectors in such a way that still most of the faults are detected.

For simplicity reasons the single-stuck-at fault model will be assumed at the structural level, to generate test vectors and calculate fault coverage.

3.2. Statistic testing

Random techniques are widely used in all kinds of algorithms. Also for testing sequential machines the random technique is used. A known good device is compared to a circuit under test. Both of the circuits get offered the same inputs. The outputs of both circuits are compared and if they differ the circuit under test is marked bad. This method is often used for very large machines. The time used to generate test vectors is short, but the fault coverage is often not guaranteed to be high.

De first method (method 1.) is based on random techniques. It only considers the state transition graph, with transitions defined as in (4). Therefore it follows that the multiplicity is dependent on the number of don't care bits in the input.

On the other hand it is possible to reach a next state from a present state with two totally different inputs. This also increases the multiplicity. When a next state is reachable through two totally defined inputs, starting from the same present state, the multiplicity for this transition is two.

The method of statistic testing assumes that with the increase of the multiplicity, the hardware concerned with the transition also increases. This means that the number of gates for a transition with high multiplicity is larger than for a transition with multiplicity of for instance one.

Because nothing is known about the implementation of the machine, it is hard to generate test vectors. Heuristics however say that a test vector containing many zero's will detect more sa1's, while a test vector containing many one's will detect more sa0's. This is true, because a logical "1" will easy excite a sa0, while a logic "0" will easily excite a sa1.

Another vector which detects certain faults is a vector which has alternating zero's and one's. This test vector will detect bridging faults more easy than other vectors. This can be assumed, because inputs next to each other in the specification, are often also implemented next to each other.

Because these vectors have statistically proven to give in most cases better results than other test vectors, the first method will use these heuristics. That is why this method is named statistic and not random.

The algorithm is as follows:

Determine a transition which will be calculated next.
Compute the multiplicity for that transition.
Dependent on the multiplicity, determine the number of test vectors for this transition, with a minimum of 1 test vector.
The test vectors are chosen first by defining all don't care bits zero, then all don't care bits one, and after that alternating zero's and one's (two times). If more than four test vectors are needed, the others are generated randomly.

The question which remains is how the numbers of test vectors for a transition depends on the multiplicity. Because exhaustive testing is very time consuming, the number of test vectors must not be linear dependant on the multiplicity. That is why the following relation is chosen,

$$\text{number of test vectors} = C_1 * \log_2(\text{multiplicity}) \qquad (5).$$

In most cases the number of test vectors is dependant on the number of don't care bits in the input. For the constant in the formula 3 $C_1 = 2$ is chosen, because the important zero and one can be used, when the multiplicity is 2.

3.3. Local optimization

This method (method 2.) makes little use of the implementation on structural level.

A transition is defined as in formula (4). When a present state and an input vector are offered to the NSL (see figure 2.) the logic computes a next state. Stuck-at faults can influence this computation and this results in a faulty next state. For every transition there are stuck-at faults which influence the correctness of the transition. Counting these stuck-at faults gives a number, called Q_1. It tells how many stuck-at faults, when present, result in a faulty next state when this transition takes place. The assumption made for this method is that the more stuck-at faults can disturb a transition, the probability for this transition to be faulty is higher. Therefore more test vectors are needed to test the transition.

In a transition it is possible that a number of input bits are don't care. But it is also possible that when these input bits are defined, some test vectors detect more faults than others. Take for instance a transition which can be disturbed by 11 stuck-at faults, and has a multiplicity of 2. Assume that the multiplicity is caused by one don't care bit in the input. It is possible that the test vector which has the don't care bit defined as logic "0" will detect 10 stuck-at faults, while on the other hand the test vector with the logic "1" defined, detects only 8 stuck-at faults. In this example there is a clear difference between the number of stuck-at faults the test vectors detect. For the correct functioning of this method, it is also important to know how many stuck-at faults are tested by each input vector.

The second algorithm is as follows:

For every transition a factor Q_1 is determined. The number of test vectors to test this transitions is derived from this Q_1 as

$$\text{number of test vectors} = C_2 * Q_1 \qquad (6).$$

Choosing the test vectors for a transition, is done by first taking that vector which detects most stuck-at faults. The next vector is chosen with the next highest number of tested stuck-at faults, and so on, until the total number of test vectors calculated by (6) is reached. When a choice is to be made between two vectors, detecting the same number of stuck-at faults, this is done randomly.

In (6) the factor C_2 is to be determined. This will be done later. The relation between the number of test vectors and the factor Q_1 is chosen, because the number of stuck-at faults is a measure for the amount of hardware that is part of the transition. The relation (6) is chosen linear, to keep the complexity of the method low.

This method is named local optimization. For a transition those test vectors which detect the most stuck-at faults, are chosen. Locally, that means for one transition, this gives the best fault coverage. For the total machine this does not have to hold, because many test vectors can test the same stuck-at faults in different transitions. The next two methods take into account the global effects.

3.4. Static global optimization

The third method (method 3.) also uses information derived from the structural level description of the machine. All calculations take place at the functional level. Like the previous algorithm this method uses the factor Q_1, which gives the number of stuck-at faults for a certain transition. Another factor is introduced. A stuck-at fault can influence several transitions. The second factor will account for this.

Just like in the previous subsection the factors are calculated for transitions as for the possible input vectors belonging to such a transition. The factor belonging to the transition prescribes the number of test vectors for such a transition, while the factor for an input vector gives the quality for it as a test vector.

The factor which is important in this subsection, consist of two parts. The first one is the same as in the previous subsection. The second is formed by summing the numbers of transitions in which the stuck-at faults, which compose Q_1, occur. This factor is called Q_2. The importance of this second factor is to tell something about the appearance of stuck-at faults in other transitions, so they will not be tested twice.

The algorithm is as follows:

For every transition and for every input Q_1 and Q_2 are calculated. These two factors determine a quality factor, as

$$\text{quality} = C_3 * Q_1/Q_2 \qquad (7).$$

The number of test vectors for a transition is equal to the quality calculated for that transition. Which test vectors are chosen, depends on the quality belonging to each input vector. First choose the vector with the highest quality, second the vector with the second highest quality and so on, until the number of test vectors for that transition is reached.

In (7) C_3 has to be determined, but this will be done later.

This method is called static global optimization. Static because the factors are calculated before test generation starts, so during generation the factors stay constant. Global because the factor Q_2 involves other transitions that are not considered at that moment. Optimization, because the best vectors for a transition are chosen, to optimize the fault coverage of the tests.

3.5. Dynamic global optimization

The method in this subsection (method 4.) uses, just like the two previous mentioned methods, only one factor, namely the Q_1. Looking at the structural level there is a set of stuck-at faults which can disturb the behaviour of the sequential machine. Subsets can effect parts of the machine, like a transition. The factor Q_1 belonging to such a transition is the number of elements in the subset of stuck-at faults. At start this factor Q_1 is the same as in method 2.

The method is as follows:

> Choose the transition which has the highest Q_1, in other words which can be disturbed by the most stuck-at faults. Choose a vector which detects most stuck-at faults. Add this vector to the set of test vectors, and remove all faults, which have been tested for, from the set of stuck-at faults. This gives a new set of stuck-at faults. Starting with this new set recalculate all the factors Q_1. Start again by choosing the transition which has the highest Q_1, until all stuck-at faults are tested.

In this method it is again important that a difference is made between two factors, namely the factor for the transition and the factor for the input vectors. With the first factor a transition is chosen which is going to be processed, while with the second the next test vector is chosen.

The name for this method is now also clear. Global optimization like in the previous method and dynamic because after every generation of a test vector the whole machine is processed again.

3.6. Expected performance of the methods

There are a few test results which are important to say something about usefulness of the four test methods. First the amount of processing time each method uses, second the number of test vectors generated and third the amount of faults that is detected by the test vectors. This is called the performance. Before continuing with the performance two other things are important.

The four methods are all suitable to be programmed into a computer algorithm. The complexity of the methods increase from method 1. quite easy to method 4. very difficult. Also a trend is clear that method 1. uses no information of the structural level, to method 4. using the most information of the structural level.

Methods 2. to 4. have one big disadvantage. They use information about stuck-at faults, which has to be extracted from the structural level description. One of the main problems is that fault simulators consume a lot of time to calculate the sets of stuck-at faults which are detected by a certain input vector. The goal of the four methods is to shorten the test generation time. When it is necessary to first fault simulate the whole machine, it is possible that the decrease in generation time at the functional level is not rewarding any more after fault simulating at the structural level. The reason why these methods are still used, is first because fault simulators do not use backtracking which causes a decrease in processing time, and second because the idea may be extended to other levels of abstraction, from which it is easier to extract the necessary information.

3.6.1. Performance for statistic testing

The first method uses no information about the implementation of the machine on the structural level. It only assumes that with increasing multiplicity, the fault probability increases. This is the reason why the number of test vectors for a transition increases with increasing multiplicity. The relation between these two is given in formula 3.

In this formula the number of test vectors is related to the multiplicity as a logarithm. This means that for multiplicity of less than about 8, the number of test vectors is equal to the multiplicity. For small machines with a small number of input bits, this method thus resembles exhaustive testing. For large machines this does not hold. For example compare a machine with four input bits and a machine with 28 input bits. The first machine can have a transition with multiplicity of e.g. 8, the second with multiplicity of e.g. 2^{27}. For the first machine this

transition will be tested with 6 test vectors, while for the second machine 54 test vectors are generated for this transition.

Depending on the size of the machine, test generation will resemble exhaustive testing for small machines and give a very large reduction in the number of test vectors for large machines. The fault coverage will be high for small machines, because they are tested exhaustively. For larger machines it hard to say how the fault coverage will be, because of the random generated test vectors. The only thing is that always the statistical more interesting vectors will be chosen, so fault coverage can be high.

3.6.2. Performance for local optimization

Not taking into account the calculations of the factors Q_1, the method 2. is a little more complex than method 1. Because it chooses that vector in a transition which detects the most stuck-at faults, the fault coverage will be high for each transition. The disadvantage of this method is that it does not use information of other transitions, so it is possible that there will be a lot of test vectors, testing the same stuck-at faults. Also it is possible that because a choice has to be made between vectors which can detect the same number of stuck-at faults, the wrong vector can be taken. This means that fault coverage depends on choices. It will be high, because for every transition the fault coverage is high.

The number of test vectors needed to cause this high fault coverage depends on the factor C_2. The expectation is that the number of test vectors will be high to reach the coverage wanted, because no information about the other transitions is used.

3.6.3. Performance for static global optimization

The complexity for this method is larger than for the previous method. This is mainly caused by the calculations of Q_2. The factor Q_2 relates other transitions with the transition under consideration. The increase in complexity will also give better results.

The factor Q_2 expresses something about the appearance of stuck-at faults in other transitions. If for a transitions all the stuck-at faults, influencing the behaviour of that transition, appear in many other transitions, the factor Q_2 will be high. This means that the number of test vectors for that transition will be low, as expressed in (7). But because the stuck-at faults appear in many transitions, the probability that they will be tested is high. So less test vectors than in method 2. will be generated and the fault coverage will be high.

3.6.4. Performance for dynamic global optimization

Method 4. is bound to give the best results. The algorithm only stops when the fault coverage is 100.00%. The number of test vectors will be small, but does not have to be minimal. The reason for this is that when a choice has to be made between two transitions with the same factor Q_1, it is not possible to tell which transition is to be computed next to give the minimal number of test vectors. So the results are sub-optimal for sure and sometimes even optimal.

The disadvantage of this method is that after generating a test vector, all the factors Q_1 have to be recalculated. That is why the complexity of this algorithm is the highest of all.

3.7. Conclusions

Four methods for generating test vectors at the functional level, using information on the structural level, are introduced. The expected performance is estimated. It is clear that with increasing use of information, the complexity increases but also the performance improves. The only question which remains is that the time to extract information from the structural level will be short enough to profit from the concept on which the methods are based.

4. USING THE METHODS

4.1. Introduction

After the introduction of the four test methods, the methods themselves have to be tested. This is done to find out if the methods will give good results compared to the conventional test methods.

In this section the four methods will be considered, how they were applied. Then there will be said something about the results which were extracted, and about the machines that were used to apply the methods to. After this the conventional test algorithms will be explained, which will be compared to the new introduced methods. The constants C_2 and C_3 from methods 2. and 3. will be determined. At last the making of test sequences is mentioned.

During the work, some ideas about fault simulation were born. Fault simulation is not the real subject of this report, but it is important before test generation can start. Therefore these ideas are also mentioned.

4.2. Generating test vectors

In this subsection for all of the test methods the generation of the test vectors will be dealt with. Because the methods are new, there is not any computer program calculating any of these methods. Therefore the calculations are done by hand. This is labour-intensive, especially for the more complex methods.

4.2.1. Generation for statistic testing

The generation of test vectors for this method is explained in subsection 3.2. The method only uses the description of the machine at the functional level, which exists of transitions. These transitions are given in terms of state bits and input bits. An example for this is given in formula (4).

The multiplicity for a transition is determined. If the multiplicity is 1, then the calculations stop and the only input is chosen as test vector. For a multiplicity of 2, the two possible inputs are chosen or if the multiplicity is caused by a don't care, this bit is taken zero for the first vector and one for the second.

When the multiplicity is larger, the statistical more important bit patterns are used. As an example, take the transition in (4). The multiplicity is 8 and the number of test vectors for this transition is 6. Four of these test vectors are formed by the statistical bit patterns: 0000, 0111, 0101, 0010. The remaining two are generated random. These can be: 0110, 0100.

For the generation of the random vectors a simple program in pascal was used. The input was the number of don't care bits, and the outputs were the bit patterns which had to be filled in.

4.2.2. Generation for local optimization

This method uses information retrieved from the structural level. The information which is needed is to calculate the factor Q_t for transitions and for inputs. That is why the machine is fault simulated exhaustively at the structural level. Because only the NSL (see figure 2.) is considered, the inputs of this logic block consist of the present state bits and the input bits. The outputs of the block are the next state bits. This is exactly the transition defined in formula (4). This is why stuck-at faults that influence this transition, are very easy combined with the description on functional level.

The fault simulation is done exhaustively. This means that for a completely specified input a set of stuck-at faults can be found, which is detected by this input for this transition. When two sets of stuck-at faults are the same, it is tried to combine the two input vectors, resulting in a don't care bit. Counting the number of stuck-at faults belonging to such a set, results in the factor belonging to an input determining how well it will test this transition. Combining all the sets of stuck-at faults, gives the number of stuck-at faults which can influence the transition. This results in Q_t.

Test generation now is done like mentioned in subsection 3.3. The Q_t for a transition determines the number of stuck-at faults. The input vector which detects most stuck-at faults is chosen. Now it is clear why similar sets of stuck-at faults are combined, as mentioned above. When this would not be done, it is possible to choose two vectors, which detect the same stuck-at faults. By combining inputs, this can be prevented.

4.2.3. Generation for static global optimization

The third method also uses information from the structural level. Part of this information can be used from the previous method, because Q_1 is needed here as well. The calculations for Q_2 can be done manually, by counting how many times a stuck-at fault appears in the sets of stuck-at faults, belonging to the transitions. This increases the time needed for test generation.

With small changes to the fault simulator, this information is calculated automatically. The extended calculations result in sets of transitions, belonging to a certain stuck-at fault. Counting the number of elements in these sets, results in the number of transitions in which a stuck-at fault appears. With these numbers the calculations for Q_2 can be done.

After determining all Q_1's and Q_2's, the number of test vectors for a transition is calculated with formula (7). The test vector which is chosen, has the next highest quality factor. Like in method 2. the inputs for a transition are combined as much as possible, to prevent that vectors which detect the same stuck-at faults, are chosen.

4.2.4. Generation for dynamic global optimization

In the previous methods, the information used from the structural level were the sets of stuck-at faults and the sets of transitions. After the calculations of the factors Q_1 and Q_2 for every transition, these sets were not used any more. The fourth method uses the sets of stuck-at faults belonging to the transitions, and keeps using them during the generation of the test vectors.

The algorithm is explained in section 3.5. The test vectors are generated using the factors Q_1 for each transition, which are calculated as in method 2. After the generation of a test vector, all the sets of stuck-at faults are recalculated and also the factors Q_1.

The functional level description of the machine under test contains transitions. These transitions are put in a table, together with the sets of stuck-at faults which can influence this transition. Also the inputs with their own sets of stuck-at faults are included. The transition with the highest Q_1 is considered. After choosing the vector which detects most stuck-at faults, all sets of stuck-at faults are recalculated by removing the detected stuck-at faults from these sets. The sets of transitions belonging to the stuck-at faults, generated for method 2. are a good help in doing this. With the newly created sets, the factors Q_1 are also recalculated, and the procedure is repeated, until all sets contain no elements any more.

4.3. Fault coverage and number of test vectors

The reason for testing the methods 1. to 4., is to compare them to the conventional test methods. There are two figures that say something about the performance of a method: the number of test vectors and the fault coverage. The number of test vectors can be found quite easy. They are counted after generation. The fault coverage is a little more complicated.

Starting the fault simulation a set of faults is defined. Here the fault model for the structural level is chosen to be the single stuck-at fault model. This means that when the machine under test is defined, also the set of stuck-at faults is known.

After fault simulation it is clear which stuck-at faults do not effect the functional level behaviour of the machine. These faults are not included in the calculation of the fault coverage. It is also clear which vectors detect which faults. This is used after testing to calculate the fault coverage.

From the set of test vectors which is generated for a method, one test vector is taken. The stuck-at faults detected by this vector are marked "detected". This is done for all test vectors. The number of "detected" stuck-at faults is related to the total number of stuck-at faults effecting the proper operation of the machine. This results in the fault coverage, expressed as a percentage.

4.4. The machines

To test the methods introduced in this report, they have to be applied to a machine. For comparison purposes not one but four machines were used. In this subsection these machines will be considered. In appendix A. the machines are described.

The starting point for the machines was a description at functional level (appendix A), consisting of a state transition table. The state and input assignment had already been done, so the only thing left to do was designing them at the structural level.

A computer program was used to design the machines at structural level. This program, called MOM (multiple output minimizer), minimizes the number of gates by using logic gates necessary for one part, for other parts of the machine as well. The result is an AND-OR realisation, with the number of inputs equal to the

number of state bits and input bits, the number of outputs equal to the number of state bits.

Table 1. Characteristics of the machines under test

	Sequential machines			
	EX4	EX6	MARK1	OPUS
states	18	8	13	10
state bits	5	3	4	4
input bits	6	5	5	5
transitions	44	31	33	30
gates	26	26	21	23
real transitions	1152	248	416	320
s-a-faults	322	278	238	258
non funct. sa-flts	28	2	13	15
funct. sa-flts	294	276	225	243

The most important parameters of the four machines are put together in table 1. The machines have got names, so they are easily addressed.

For all the parameters in table 1. the prefix "the number of" has to be added. The explanation of the parameters is:

-"states" expresses how many states are defined at the functional level;

-"state bits" is equal to the number of memory elements. There is one machine (EX6) which has a complete state specification. The other machines have more states implemented at the structural level than were defined at the functional level;

-"input bits" will need no explanation;

-"transitions" gives the possible transitions at the functional level, not taking into account the multiplicity for that transition;

-"gates" is the number of AND- and OR-gates at the structural level calculated by the program MOM. These are implemented in the NSL to compute the transitions;

-"real transitions" are all transitions possible at the functional level, taking into account the multiplicity of all transitions. The transitions together form the set of input vectors from which test vectors can be chosen;

-"sa-faults" are the stuck-at faults which can occur in the structural level description of the machine;

-"non funct. sa-faults" are the stuck-at faults that do not influence the behaviour of the machine at structural level;

-"funct. sa-faults" are the stuck-at faults with which the fault coverage is measured. These faults can effect the proper operation of the machine.

Three of the machines, EX6, MARK1 and OPUS are small sequential machines, so in reality testing them exhaustively has to be considered. Because EX4 has 18 states, test generation will have higher priority. To compare the test methods these machines satisfy.

During fault simulation the set of faults was decreased by fault collapsing. This means that e.g. a sa0 at the input of an AND-gate can not be distinguished from a sa0 at the output. Therefore these equivalent faults are only calculated once, to decrease processing time.

The program MOM designs the machines in a AND-OR implementation. This means that when an input is offered to the NSL, this line fans out into several gates. Fault simulations did not take into account this fan-out. This means that the multiple faults were not computed for inputs. The outputs of the AND-gates however could also fan-out to different OR-gates. The multiple faults created by this fan-out were considered.

4.5. How to compare the methods

To express the performance of the test methods introduced in this report, two figures are important, the number of test vectors and the fault coverage. To compare these test methods with conventional methods, these conventional methods have to be applied to the defined machines as well.

There are two levels discussed here, the structural level and the functional level. As mentioned in section 2. there are a few methods known which generate test vectors at the structural level. For the functional level only exhaustive testing is mentioned.

4.5.1. Structural level comparison

Three of the four methods introduced in this report use information retrieved from the structural level. Test generation however takes place at the functional level. Four machines were tested using the new test generation algorithms and a conventional structural level algorithm, PODEM.

However, the program that implements PODEM does no accept more than sixteen inputs. During fault simulation, all the inputs of the AND-gates are considered as the inputs, not as fan-out from the real inputs. This means that an input with a fan-out of e.g. twelve, is split up into twelve different inputs, all having

the same logical value. For the PODEM several tricks are used, to let the program calculate the machine. This introduces extra AND-gates with one input and one output. Each gate implies four extra stuck-at faults (two times a sa0 and a sa1). The extra AND-gates were only used as a sort of buffer. With fault collapsing however there are no extra faults introduced. Therefore the results can be compared. But because the number of stuck-at faults for the PODEM-input is just a little higher than for the fault simulations, the comparison of fault coverage is not exact, but satisfactory.

4.5.2. Functional level comparison

From the four methods, method 1. uses no information about the implementation at the structural level, while the other three do use this information. This means that this method can be qualified as a functional level test generation algorithm. The other three methods do use this information. Method 2. uses little and method 4. uses much information. Method 4. is thus the algorithm combining the structural and the functional level most.

In order to check the concept of test generation by using structural level information at the functional level, the following comparison is made. Because method 4. uses most of the structural level information, these results will be compared to the result of method 1. The only restriction however is that the number of test vectors used for method 1. will be the same as the number of test vectors generated by method 4. This is done to see how well the use of structural level information improves the test results.

4.6. Determination of C_2 and C_3 in methods 2. and 3.

In the methods 2. and 3. there are two constants, C_2 and C_3, which are not determined yet. This will be done here.

4.6.1. Determining C_2

The number of test vectors which have to be generated for a transition depends on the factor Q_t and the constant C_2, defined in section 3.3. The factor Q_t is calculated after exhaustive simulations. The constant C_2 has to be defined as to give the best fault coverage for the entire machine. The dimension of C_2 is the number of test vectors per number of stuck-at faults.

The sequential machines considered are medium sized. This will give a different value for the constant C_2, than if large or small machines would be considered. Because nothing is known about the average number of stuck-at faults a test vector detect, an assumption is made of about 5. If this number is taken 1, which means one test vector for one stuck-at fault, the generation would result in as many test vectors as there were stuck-at faults. This would result in to many test vectors, so 5 is expected better.

The simulations resulted in the factor Q_1 for every transition. This factor was about 10 to 20 for every of the four machines. Now it is easier to determine C_2. This is taken three values, $C_2=1/4$, $C_2=1/5$ and $C_2=1/6$. This result in three new methods for local optimization, method 2.1, 2.2 and 2.3 respectively.

A note is to be made for this choice. Because the machines are medium sized, this values will not be suitable for other kinds of machines, because for larger or smaller machines the average number of stuck-at faults detected by a test vector will be different. For the machines under consideration, the factors are expected to give good results.

4.6.2. Determining C_3

The method 3. is developed, to create less test vectors than there were generated by method 2. The factor resulted by dividing Q_1 and Q_2 (formula (7)), was about 0.10. Therefore the factor C_3 was chosen three values, $C_3=20$, $C_3=25$ and $C_3=30$, called method 3.1, 3.2 and 3.3 respectively. For method 3.1 and 3.2 the number of test vectors calculated by formula (7), was rounded to a whole number, while for method 3.3 the number of test vectors was taken as the highest integer less or equal than the factor resulted. This last is done to minimize the number of test vectors.

A fourth method was developed, named method 3.4. The three methods mentioned earlier, assumed that the factor C_3 was constant. As in statistic testing (method 1.), this method assumes that the number of test vectors is dependent on the multiplicity of a transition. Therefore a new type of multiplicity has to be introduced.

A transition is composed of a present state, a next state and an input (3). After specifying the machine, it is possible that there are more possibilities to transit from one state to another. This is called multiplicity. At the functional level don't cares are introduced to denote multiplicity, if it does not care what value an input has for that transition.

These don't care bits can also be used in the realisation at the structural level. If this is done, it means that the hardware becomes simpler. This is the starting point for the method 3.4. Assume that the higher the multiplicity for a transition, the less hardware is needed to realise the transition. With less hardware this also means that the fault probability for that transition decreases, so less test vectors are needed to test the transition. Therefore C_3 in method 3.4 will be calculated as

$$C_3 \sim 1/\log_2(\text{multiplicity}) \qquad (8).$$

In (8) $\sim$ means "is related to".

The multiplicity for the machines under test is mostly caused by don't cares, the number of input bits is also used to calculate C_3. Eventually the number of stuck-at faults per test vector has to be assumed, and this is taken 6. The resulting C_3 is

$$C_3 = 20 * \text{noib} / \text{nomult} \qquad (9).$$

in which noib means number of input bits and nomult is $\log_2$(multiplicity).

4.7. Making of test sequences

Applying test vectors to a sequential machine has to be done in a certain sequence. Test generation for combinatorial circuits is done easier, because the sequence of applying the test vectors is not important. For sequential machines scan technique makes use of this advantage, because the state which the machine is in, can be shifted into the memory elements. But the advantage of the test methods introduced in this report is that they treat a sequential machine as a real sequential machine. This means that the state of the machine can only be reached by applying a sequence of inputs from a certain starting state, to the state required. This means that the test vectors have to be applied in a certain sequence.

The four methods used do not take into consideration the generation of sequences of test vectors. This has to be done after the test generation. At the European Design Automation Conference in Glasgow, C. Jay [4] published an algorithm that generates a sequence of vectors, starting from single transitions. After the generation of test vectors by the four methods, the algorithm can be used to generate test sequences. These sequences can be applied to the sequential machines.

4.8. Fault simulation

(Note: This paragraph is not really necessary for this report)

Just like for test generation, computer algorithms have been developed to do fault simulations. As denoted in section 3.6. these fault simulators consume a lot of processing time. That is why some attention was paid to fault simulation. Some ideas followed, but a real algorithm was not developed. In this section a short impression of the ideas is given.

The starting point for the fault simulation is the description of the machine at the structural level. The fault model assumed is the single stuck-at fault model. Attention is paid to the NSL only.

Conventional methods are concurrent, parallel and deductive fault simulation. The equivalence of these methods is that they all work from input to output. The inputs are applied to the logic and sets of stuck-at faults for each gate are computed. During the calculations a lot of sets of stuck-at faults is needed to give the resulting set of stuck-at faults, which for a sequential machine disturb the transition. The first idea is not to work from inputs to outputs, but from outputs to inputs.

Given a correct functioning logic block, say the NSL in a Moore-type sequential machine. Apply a present state and a fully specified input to the NSL. Calculate every output of each gate so the next state will be clear. Now the gates will be marked with one of three terms: ACTIVE, ALMOST ACTIVE and NOT ACTIVE.

The definition for each term is:

ACTIVE:
- a (N)OR-gate is active if only one input is logic "1";
- a (N)AND-gate is active if only one input is logic "0";

ALMOST ACTIVE:
- a (N)OR-gate is almost active if all inputs are logic "0";
- a (N)AND-gate is almost active if all inputs are logic "1";

NOT ACTIVE:
- a (N)OR-gate is not active if more than one input is logic "1";
- a (N)AND-gate is not active if more than one input is logic "0";

A failure in this method is that it will work properly for fan-out free logic. When the NSL, which is a combinatorial circuit, contains fan-out, the method proposed here, will not work properly without justification. The expectation is that these justification is only small. To explain the algorithm, the bad assumption of a fan-out free NSL is made.

When a signal line has a certain logic value, the stuck-at fault influencing this signal has the opposite value. This means if a signal line carries logic "1" ("0"), the signal is disturbed by a sa0 (sa1). This will be called the opposite fault.

There are two things which need a different explanation, fault excitation and fault propagation. First fault excitation will be dealt with.

The output of a gate whether it is ACTIVE, ALMOST ACTIVE or NOT ACTIVE, is disturbed by its opposite fault. This means that for a certain transition in the NSL the set of stuck-at faults for that transition contain the opposite faults of the outputs. If a gate is marked ACTIVE, the opposite fault for the input which caused that output is also included in the set of stuck-at faults. If a gate is marked ALMOST ACTIVE, all the opposite faults for the inputs are included. If a gate is NOT ACTIVE, no input with opposite faults have to be included.

Second fault propagation will be considered. If a gate is marked ACTIVE, the only way it will propagate a stuck-at fault is via the input which caused the gate to be ACTIVE. If a gate is marked ALMOST ACTIVE, all the stuck-at faults having been propagated to the inputs of this gate, will also propagate through this gate. If a gate is marked NOT ACTIVE, there will be no stuck-at fault which can propagate through this gate.

Because the idea for this algorithm is to work from outputs to inputs, the propagation is to be denoted different. A better word would be the consideration. Because a gate does not have to propagate stuck-at faults via all of its inputs, only that part of logic has to be considered which is connected to those inputs.

The justification for not fan-out free logic is assumed not to be difficult. By finding the paths from a fan-out point to the outputs, it is possible to find out how signal can propagate through the gates. This means that also the effects of stuck-at faults can propagate via these paths. This finding of paths can be done before the fault simulation starts, because these paths do not change during the simulation.

The algorithm is repeated shortly:

> Specify an input and a present state. Calculate all the signals in the NSL. Mark all gates as mentioned above. Consider one gate at the output and use the excitation rules which refer to the mark of the gate. Use the propagation rules to determine which parts of logic have to be processed further. Repeat these rules for all gates which have to be considered. Repeat all these calculations for every input and every present state.

Until so far the basic algorithm, of which an example is given in appendix B. An extension may be made by not totally specifying the inputs, but accepting don't

cares. The reason for this is explained in section 4.2. There it says that sets of stuck-at faults detected by two inputs can be the same, so the inputs may be put together, introducing a don't care. This idea is not worked out further, so it is only mentioned.

4.9. Conclusions

In this section it becomes clear that it is not possible to compare the processing time of the four test methods, because they are all calculated by hand. There are however two important figures saying something about the performance of a method. These figures are the number of test vectors generated and the fault coverage.

The four introduced test methods are all based on the concept to use structural level information to generate test vectors at the functional level. With increasing use of information the complexity of the method increases. To find out if the concept holds, the methods are compared to conventional test methods. For the structural level test generation the PODEM algorithm is chosen, while for the functional level method 1. is considered.

5. RESULTS OF THE TESTING METHODS

5.1. Introduction

In this report a new concept of test generation for sequential machines is introduced. Four test methods were defined and were used on four machines. To find out if the concept improves test generation, the results of these methods have to be compared to the results of the conventional test methods. In this section the results of the test generation of the new methods and of the conventional methods will be given. After that a comparison will be made between these results, to find out if the concept improves test generation.

5.2. Tables of results

Table 2. Test results for each machine

EX4									
	tm1	tm2.1	tm2.2	tm2.3	tm3.1	tm3.2	tm3.3	tm3.4	tm4
rntv	35.24	15.10	11.46	10.59	4.77	5.12	4.60	5.82	2.87
cov	100.00	100.00	100.00	100.00	96.94	96.94	96.26	96.94	100.00
EX6									
	tm1	tm2.1	tm2.2	tm2.3	tm3.1	tm3.2	tm3.3	tm3.4	tm4
rntv	72.58	41.94	33.87	27.02	31.45	34.27	34.27	46.77	13.31
cov	99.64	98.18	96.01	95.65	95.65	95.65	95.65	97.07	100.00
MARK1									
	tm1	tm2.1	tm2.2	tm2.3	tm3.1	tm3.2	tm3.3	tm3.4	tm4
rntv	50.96	24.52	18.75	16.83	10.58	12.98	11.78	13.46	7.21
cov	100.00	100.00	100.00	99.56	99.11	99.11	99.11	99.11	100.00
OPUS									
	tm1	tm2.1	tm2.2	tm2.3	tm3.1	tm3.2	tm3.3	tm3.4	tm4
rntv	55.63	23.75	20.00	15.63	12.19	15.00	15.31	17.19	12.50
cov	97.94	96.71	96.71	96.30	95.06	94.24	95.88	96.30	100.00

In table 2. the results are given for the calculations of the four test methods. In table 3. the results are given for the comparison of the introduced concept and the conventional test generation algorithms.

Table 3. Test results for the concept and the conventional methods

EX4			
	tm4	tm14	podem
rntv	2.87	2.87	4.17
cov	100.00	76.53	100.00
EX6			
	tm4	tm14	podem
rntv	13.31	13.31	16.13
cov	100.00	76.45	99.73
MARK1			
	tm4	tm14	podem
rntv	7.21	7.21	9.85
cov	100.00	74.22	100.00
OPUS			
	tm4	tm14	podem
rntv	12.50	12.50	16.45
cov	100.00	71.19	99.73

In the two tables there are a few things that will be explained. The tables are divided in results per machine. This is to make comparison easier. The abbreviations used in the tables mean: tm3.2 = test method 3.2, rntv = relative number of test vectors, cov = fault coverage. The last two figures are each expressed as a percentage. The number of test vectors is related to the number of real transitions as mentioned in table 1. The fault coverage is taken relative to the number of functional stuck-at faults, also mentioned in table 1. The only exception for this is the fault coverage for PODEM, because it has its own set of stuck-at faults as mentioned in 4.5.1.

5.3. Analyzing the methods 1. to 4.

The next section is related to table 2.

5.3.1. Results for statistic testing

The machines that were tested by the four methods, were small or medium sized machines. Therefore the method 1. will generate almost as much test vectors as for exhaustive testing. Only EX4 gives a different result, but the other machines are tested by half or more of all the possible vectors. This is comparable to exhaustive testing. The reason why EX4 differs, is because it is the largest machine.

The fault coverage for the machines is high. If a second time test vectors are generated, the fault coverage will be different, because of the random choices of test vectors. The reason that the fault coverage is high is first because the method resembles almost exhaustive testing and second because the statistic more important vectors were used. Because the machines are small to medium sized, the multiplicity of the transition is low, so the main part of the test vectors consists of these statistical vectors. For larger machines the fault coverage will not be this high, because the random technique will influence the fault coverage more.

5.3.2. Results for local optimization

The reason to use the number of stuck-at faults as measure for the number of test vectors, resulted in less test vectors than generated by the method 1. Still the number of test vectors is high for all the machines, because a lot of vectors will test the same stuck-at faults.

For method 2. the constant C_2 had to be determined. For method 2.1 this was chosen to be $C_2 = 1/4$, method 2.2 had $C_2 = 1/5$ and method 2.3 used $C_2 = 1/6$. Dependent on this constant C_2, the number of test vectors was calculated. If C_2 decreases, also the number of test vectors for a transition decreases. Therefore the total amount of test vectors decreases. This is well noticed in the table 2.

For the fault coverage the decrease of test vectors also results in decreasing fault coverage. For EX4 it makes no difference which test method is used, the fault coverage is 100.00%. For the other machines however, this decrease is clearer.

The question remains if this increase in test vectors is worth the increase in fault coverage. For EX6, an increase of more than 8% of test vectors, results in only 2% more detected stuck-at faults (compare method 2.2 to 2.1). This may be worth if a high fault coverage is required. When the testing time has to be short, method 2.2 is preferred.

This method generates relative the most test vectors for EX6. For the other machines, the relative number of test vectors is about 15%, while for EX6 this number is about 35%. This difference will be explained in the section about implementation.

An assumption was made about the average number of stuck-at faults a test vector would detect. Looking at the results, this number increases with increasing size of the machine. For EX6 and OPUS (smallest machines, looking at the number of real transitions) the average number of stuck-at faults per test vector

of about 5 is a good choice, while for EX4 and MARK1 this figure can be higher for a good fault coverage.

5.3.3. Results for static global optimization

As expected the methods 3.1, 3.2 and 3.3 generate less test vectors than the method 2. This is caused because information of other transitions is used. The fault coverage however is less than that for method 2., but this decrease can be neglected considering the decrease of the amount of test vectors.

For the second time EX6 differs in results from the other three machines. For EX6 the relative number of test vectors is about 35%, while for the other machines this number is about 10%. The reason for this will be explained in the section about implementation. As mentioned in section 3.6.3. the method 3. would generate less test vectors than the method 2. For three machines this holds, but for EX6 this is not true.

Three choices were made for the constant C_3. The quality (7) calculated with the first two, $C_3 = 20$ and $C_3 = 25$, is rounded to an integer to determine the number of test vectors. For the third, $C_3 = 30$, the quality was taken as the highest integer less or equal than the quality to determine the number of test vectors. For all the methods the ratio Q_1/Q_2 for each transition is the same. This means that for the methods 3.1 and 3.2 the transitions with this ratio low, get assigned a test vector easy, while for the method 3.3 this ratio has to be higher. For transitions with a higher ratio Q_1/Q_2, the number of test vectors will be high for the method 3.3, while for the method 3.1 and 3.2 the number of test vectors will be less. Looking at the results in table 2., EX4 has many transitions with this factor low. That is why method 3.3 generates the least test vectors, compared to method 3.1 and 3.2. For EX6 and OPUS the most transitions will have a high Q_1/Q_2. MARK1 gives a good balance between the appearance of high and low ratios.

The method 3.4 gives a higher fault coverage than the methods 3.1 to 3.3, but it also generates more test vectors. The reason for this is that for most transitions multiplicity is caused by don't cares. The assumption for this method seems thus right. But the program MOM did not fully make use of the don't cares. Because the program MOM tries to minimize the number of gates, parts of hardware are shared between different parts of the machine. If a transition has an input vector containing one don't care, this can be implemented in such a way that it is redundant. This means that the defined bit on the don't care position certainly can have effect on the transition, while it is defined as don't care. This is why this method is not useful.

5.3.4. Results for dynamic global optimization

The fourth method gives the best results. The fault coverage is maximum, while the number of test vectors is low. As mentioned in subsection 3.6.4. the number of test vectors is minimal or close to minimal. For the four methods introduced here, the number of test vectors is minimal.

One exception of this minimal amount of test vectors is OPUS. Test method 3.1 generates less test vectors than test method 4., but the fault coverage of method 4. is 100.00%, while that of method 3.1 is less. This means that also for OPUS method 4. gives the best results.

5.4. Comparing the new concept to the conventional methods

Of the four methods method 1. uses no information and method 4. uses the most structural level information in generating test vectors. They both generate test vectors on the functional level. PODEM however makes use of only the structural level. For comparison, the method 1. is also considered to be a conventional functional level test generation algorithm. The next section is related to table 3.

5.4.1. Functional level comparison of the concept

Because method 1. does not use any information of the structural level and because it resembles exhaustive testing for the machines that were tested, this method was used to compare the concept to.

The way method 1. was compared to method 4., was to restrict the number of test vectors method 1. could generate, to the number of test vectors generated by method 4. The generation took place as to give each transition the same chance of being chosen as test vector. Don't cares were filled in with the statistic more important patterns. The results for this method called method 14 are put in table 3.

The restriction for test method 14 causes a low fault coverage. The reduction in the number of test vectors is 5 to 10 times, while the fault coverage is still about 75%. The reason that the fault coverage does not decrease 5 to 10 times is because the input belonging to a transition, already detects many stuck-at faults, while the don't care bits are not even determined. This means without specifying the don't care bits, many stuck-at faults belonging to the transition are already detected. The don't care bits for the method 1. are specified with the statistic

more important vectors, while for method 4. the specification for that vector is done as to detect the most stuck-at faults which are still to be tested.

The test vectors generated for method 14. detect, even without the don't cares specified, already a lot of stuck-at faults. With the statistic more important test vectors, the fault coverage can only improve, but the low number of test vectors allowed, result in a bad fault coverage.

5.4.2. Structural level comparison of the concept

The structural level test generation algorithm used, was the PODEM algorithm. This algorithm used a slightly different machine than the four test methods of this report. Therefore the set of stuck-at faults which had to be tested was larger. But also the stuck-at faults which are redundant for the functional behaviour of the machines are considered, as explained in subsection 2.5. PODEM will try to test all stuck-at faults, while the four methods of this report do not concern about non functional stuck-at faults.

The results for the PODEM test generation were close to the results of method 4. The fault coverage was almost maximal, while the number of test vectors were small. But for EX4 the number of test vectors generated by PODEM is much more than were generated by method 4. This could mean that for large machines the method 4. generates less test vectors than PODEM does. For go-no go tests method 4. is thus preferable, because it decreases testing time.

5.5. Implementation

The results for EX4 are the best for all the four test methods, compared to the other three machines that were considered. The relative number of test vectors are the least for all the methods, while the fault coverage is always high. There are a few reasons that this can be explained.

The first reason is that EX4 is a large machine compared to the others. It has 18 states, while the next largest machine has 13. At the structural level this means that the number of state bits for EX4 is at least 5, while for the other machines 4 or even 3 (for EX6) state bits are sufficient.

The inputs for a sequential machine are normally dependent on the environment of the machine. If the machine is part of a larger circuit, the inputs can well be

outputs of another sequential machine. This means that the inputs can normally not be assigned as to give the minimal implementation.

Defining a sequential machine normally means working with a given input assignment. The only thing that can be influenced, is the state assignment. For the machine EX4 6 input bits are defined. Because in every state all inputs are allowed, this means that there are 18 states times 2^6 inputs gives 1152 real transitions (table 1.). For the other machines the number of real transitions is much less. This explains why EX4 gives such good results for all the test methods. The number of test vectors is namely related to the number of real transitions.

Looking at the implementation further, a second thing is noticed. The number of gates for EX4 is not the largest, because also EX6 is implemented with 26 gates. Looking at the number of test vectors generated by method 4. shows that EX4 is tested with 33, while EX6 needs 32. These results are almost the same, while the relative number of test vectors differs, because the number of real transitions differs.

Still this does not really explains the reason why EX4 only needs about 3% of all the possible test vectors. This can be explained by looking at the inputs for both machines further. The number of transitions for EX4 is 44, while for EX6 this is 31. The number of input bits for both machines is 6 and 5 respectively. EX4 has one bit used to reset the machine, while EX6 does not. Furthermore the number of states for EX4 is 18, while for EX6 this is 8. EX4 can be resetted from each state, using one input bit. This already causes 18 transitions and leaves just 26 others. This means that from each state there are only two or three possible transitions. For EX6 this number is much larger. There are 31 transitions and 8 states, which means an average of about four transitions per state.

For EX4 the transitions have a large multiplicity. This multiplicity is caused by don't cares. Except for the bit that is used to reset the machine, all the bits are only used in two to six transitions. For EX6 all the input bits are used for more transitions.

A stuck-at fault in a machine can have effect on a transition in two ways. The first one is because the logic for that transition contains the fault, the second, because the rest of the logic contains the fault. For the AND-OR implementation this can be made clear as follows:

> An AND-gate is used to make sure that the next state bit will be logic "1" for that transition. A sa0 in this gate will cause a fault. This is the first type.
> If the next state bit is to be logic "0" for the transition, a sa1 in an AND-gate, which is implemented for another transition can cause a faulty next state. This is the second type.

A transition consists of a present state, an input and a next state. Assigning these results in a bit pattern. Take for example two transitions:

```
100   -0--   001        (10),
101   -01-   011        (11).
```

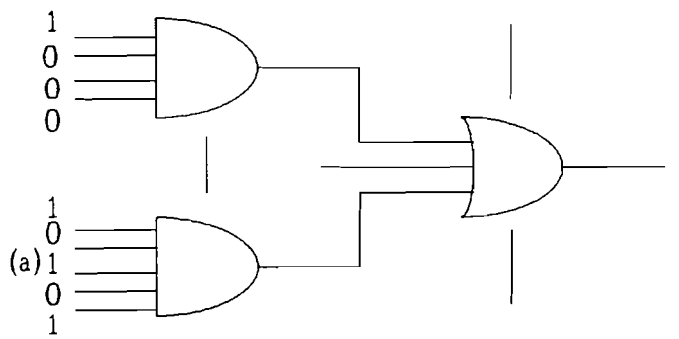

Figure 4. Part of the implementation of a sequential machine

Part of the implementation for these two transitions is given in figure 4. The implementation is given in an AND-OR fashion. The output is the second next state bit. If transition (10) is tested, the input vector used can be e.g. -00-, but also -01-, in which '-' means don't care. The second vector will also detect the stuck-at fault in which line (a) is sa0, while the first vector will not. This is also a kind of multiplicity, but now for test vectors. This multiplicity is already introduced in the generation of the test vectors for the methods 2., 3. and 4., but was not called multiplicity before.

EX4 has many input bits which are not used in many transitions. Therefore the multiplicity of test vectors explained before, has not a great effect on EX4. Because EX6 uses many input bits in many transitions, this type of multiplicity appears more. Also this multiplicity for one transition appears more, which means that EX6 has more transitions in which stuck-at faults are detected by different test vectors. This means that transitions in EX6 have to be tested more on the average than in EX4.

EX6 uses three state bits to implement eight state. This means that EX6 is fully specified, considering the states. EX4 however only has 18 states, but 5 state bits. This means that EX4 has 32 states realised. Therefore the effect explained in figure 3. in which logic for different transitions resembles, can be decreased with the right state assignment.

Continuing with this reasoning and taking into account the method developed by Devadas [3], a good state assignment is very important. On the one hand it

can lead to smaller implementation, while on the other hand it can facilitate test generation.

5.6. Conclusions

The method 1. uses no information of the structural level implementation. This is the least complex method. Method 4. however uses most of this information and is the most complex method. The results for method 4. are the best, if number of test vectors and fault coverage are considered.

Also compared to the conventional test algorithms, the concept of using structural level information at the functional level gives better results. PODEM comes with coverage near the results of method 4., but the number of test vectors is higher and the complexity is larger because it uses backtracking. Method 4. does not generate test vectors that detect redundancies, while PODEM does. This is an advantage of this concept.

6. CONCLUSIONS

The concept checked in this report is to use the structural level information for functional level test generation for sequential machines. The reason to do so is first to find a good fault model, second to avoid the testing of redundancies and third to avoid testing faults more than once.

Four methods were described, from which the first uses no structural level information, to the last that uses the most information. The complexity of the methods increases with increasing use of the structural level information.

The results that are used for comparison, are the number of test vectors generated and the fault coverage. With increasing use of the structural level information, these two results improve. The method that uses most of the structural level information, gives the smallest number of test vectors, while the fault coverage is maximal.

To find out if the concept holds the methods are also compared to the conventional test generation algorithms. The method which is the best representative for the concept uses the most structural level information. This method also gives better results than the conventional functional and structural level test generation algorithms. PODEM gives the coverage which is close to the coverage from the method considered, but it uses more test vectors. For the largest machine that was checked (EX4) it uses almost two times more test vectors.

The complexity of the concept is for a great deal dependant on how hard it is to extract the information from the structural level description. In this work this was done by exhaustive fault simulation of the machine under test, but perhaps simpler methods can be found. PODEM, however, uses backtracking to generate test vectors. It is well possible that even with exhaustive fault simulation the complexity for the introduced methods is smaller than the complexity of the PODEM algorithm.

The concept of using structural level information for functional level test generation gave for all the considered examples better results than both the pure functional level testing and the pure structural level testing.

REFERENCES

[1]. Lisanke R.
LOGIC SYNTHESIS AND OPTIMIZATION BANCHMARKS,
USER GUIDE, Version 2.0.
Microelectronic Centre of North Carolina, USA,
December 16, 1988.

[2]. Gupta, A.K. and J.R. Armstrong
FUNCTIONAL FAULTMODELING AND SIMULATION FOR VLSI DEVICES.
IEEE Design Automation conference,
proceedings 1985, p. 720-726.

[3]. Devadas, S., H.T. Ma, A.R. Newton and A. Sangiovanni-Vincentelli
A SYNTHESIS AND OPTIMIZATION PROCEDURE FOR FULLY AND EASILY TESTABLE SEQUENTIAL MACHINES.
IEEE Transactions on Computer-Aided Design,
Vol 8, no 10, october 1989, p. 1100-1107.

[4]. Jay C.
EXPERIENCE IN FUNCTIONAL-LEVEL TEST GENERATION AND Fault coverage IN A SILICON COMPILER.
European Design Automation Conference
proceedings 1990, p. 485-490.

[5]. Fujiwara H.
LOGIC TESTING AND DESIGN FOR TESTABILITY
London: The MIT press
ISBN 0-262-060960-5

APPENDIX A. FUNCTIONAL LEVEL DESCRIPTION OF THE MACHINES UNDER TEST

The four machines that were used to test the introduced methods, are described in this appendix. First the state assignment is given, which contains the number of state and the number of state bits as expressed in table 1. The description of the machines is at the functional level and consists of the state transition table.The transitions are of the form expressed in formula (3). In the transitions * means that the state is don't care.

EX4

State assignment		Transitions							
1	00000	*	0-----	1		6	1---1-	16	
2	10100	1	1-----	3		5	1---0-	17	
3	01110	3	1-----	2		6	1---0-	17	
4	01111	4	1-----	2		16	1-----	18	
5	01100	2	1-----	7		17	1-----	18	
6	01101	7	1-----	9		18	1-----	8	
7	00100	9	10----	9		8	10----	8	
8	00110	9	11----	14		8	11----	12	
9	10010	14	1-----	15		12	1-----	13	
10	10000	15	1-1---	10		13	1----1	4	
11	10001	15	1-0---	11		13	1----0	6	
12	10110	10	1-0---	3					
13	01010	11	1-0---	3					
14	11000	10	1-10--	3					
15	11100	11	1-10--	3					
16	11110	10	1-11--	5					
17	11111	11	1-11--	5					
18	01000	5	1---1-	16					

EX6

State assignment		Transitions							
1	011	1	11---	3		5	--0--	2	
2	001	1	00---	2		5	--11-	8	
3	111	1	10---	4		5	0-10-	5	
4	000	2	0-0--	2		6	----1	2	
5	101	2	--1--	5		6	10--0	4	
6	100	2	110--	3		6	00--0	2	
7	010	2	100--	4		6	11--0	3	
8	110	3	10---	4		6	01--0	6	
		3	00---	2		7	--0--	2	
		3	11---	3		7	101--	7	
		3	01---	6		7	011--	6	
		4	010--	6		7	111--	3	
		4	--1--	7		7	001--	2	
		4	110--	3		8	101--	7	
		4	000--	2		8	--0--	2	
		4	100--	4		8	0-1--	8	
		5	1-10-	8		8	111--	3	

MARK1

State assignment		Transitions					
state1	1001	*	0----	1	8	1----	14
state3	1011	1	1----	3	9	1----	14
state4	0100	3	1----	4	10	1----	11
state5	1101	4	1-111	13	11	10---	13
state6	1100	4	1-110	10	11	11---	12
state7	0011	4	1-10-	9	12	1----	13
state8	0110	4	1-011	8	13	1----	14
state9	0001	4	1-010	7	14	1----	3
state10	1010	4	1-001	6			
state11	1110	4	1-000	5			
state12	0010	5	1----	14			
state13	1000	6	1----	14			
state14	0111	7	1----	14			

OPUS

State assignment		Transitions					
init0	0000	*	--1--	init0	IOwait	01001	RMACK
init1	0100	init0	--0--	init1	IOwait	11001	WMACK
init2	1010	init1	--00-	init1	IOwait	--01-	init2
init4	0010	init1	--01-	init2	RMACK	--0-0	RMACK
IOWAIT	1000	init2	--0--	init4	RMACK	--0-1	read0
RMACK	1001	init4	--01-	init4	WMACK	--0-0	WMACK
WMACK	0001	init4	--00-	IOwait	WMACK	--0-1	write0
read0	0110	IOwait	0000-	IOwait	read0	--0--	read1
read1	1100	IOwait	1000-	init1	read1	--0--	IOwait
write0	1110	IOwait	01000	read0	write0	--0--	IOwait
		IOwait	11000	write0			

APPENDIX B. EXAMPLE OF FAULT SIMULATION AS IN 4.8.

In this appendix an example will be given of the fault simulation introduced in section 4.8. The function to be implemented is

$$f = x.y'.z' + x.y.z \qquad (B.1).$$

In this formula x,y,z and f are logic variables, negation is denoted by ' and . and + are the logic operators AND and OR respectively. Apply the vector x=1, y=0 and z=0. The implementation and the applied vector are drawn in the next figure. In this figure the gates are named with capitals, while the signal lines are named with small letters.

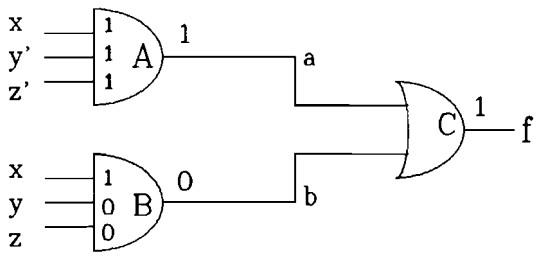

Figure B.1. Implementation of the function f with an input vector.

First the gates have to be marked. In this example this is

gate A = ALMOST ACTIVE
gate B = NOT ACTIVE
gate C = ACTIVE.

The set of faults belonging to this input vector is formed now. First the opposite fault for the output has to be added. This means that signal line f with the stuck-at faults sa0, written as f/0, is added. Because gate C is ACTIVE the only way through which faults can propagate is via signal line a. This means that gate B does not need to be calculated any further. For the signal line a the fault a/0 has to be added. Now gate A has to be considered. This gate is marked ALMOST ACTIVE, so all the inputs can propagate all the fault which have propagated so far. This means that in a larger circuit all the logic in front of gate A has to be processed. In this example, only the stuck-at faults sa0 for every input of gate A have to be added. The inputs are reached and the resulting set of faults contains five stuck-at faults.