Debugging integrated circuits: A.I. can help!*

M. MARZOUKI B. COURTOIS

Computer Architecture Group, IMAG/TIM3 Lab.
46 Avenue Félix VIALLET
38031 GRENOBLE Cedex - FRANCE

ABSTRACT

PESTICIDE, "a Prolog-written Expert System as a Tool for Integrated Circuits Debugging", relies on the knowledge of structural and functional properties of both combinational and sequential ICs. Its external environment is constituted by a scanning electron microscope used in voltage contrast mode and linked to CAD tools.

KEYWORDS
Debug, diagnosis, expert system, test, e-beam testing, PROLOG.

1. INTRODUCTION

The problem of integrated circuits testability is set at many levels, so that different tools and methods have to be used to solve it, according to the availability of information about the device under test.

This paper is concerned with prototype validation of VLSI's, so it is assumed that all the necessary information are available. In order to achieve this prototype validation process, and considering the high degree of integration of VLSI's, new test tools have been used, such as scanning electron microscope, which allows both good observability and controllability in the same tool [11, 12, 13, 14, 15]. But now, the problem is set at another level: since design or test engineers are not supposed to be well experienced in manipulating such tools, appropriate test methods must be defined for interpreting the results of the device observation, in order to have the defect identification and localization diagnosis completely automated.

For that purpose, the choice of an expert system has been made [10].

This choice is justified in one hand by the inherent qualities of expert systems (maintainability, user-friendly interfaces, adaptability, versatility), and in the other hand by the suitability of the topic (different fault hypotheses can be modeled, possibility to take into account different hierarchical specification levels, suitability to easily handle a temporal model of sequential circuits, possibility to handle a large mass of data, possibility to work with incomplete data). Moreover, the choice of PROLOG, a logic programming language, for building expert systems results from a deep study of the problem, which has demonstrated that PROLOG is suitable for implementing tasks such as modeling, design ([11], [12]), simulation [13], test pattern generation [14] and fault diagnosis of complex VLSI circuits [15, 16, 17, 18, 19].

After a brief description of the expert system environment (section 2), section 3 will detail the modeling of the device under test and how this modeling is used to feed on the databases, section 4 describes the strategies adopted for fault detection and localization within the circuit under test, while section 5 provides some simple execution examples of PESTICIDE.

2. THE EXPERT SYSTEM ENVIRONMENT

The expert system is composed by [20]:
- the PROLOG interpreter,
- a "static" database, containing all the necessary information about the device under test,
- a "dynamic" database, containing information about the test currently carried out,
- a control program, made of PROLOG rules, which is the expression of the expertise about the domain.

It is the core of a Data Processing System (D.P.S.), which assumes the following functions:
- Partitioning the circuit structure by means of C.A.D. tools
- Determining the observed nodes status, by comparing observed and simulated logical values
- Establishing the diagnosis using PESTICIDE.

The main inputs of the D.P.S. are some hypotheses about the type of test (formulated by an operator), the circuit structure description, circuit functioning simulation results, both coming from external C.A.D. tools, and a S.E.M. image of the device under test, coming from the Data Acquisition Tool (D.A.T.). This D.A.T. is functionally divided into three components which are [20]:
- an exerciser, allowing to control the debugged circuit by simulating the environment of the device under test,
- the Scanning Electron Microscope, used in voltage contrast mode, and controlled via its control sub-system,
- an image processor which process unit is made up of 8 microprocessors working concurrently on 8 sub-images.

The integration of the D.P.S. and the D.A.T. will form a completely automated integrated debugging system based on

* The work reported hereafter has been partly supported by an European collaboration among BTRL (U.K.), CNET (France), CSELT (Italy), IMAG (France) and Trinity College Dublin (Ireland) aimed at fully automating the electron beam testing procedure for design validation of VLSI circuits.

CH2696-3/89/0000/0184$01.00 © 1989 IEEE
3. MODELING OF THE CIRCUIT- DATABASES

In this modeling, the device is hierarchically decomposed into interconnected blocks. Three basic components describe the circuit: the block, the wire (connection) and the block interface.

Hypothesis:
A sequential circuit is considered as a sequential one if, at a given step of the test process, its partitioning provides blocks interconnected in such a way that loops can appear. Thus, this partitioning represents a sequential system of blocks.

Given this hypothesis, the proposed model makes it possible to represent sequential circuits as easily as combinational ones, by means of associating to each component the relevant parameters.

3.1. Modeling of the circuit

The block parameters are its name, and a list of block interfaces, made up of input, output or bidirectional block interfaces connecting the block to its external environment.

The block interface allows to connect the blocks between themselves and to the pads of the device. The major reason of introducing this notion of block interface is to allow fault detection on wires in one hand, and in the other hand, to allow fault detection in sequential circuits, thanks to time parameters. Parameters associated to a block interface are its name, its state (erroneous or not, relatively to the current test session), its logical value, the measuring instant of this value, its type (input or output of a block, or bidirectional), a delay, measured in clock cycles, representing the time after which a block interface value can be taken into account. This delay is evaluated as the functioning time of the block plus the stabilization time of the block interface value (if type = output) or the propagation time of the wire (if type = input). The last parameter is the hold time of the block interface.

The wire has also a name and an associated list of block interfaces, connected either upstream or downstream to the wire.

These three components form the basis of the databases.

3.2. The "static" database

The "static" database contains all the available information about the device under test. These data consist of:

* The definition of the blocks, with their two parameters
* The definition of the block interfaces, restricted to the name, the type, the delay and the holding time, which constitute the "static" parameters of the block interface
* The definition of the wires, with their two parameters The "static" database contains also two additional information concerning the block interfaces.

* A condition, stated for each bidirectional block interface, to know the way (input or output) to switch this block interface to, according to a list of conditions, which are particular values of some other given block interfaces
* A coverage cone, stated for each block interface, which expresses the functional dependency [22] between an output-type block interface and the input-type block interfaces that belong to the list of members of the coverage cone. Although not necessary, this notion of coverage cone allows to improve the functioning of the expert system (fig. 2):

Suppose that both an error on output O2 and inputs I3 and I5 are observed. If all the inputs of the block are undistinguishable, then the expert system can activate propagation rules on both the paths "O2 - I3" and "O2 - I5", while the input I5 is not functionally related to output O2. Such a wrong activation of propagation rule can easily be avoided when the expert system knows what is the coverage cone of each block output, making it possible to save time by activating only the right propagation rules, and avoiding then a lot of useless datapath examinations.

3.3. The "dynamic" database

The "dynamic" database contains data about the current test, and consists of the following information:

* Instances of block interfaces qualified by the name of the concerned block interface, and its "dynamic" attributes, which are its state (erroneous or not), its logical value and its way (input or output), at a given measuring instant.
fault, fault model, for the same type of circuits), test, that states the fault model assumed for this test session.

combinational circuits), *combinational" fault

constraints are due to the application domain of the expert results.

measuring instant, to have the expert system give correct examination of either the observed logical value or the state of system, and are verified by this system.

4. FAULT DETECTION AND LOCALIZATION

Localizing faults either on wires or within blocks implies the examination of either the observed logical value or the state of block interfaces. These two parameters are depending on their measuring instant from the beginning of the test process, therefore it is compulsory to examine them at the "good" measuring instant, to have the expert system give correct results.

4.1. Evaluation of a measuring instant validity strategy

An evaluation strategy for the validity of a measuring instant has been defined, which is based on the following parameters of a block interface or an instance of it: the delay, the hold time, and the measuring instant itself.

i) Examination of a block:
The inputs and outputs of a block must satisfy the following relations:

R1) output block_int. instant ≥ input block_int. instant + output block_int. delay
R2) output block_int. instant ≤ input block_int. instant + output block_int. delay + input block_int. hold time

This is sufficient when there is one and only one member of the coverage cone of each output. If there is more than one member, the variant is:

R1') output block_int. instant ≥ "MAX(input block_int. instant)" + output block_int. delay
R2') output block_int. instant ≤ "MAX(output block_int. instant)" + output block_int. delay + "MIN(input block_int. hold time)"

ii) Examination of a wire:
Since the "delay" parameter of a block interface has different meanings depending on the type of this block interface, the relations R1 and R2 can be easily adapted to the case of a wire. The relations R1' and R2' have no significance in this case, because only one upstream block interface can be connected to a given wire (otherwise, a short is detected).

The relations R1 and R2 then become:

R1") downstream block_int. instant ≥ upstream block_int. instant + downstream block_int. delay
R2") downstream block_int. instant ≤ upstream block_int. instant + downstream block_int. delay + upstream block_int. hold time

4.2. Fault detection on a wire

This strategy is used to detect other defects than shorts, which are detected at the verification of data consistency step. These other defects can be cut of a wire or occurrences of signals not sufficiently amplified.

The strategy used is the following one:

If all the block interfaces connected to the examined wire have not the same logical value, then this wire is faulty.

Note that the logical value is supposed to be observed at the relevant instant, then it is necessary to apply the strategy used to evaluate the validity of a measuring instant. The following example (fig. 3) can illustrate the fault detection process on a wire:

![Fig. 3. Fault detection on a wire](image)

In this example, the block interface IB₁ is connected upstream to the wire, and IB₂ to IB₅ block interfaces are connected downstream. Assuming that the expression:

```
VAL(IB / TIME)
```

represents the logical value of the block interface IB observed at the instant TIME, the wire of fig. 3 is correct if and only if the following relations are true:

```
VAL(IB₁ / t) = VAL(IB₂ / t + delay(IB2))
... = VAL(IB₅ / t + delay(IB₅))
```

where delay(IB) is the delay parameter associated to the block interface IB.

4.3. Fault localization within blocks

The strategy for fault localization within blocks is adopted according to the fault hypothesis made by the user for a given test session. Before presenting the four strategies, some important remarks must be made:

Remark 1:
Fault localization within blocks necessitates to have the circuit under test functioning, even if it is an off-line test, so bidirectional-type block interfaces will not be mentioned any more in this section. Indeed, all the block interfaces values will be instantiated during the functioning of the circuit, particularly those that are switching conditions for bidirectional-type block interfaces, consequently these conditions will be applied to decide the way (input or output) to switch each block interface type to.
Remark 2:
Sequential circuits are considered, but special feedback loops elimination or transformation procedures are not used. This fact can be easily explained: considering different instances of each block interface makes it possible to have the device debugged in a step-by-step functioning mode. Therefore, it is possible to have at any given moment an instantaneous image of the circuit under test, and consequently, a differentiation between external inputs and internal states of a sequential system of blocks is totally useless.

Remark 3:
When purely combinational devices are considered, the following parameters are ignored:
* delay and hold time of a block interface,
* measuring instant of a block interface instance.

This is possible since we can consider, in this case, that the delay is null, that the hold time is infinite, and then that the measuring instant does not matter.

The different strategies can now be described: they are the expression of classical rules for propagation/not propagation of errors within a given device.

S1) "Single combinational" fault localization strategy:
When this hypothesis is formulated, the expert system searches first all the block interfaces which state parameter is declared as erroneous, and then, for each of them, apply the following strategy:
* examine the block which has this block interface as an output,
* if none of the members of the coverage cone (inputs) of this block interface is erroneous, then this block is faulty, and manifests a DIRECT-type error on its primary output,
* else, if one of the members of the coverage cone is erroneous, then go upstream and examine other blocks, until reaching the faulty block that manifests a PROPAGATED-type error on the current block interface.

At the end of the process, the system provides as many results as the number of erroneous block interfaces.

S2) "Multiple combinational" fault localization strategy:
When this hypothesis is formulated, the process searches first the faulty blocks which manifest a DIRECT-type error, in exactly the same way as for the single fault case. After that, the process searches the blocks that are assumed to be faulty, by applying a specific strategy to each of the faulty blocks. Some blocks are called "assumed to be faulty" because it can happen that the knowledge the system has about them is not sufficient to decide whether they are really faulty or not. Therefore, it will be necessary to apply afterwards to them specific treatments. These blocks are such that they have an erroneous input. The strategy is the following one:
* for each faulty block, search its erroneous outputs,
* for each of these block interfaces, examine the block(s) which has (have) this block interface as an input,
* if this block is not already declared as faulty, assume it to be faulty
* apply this strategy to all downstream blocks of this one.

S3) "Single Sequential" fault localization strategy:

S4) "Multiple Sequential" fault localization strategy:

They will not be detailed here because they are exactly the same as, respectively, the S1 and S2 strategies, except that a referenced value or status is assumed to be a "good" one, that is, a value or status captured at the relevant instant. This is done by using, at each step of the reasoning process, the strategy for evaluating the validity of a measuring instant [10].

Note that the fault localization process is interactive: the user (a design or test engineer) asks questions to PESTICIDE. If the expert system has a sufficient knowledge about the circuit and the current test, it can then directly provide the answer, otherwise, PESTICIDE asks the user for some additional information, like the state of a not yet observed block interface. Further developments of PESTICIDE will allow it to also ask for applying some given input pattern sequence.

4.4. Taxonomy and internal examination of blocks

After having been achieved, the fault localization within blocks leads to a block classification. There are three categories of blocks:

1) Faulty blocks
2) Blocks assumed to be faulty
3) Blocks assumed to be correct

The blocks that are neither faulty, nor assumed to be faulty are just assumed to be correct, because this classification has been obtained according to a particular test sequence.

For each category of blocks, some specific treatments must be applied. These treatments are not implemented yet, but can consist of:

1) For a faulty block: partitioning it on its turn, and applying to it the same expertise rules as for the whole circuit, in order to obtain a finer fault localization.

2) For a block assumed to be faulty (in the multiple fault case): simulating it (at the functional level) with the same erroneous inputs as those observed during the test sequence. If the simulation process gives the same outputs as the observed ones, then this block will be classified in the third category, else it can be deduced that the block is really faulty, and belongs in fact to the first category.

3) For a block assumed to be correct: making on-line test pattern generation, for an off-line type test. This idea of carrying out a real-time off-line test pattern generation seems to be quite reasonable, since this test will be applied to a particular block with a few number of inputs, and since the test process we are concerned with is a debug process, that is, the question of time response is not so crucial in this case.

It can be also possible, for the same reasons, to apply an exhaustive test, since it will be restricted to a particular block and since the coverage cone notion will be helpful to limit the length of the test sequence. This test will then be a pseudo-exhaustive test [22].

5. PESTICIDE EXECUTION EXAMPLES AND EVALUATION

In this section, some PESTICIDE execution sessions will be presented and detailed. For a better understanding of these
results, the example is restricted to a four-bits adder, represented as four interconnected blocks, each block being a single-bit adder (fig. 4). This choice is not so restrictive, since, up to now, PESTICIDE only uses topological propagation rules, then we are not really interested in what a block is, rather are significant the number of the blocks, their interrelations and the number of the observed erroneous block interfaces. The example is such that there is no faulty wire in the circuit, since this type of defect and its localization is easy to understand.

Fig. 4. Representation of the dynamic database for a four-bits adder.

5.1. Results obtained with the single fault hypothesis

The databases correspond to fig. 4, where some erroneous block interfaces are depicted, when the single fault model is assumed. The bold lines represent the user inputs, while the other lines are the PROLOG interpreter outputs. Line 02 is a request to the interpreter to load the PROLOG database with the files "util", "diagsesq", "statadd4" (static database), and "dynadd4f" (dynamic database). "Util" and "diagsesq" contains the PROLOG program, that is, the inference rules.

01 C-Prolog version 1.5
02 I- [util,diagsesq,statadd4,dynadd4f].
03 util consulted 3876 bytes 1.24 sec.
04 diagsesq consulted 6520 bytes 2.8 sec.
05 statadd4 consulted 4112 bytes 1.9 sec.
06 dynadd4f consulted 1664 bytes .820003 sec.
07 yes
08 I- preliminary.
09 yes
10 I- singlecomb(-blocklist,-hint,-type).
11  _blocklist = [add0,add1]
12  _hint = co0
13  _type = propag ;
14  _blocklist = [add0,add1,add2]
15  _hint = s2
16  _type = propag ;
17  no
18  I- halt.

"Preliminary" is a PROLOG predicate which role is to establish the reverse relations between the blocks, the block interfaces and the wires in order to know, for each block interface, to which block and to which wire it is related. This operation of establishing the reverse relations is done as follows:

i) relations between blocks and block interfaces:
From each block declared in the static database, the list of corresponding block interfaces is extracted, and the reverse relations are appended to the database. For example, look at the definition of block ADD0:

block(add0,[ci0,a0,b0,s0,co0]).

The extracted list of block interfaces is [ci0,a0,b0,s0,co0], and the reverse relations (named "primaryblock(block_interface, block)") are:

primaryblock(ci0,add0),
primaryblock(a0,add0),
primaryblock(co0,add0).

ii) relations between wires and block interfaces:
The same process is handled from each wire in the static database. For example, look at the definition of wire W5:

wire (w5,[co0,ci1]).

The extracted list of block interfaces is [co0,ci1], and the reverse relations (named "primarywire(block_interface,wire)") are:

primarywire(co0,w5),
primarywire(ci1,w5).

The advantage of carrying out these two processes once and for all by executing the "preliminary" predicate at the beginning of a PESTICIDE work session is a considerable gain in CPU time.

"Singlecomb" is a PROLOG predicate which role is to effectively localize the faulty block. The first argument, "_blocklist", is the name of the faulty block, if it manifests a direct type error on the concerned block interface ("_hint"), or the list of blocks which constitute the propagation chain if the manifested error is a propagated one.

Then, the following PROLOG question:

singlecomb(_blocklist,_hint,_type).

can be interpreted in plain language by the following question:
what is the block (if any) that manifests a direct-type error on the block interface "_bint", and what is the list of blocks (if any) that constitutes the propagation chain from the faulty block to the block interface "_bint" where a propagated-type error is manifested?

In this case, there are three answers, since three output-type block interfaces are declared as erroneous (fig. 4):

i) first answer, lines 11 to 13:

The concerned block interface is co0. Since the "preliminary" predicate execution has appended the relation "primaryblock(co0,add0)" to the static database, and since this block has no erroneous input, then the faulty block is add0, and the error manifested is a direct-type one (co0 is a primary output of add0).

ii) second answer, lines 14 to 16:

The concerned block interface is s2, and is a primary output of block add2 (deduced from the relation: primaryblock(s2,add2)).

Add2 has an erroneous primary input, named ci2, which belongs to the coverage cone of s2, then a propagation rule must be activated, to detect the faulty block which is actually responsible of the error manifested on s2.

This propagation rule first detects that block interface ci2 is linked to block interface co0, via wire w9, thanks to the relations: primarywire(ci2,w9) and wire(w9,[co0,ci2]).

After that, co0 is identified as a primary output of block add0 (from the relation: primaryblock(co0,add0)).

Since add0 has no erroneous input, then this block is faulty and the backtrace process can be stopped.

The propagation chain is then: [add0,add2].

iii) third answer, lines 17 to 19:

The concerned block interface is s2, and is a primary output of block add2 (deduced from the relation: primaryblock(s2,add2)).

Add2 has an erroneous primary input, named ci2, which belongs to the coverage cone of s2, then a propagation rule must be activated, to detect the faulty block which is actually responsible of the error manifested on s2.

This propagation rule first detects that block interface ci2 is linked to block interface co0, via wire w9 (fig. 4), thanks to the relations: primarywire(ci2,w9) and wire(w9,[co0,ci2]).

After that, co0 is identified as an erroneous primary output of block add1 (from the relation: primaryblock(co0,add1)).

Add1 has an erroneous primary input, named ci1, which belongs to the coverage cone of co0, then a propagation rule must be activated twice, to detect the faulty block which is actually responsible of the error manifested on s2.

This second activation of the propagation rule detects that block interface ci1 is linked to block interface co0, via wire w5, thanks to the relations: primarywire(ci1,w5) and wire(w5,[co0,ci1]).

After that, co0 is identified as a primary output of block add0, from the relation: primaryblock(co0,add0).

Since add0 has no erroneous input, then this block is faulty and the backtrace process can be stopped.

The propagation chain is then: [add0,add1,add2].

Now, since there is no other answer (line 20), the work session is halted by the user (line 21).

5.2 Results obtained with the multiple fault hypothesis

01 C-Prolog version 1.5
02 ?- [util,diagseq,statadd4,dynmult4].
03 util consulted 3876 bytes 1.24 sec.
04 diagseq consulted 6520 bytes 2.72 sec.
05 statadd4 consulted 4112 bytes 1.92 sec.
06 dynmult4 consulted 1664 bytes .8 sec.
07 yes
08 1?- preliminary.
09 yes
10 1?- multiplecomb.
11 *** add0 faulty
12 * add1 assumed to be faulty
13 * add2 assumed to be faulty
14 yes
15 1?- halt.
16 [ Prolog execution halted ]

"Multiplecomb" is a PROLOG predicate which role is to effectively localize the faulty blocks, and the blocks assumed to be faulty, relatively to each faulty block found.

Then, the following PROLOG question (line 10):

multiplecomb.

can be interpreted in plain language by the following question:

what are the faulty blocks, and, for each of them, what blocks can be assumed to be faulty, that is, what are the blocks that necessitate a deeper examination?

In this case, there are three answers: one faulty block, and two assumed to be faulty.

The first answer (line 11) is deduced from the fact that block add0 has an erroneous output (co0) and no erroneous input, in the same way as for the single fault case.

Since the faulty block(s) has (have) been identified, the forward propagation process can be started.

The execution of the "preliminary" predicate (line 08) has appended the static database with some primaryblock (block_interface,block) and primarywire (block_interface,wire) predicates, then it can be detected that block interface co0 is linked to block interface ci1 via wire w5 (relations used: primarywire(co0,w5) and wire(w5,[co0,ci1])).

After that, ci1 is identified as an input of block add1, thanks to the relation primaryblock(ci1,add1), and ci1 is an erroneous block interface, then it is deduced that block add1 has an erroneous input: this is sufficient to assume add1 to be faulty (line 12), according to the definition of such type of block.

The forward propagation process continues, detecting that block interface ci2 is linked to block interface co0 (primarywire(co0,ci2) and wire(w9,[co0,ci2])), and that ci2 is an erroneous primary input of block add2 (primaryblock(ci2,add2)). It can then be deduced that block add2 is also assumed to be faulty (line 13).

5.3. CPU time evaluation

The table of fig. 5 gives the cpu time measured for a two-bits adder, a four-bits adder, an eight-bits adder and a sixteen-bits adder, either in the single fault case or the multiple fault case, with a varying number of erroneous block interfaces.

These results have been obtained using the C-PROLOG version 1.5 interpreter, on a SUN 3/160 computer under the version 4.2 of the UNIX operating system.
Table 1: CPU times (seconds) taken by the localization of faulty blocks

<table>
<thead>
<tr>
<th>Adder Size</th>
<th>Single Fault</th>
<th>Multiple Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bits</td>
<td>4 errors</td>
<td>4 errors</td>
</tr>
<tr>
<td>4 bits</td>
<td>6 errors</td>
<td>6 errors</td>
</tr>
<tr>
<td>8 bits</td>
<td>10 errors</td>
<td>10 errors</td>
</tr>
<tr>
<td>16 bits</td>
<td>12 errors</td>
<td>12 errors</td>
</tr>
</tbody>
</table>

Fig. 5. CPU time (seconds) taken by the localization of faulty blocks

6. CONCLUSION

PESTICIDE is already implemented, but not completely evaluated yet. Nevertheless, some remarks can be formulated about this system:

First of all, it is important to note that this system can not only take into account combinational circuits, but also sequential ones, by means of an adequate data structure.

Secondly, the idea of considering a sequential circuit at a given fixed instant as a combinational one seems to be interesting, since it allows to avoid the use of specific procedures for each type of circuit.

Thirdly, it can be noted that this expert system relies rather on what is called a "deep knowledge" of electronic devices structure and behavior, than on empirical (sometimes also called "shallow") knowledge, namely, established relations between a collection of causes and a collection of symptoms/effects. Consequently, the diagnosis methodology used in PESTICIDE can be classified into the "diagnosis from first principles" methods category.

Fourthly, the fact that no classical, well-known, fault model is used must be pointed out. Indeed, the very nature of PESTICIDE implies that the only starting point of the fault localization process is that an error has occurred, more precisely, that a discrepancy between a reference ("golden") value and a measured one has been observed.

Beyond the fact that, with this extremely general fault concept, the expert system can be used to localize within a digital circuit faults that can hardly be identified using classical C.A.D. tools such as simulators, the fault localization methodology presented in this paper leads us, for the same reason, to optimistically consider its extension to the debug of electronic systems, and/or to the debug of boards, provided that sufficient observability conditions are guaranteed.

REFERENCES

[1]: FAZEKAS P., FEUERBAUM H. P., WOLFGANG E. "Scanning electron beam probes VLSI chips". Electronics, 14 JULY 1981
[5]: TAMAMA T., KUJI N. "Fully automated electron beam tester combined with CAD database". International Congress on Electronic Microscopy Kyoto, JAPAN. 1986
[6]: CONCINA S. E., LIU G. S. "Integrating design information for IC diagnosis". 24th Design Automation Conference, Florida, USA. JUNE 1987
[8]: GORLICH S., HARBECK H., KEBLER P., WOLFGANG E., ZIBERT K. "Integration of CAD, CAT and electron-beam testing for IC-internal logic verification". International Test Conference, Washington, USA. SEPTEMBER 1987
[9]: RICHARDSON N. "E-beam probing for VLSI circuit debug". VLSI systems design. AUGUST 1987
[13]: GULLICHSEN E. "Heuristic circuit simulation using PROLOG”, INTEGRATION, the VLSI journal North-Holland Publishers 1985

[16]: ESHGHI K. "Application of metalevel programming to fault finding in logic circuits". 1st international logic programming conference Marseille, FRANCE. SEPTEMBER 1982

[17]: DUDA R. O. "Applications of expert systems". Automatic test program generation workshop California, USA. MARCH 1983


[22]: Mc CLUSKEY E. J. "Verification testing - a pseudo-exhaustive test technique". IEEE transactions on computers Vol c-33, No 6. JUNE 1984

[23]: DAVIS R., SHROBE H. "representing structure and behavior of digital hardware". IEEE computer journal, pp. 75-82. OCTOBER 1983