A New Approach to Build a Low-Level Malicious Fault List Starting from High-Level Description and Alternative Graphs

A. Benso, P. Prinetto, M. Rebaudengo, M. Sonza Reorda
Politecnico di Torino, Torino, Italy
R. Ubar
Tallinn Technical University, Tallinn, Estonia

Abstract

In this paper a new approach is presented to build a list of faults to be used by the fault injection environment; the list is built starting from a high-level description of the system. The approach especially aims at identifying malicious faults i.e., faults having a critical impact on the system reliability. To overcome the complexity problem inherent in low-level descriptions, high-level ones are exploited, and alternative graphs are applied to carry out the cause-effect analysis, to build up a fault tree and to carry out fault collapsing. The reduced high-level malicious fault list is converted so that it can be used together with the low level description for the final fault injection.

1. Introduction

Computers are becoming more frequently used in time or safety critical applications such as avionics, public transportation systems, process industry, telecommunications, etc., where a computer failure could be very costly. As a consequence, there is an increasing demand for systems with high dependability. In the last decade, fault injection has emerged as a useful mean to support the dependability validation of fault-tolerant systems [6].

The objective of fault injection is to mimic the effects of fault originating inside a chip as well as those faults affecting external buses.

In the majority of the published works, the fault location, the fault type and the fault insertion time, are randomly selected [1,4]. For ultra-reliable life-critical systems the random selection of faults is not an adequate method for determining the fault coverage [8].

Several works [1,3] demonstrated that with randomly selected fault lists the percentage of faults which do not produce errors ranges from 10 to 66%, depending from the system. The goal of the fault injection experiment is to exercise the system's fault processing capabilities. Faults which fail a system in the absence of system fault detection capabilities are defined to be *malicious* [8]; a malicious fault, if undetected in presence of fault processing mechanisms, will fail the system under test. Using malicious faults to estimate fault coverage eliminates the possibility of fault injection experiments to produce no errors.

In [8] a method is presented to generate a malicious fault list. The system under test is described by a data flow graph, the fault tree is constructed by applying the Instruction Set Architecture fault model to the data flow description with a reverse implication technique, the fault injection is performed, and fault collapsing on the fault tree is employed. The method proposed is very costly in terms of CPU time and it seems not applicable to more complex systems.

In this paper a new approach to generate a malicious fault list is presented, based on two different levels: at a high-level, alternative graphs (AGs) [9,11] are applied to build the fault tree; a fault collapsing is then made on that tree, and the reduced fault list is converted so that it can be used together with the low level description.

The approach developed in this paper helps to overcome the complexity problem inherent in low-level descriptions, and allows to carry out a comprehensive fault analysis in order to sort out the *no response* faults, i.e. faults which give no information while evaluating the system fault coverage by fault injection. AGs give a comprehensive formalism that can be adopted at different levels of abstraction. Moreover, they guarantee the generality of the fault model and the possibility of using uniform techniques at different design representation levels, independently of the adopted fault model and for both fault trees construction and fault analysis.

In comparison with [8], in the present work fault trees (FTs) are created directly from AGs without expanding the full data flow, whose size can sometimes explode. Differently from [8] control faults can be explicitly handled as they are directly inherent to the AG-model.

Thanks to the generality of AGs, the present approach is independent from the fault model. This independence allows the easy introduction of hierarchy and multilevel representation into fault handling. This can be exploited, for example, to improve the results of the high-level fault analysis through collapsing the high-level faults by low-level ones in the later design phase, when the information about the implementation details are available.

The paper is organized as follows: Section 2 describes the Environment adopted; in Section 3 we briefly summarize the Alternative Graphs; Section 4 presents the meth-

ED&TC '97 on CD-ROM

ods used to build AGs from a High-level description; the generation of the Fault List is presented in Section 5; Section 6 presents the High-level Malicious Fault List generation and the High-level Fault List collapsing using AG simulation; Section 7 draws some conclusions.

2. The environment

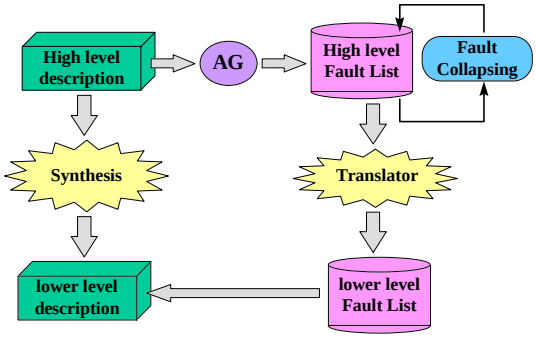

Fig. 1: The environment.

The methodology proposed in this paper to create a low-level malicious fault list starting from a high-level description of the system is organized as shown in Fig. 1. The main steps are the following:

- build Alternative Graphs from the High-level description of the system;
- create a High-level Fault-List;
- generate a High-level Malicious Fault List and collapse the High-level Fault List using AG simulation;
- translate the obtained High-level Fault List into a Low-level Fault List.

3. Alternative graphs and digital systems

Alternative Graphs [9,11] can be used for representing digital functions $y = F(Z)$ of components or subcircuits in digital systems. Here, y is an output variable and Z is a vector of input variables of a component.

Alternative Graph is a directed acyclic graph $G_y = (M, \Gamma, Z)$ with a set of nodes M, with a single root node $m_0 \in M$, and relation Γ in M where $\Gamma(m) \subset M$ denotes the set of successor nodes of m. Nonterminal nodes m for $\Gamma(m) \neq \varnothing$, have variables $z_i \in Z$ as labels. Terminal nodes m for $\Gamma(m) = \varnothing$ have variables $z_i \in Z$, functional sub-expressions of $F(Z)$, or constants as labels. Let us denote by $z(m)$ the label of a node m. In the graph G_y, for all non-terminal nodes m, $\Gamma(m) \neq \varnothing$, a one-to-one correspondence exists between the values of label variable $z(m)$ and the successors, $m_k \in \Gamma(m)$ of m.

The methods for generating AGs for different representation forms of digital functions are described in [10,11].

Alternative graphs were introduced for representing the functions of a system to reveal explicitly the functional signals cause-effect relationships. By tracing an AG-model of a system for the given time (i.e., for the given status and input pattern), we determine a subset of nodes and, correspondingly, a subset of variables, which are responsible for the system's behavior at the considered time. If the behavior of the system has a failure, then only the mentioned subset of variables can be suspected for causing the failure. Hence, AGs are a way to represent concisely the diagnostic information of a system, and provide a powerful means for a fault analysis in general, and for selecting malicious faults in fault-tolerant systems validation in particular. In this section, we describe some issues concerning how to represent high-level descriptions through AGs.

Consider a digital system $S = (Z, F)$, where Z is a set of digital variables z (the numbers $V(z)$ of possible values for $z_k \in Z$ are finite), and F is a set of digital functions on Z. The system S can be represented by an AG-model such that for each function $z_k = f_k(Z_k)$, $z_k \in Z$, $f_k \in F$, $Z_k \subseteq Z$, there exists an AG, G_k.

To generate a high-level AG-model for a digital system, a specification of the system as a connection of high-level blocks each described as a high-level function (or a set of controlled functions) is needed.

As an example, a simple VHDL function as a description of a digital system that computes the sum of the elements of an array is reported in Fig. 2. This example will be used throughout the paper to illustrate the steps composing the method we propose.

```
function sum_array
(input: in my_array) return integer is
variable temp:integer:=0;
variable j:integer:=0;
begin
       for j in 0 to N
              temp := temp + in(j);
       next j;
  return temp;
end function;
```

Fig. 2: A VHDL example.

The AG-model of the system is represented in Fig. 3. In the model the VHDL cycle `for j in 0 to N` is substituted by the next-state function $j=F_1(j',N)$ for the control part of the system. The data part accomplishes the function $temp=F_2(j',temp',in(j'))$ which is controlled by the value j'. The time of the model is represented by states of the finite state machine (FSM), i.e., by values of the state variable j. By apostrophe we denote that the value of the variable belongs to the previous time (i.e., to the previous state).

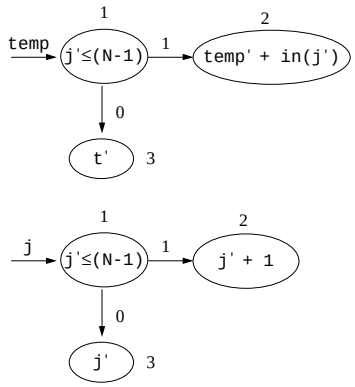

Fig. 3. An AG-model for the VHDL example.

4. Building the Fault List

Once the AG has been built from the high-level description, the analysis described in this section should be performed which leads to the generation of a high-level Fault List.

4.1. Fault tree generation by tracing AGs

Each path in an AG G_z describes the behavior of the system variable *z* in a corresponding mode of operation. The faults having effect on the behavior at a given test pattern can be related to nodes along the path activated by the pattern only, not to the remaining nodes of the graph.

If an erroneous signal is detected at *z*, then the set of candidate faults which can explain the misbehavior of *z*, can be created by fault site backtracking procedure. This set will be represented in the form of a Fault Tree, whose nodes are labeled by a pair: the variable representing the failed observation point, and the expected value of the variable for the error free case. With each node of the tree a fault list will be associated.

When creating the FT for an observation point *z* first, the path activated by a test pattern will be determined in the graph G_z. All the label variables (or expressions) of nodes on this path together with the expected values of variables for the error free case are put into the FT as successors of the root node. For nodes labeled by expressions, for each variable in the expression, a successor node is created, labeled by the variable and its value. For each leaf node in the tree labeled with a variable z^* which corresponds to an internal signal of the system (i.e., it is not an input signal), and which is represented by a corresponding graph G_{z^*}, again, an activated path in G_{z^*} is determined whose nodes form the set of successors of z^* in the FT. This procedure is recursively repeated until all the leaves in the FT are labeled by input variables.

The procedure begins at the last pattern of the given test sequence. Traversing a graph for the pattern at a given time moment *t*, then for all variables *z'* found in AGs with apostrophe, a test pattern for the previous time moment *t-1* should be taken for traversing the graph G_z.

In Table 1, a test experiment is given for executing the VHDL program in Fig. 2. Let have N = 2.

time	temp	j	in(j)
0	0	0	0
1	10	1	10
2	20	2	10

Tab. 1: An application example.

By tracing the AGs in Fig. 3, starting with the last test pattern given for t=2 (the time is determined by the value of the variable `j`), we create a fault tree consisting of the nodes 1-6 in Fig. 4. For example, in the graph G_{temp} the nodes 1 and 2 are traced, which give the predecessors nodes 2 and 3 for the root node 1 in FT. Further on, as the node 2 has the variable `j'` in his label expression, we create a new node 4 as the successor for node 2, labeled by the variable `j'` and its corresponding value taken from the test pattern. As the value of `j'` was created at the previous time moment, the graph G_j for finding successors of the node 4 is traversed for the pattern given at t=1.

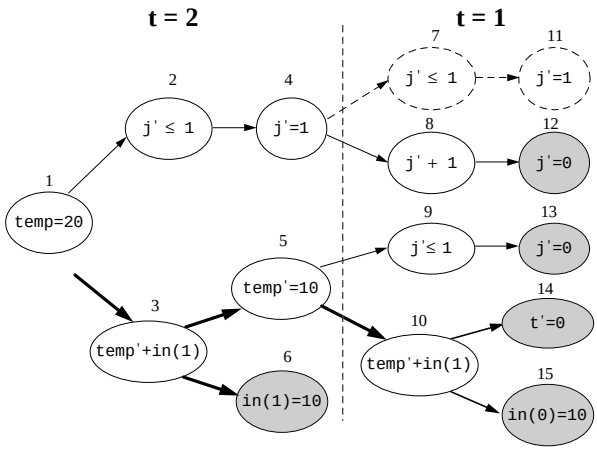

Fig. 4: Fault tree for the AG-model in Fig. 3.

The number of nodes in a FT is generally less than the number of nodes contained in the traceback cone [10] (or transitive fan-in [7]) in the corresponding digital network. Additional fault analysis helps to minimize the size of a fault tree. If the whole fault set associated with a node in FT is collapsed, then no successors for that node are to be created. For example, in Fig. 4, the construction of the tree stops at node 7 due to the collapsing of all faults at this site: adding node 11 to the tree becomes useless.

At this point, the benefits of the present approach in comparison with [8] manifest themself clearly: instead of constructing the full data flow diagram (as in [8]), combining the fault tree construction from AGs with fault collapsing allows us to represent only a part of the data flow in the final fault tree.

4.2. Injection space and Injection time

After creating the FT for an AG-model, the following steps provide a fault list where, for each fault, the following information are provided:

- variable to be injected
- faulty value to be injected in the variable
- injection time.

From a high-level description it is possible to define two sets of variables: *control variables*, that control the execution of the system, and *data variables*, that store the data needed during the execution. The concept of *time* is strictly related to control variables; we can assume that each time slot is uniquely defined by the values of the control variables and, when necessary by some additional *virtual variable*, which are not related neither to any actual data in the program nor to any space in memory.

In the example reported in Fig.2 `j` is a *control*, whereas `input` and `temp` are *data variables*; the number of times (number of test vectors) that the function is called by the main process is a *virtual variable*.

The *injection space* (i.e., the set of variables in which it is possible to inject a fault) corresponds to all the *control* and *data variables* (Fig. 5).

Therefore, in the example (Fig.2) a fault could be defined as follows*: change the value of `j` to 3 when j=2 and it is the fourth time the function is called*; *j* is the injection target, 3 is the faulty value to inject and "*when j=2 and it is the fourth time the function is called*" defines the injection time.

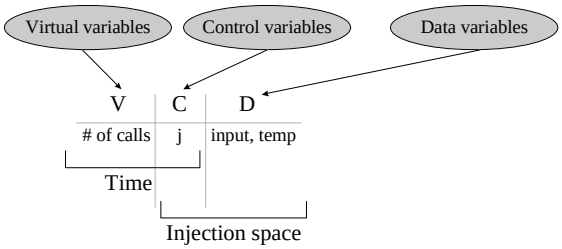

Fig. 5: Variable groups.

5. Malicious fault list generation

The Malicious Fault List (*MFL*) generation for the given FT is carried out iteratively for all nodes $m \in FT$, supposed that the Fault Lists $F(m)$ for these nodes are given. The goal of the procedure is to iteratively calculate the corresponding Malicious Fault Lists $MF(m)$ for nodes $m \in FT$. For the root node m_0 we assume that all faults in $F(m_0)$ are malicious, i.e., $MF(m_0) \equiv F(m_0)$. In this procedure, in each iteration we have a reference node m_i with a known (given or calculated) malicious fault set $MF(m_i)$ and a node m_j from the set of successors of m_i for which the malicious fault set $MF(m_j)$ should be calculated. Let $z(m)$ be the variable and $E(m)$ the expression which serve as the label at the node m with the fault list $F(m)$, i.e., the faults $f \in F(m)$ are manifesting themself as erroneous changes in the values of $z(m)$ or $E(m)$.

During the MFL generation, fault collapsing is carried out whenever possible, to minimize the whole fault set to be injected. The following rule, based on the dominance relationship between faults, is adopted for the malicious fault set reduction: if the node $m_j \in FT$ is a successor of the node $m_i \in FT$ and the fault $f_j \in MF(m_j)$ dominates $f_i \in MF(m_i)$ (i.e., causes the same erroneous change at $z(m_i)$ as f_i), then the fault f_i has to be removed from $MF(m_i)$.

The collapsed malicious fault set for the given fault tree FT is calculated according to the algorithm presented in Fig. 6.

1. Take the root node as the reference node m_i
2. If there is a node m among successors of m_i where $MF(m)$ is not yet created, take it as the node m_j to be inspected; else, if for all m the set $MF(m)$ is already created, go to step 5.
3. Include into $MF(m_j)$ all $f_j \in F(m_j)$, which are dominant over at least one $f_i \in F(m_i)$. If for $f_i \in F(m_i)$ at least one dominant fault $f_j \in F(m_j)$ was found, mark $f_i \in F(m_i)$ as *collapsed* (if it does not have this mark yet)..
4. For all $f_i \in F(m_i)$ not marked as collapsed, try to find a new f_j at m_j, which was not included into $F(m_j)$. If such a fault is found, include it into $MF(m_j)$ and mark f_i as collapsed (if it doesn't have this mark yet). Go to 2.
5. Include into $MF(m_i)$ all $f_i \in F(m_i)$ which are not marked as collapsed. Mark the node m_i as *investigated.*
6. Take a node which is not a leaf and whose predecessor is marked as investigated as the reference m_i, and go to 2; else, if there are not left such nodes go to *END.*
7. *END.*

Fig. 6. Fault collapsing algorithm

All the not empty sets of malicious faults $MF(m)$ at nodes of the fault tree FT represent the faults to be injected at corresponding variables (fault sites) $z(m)$.

Different steps of MFL generation and fault collapsing are illustrated in Fig.7. Arrow 1 illustrates step 3 of finding dominant faults over $f_i \in F(m_i)$, arrow 2 illustrates step 3 of collapsing faults in $F(m_j)$, arrow 3 illustrates step 4 of finding new faults not existing in $F(m_j)$ faults for including them into $MF(m_j)$, and arrow 4 illustrates step 5 of fixing $F(m_i)$. The collapsed parts of fault sets are colored.

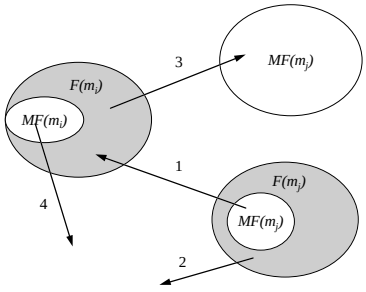

Fig. 7. Fault collapsing.

For fault backtracking and fault collapsing, combination of the following possibilities is used.

1. Local fault simulation. A novel approach of local fault simulation based on using AGs is proposed. The idea lies in inserting one by one the faults $f_j \in F(m_j)$ into the graph $G_z(m_i)$ for the reference variable $z(m_i)$ for tracing the graph and for checking if the fault causes the same error as some of the faults $f_i \in F(m_i)$. As all the possible faults are explicitly determined at nodes of AGs, the local simulation with the goal of establishing fault dominance between f_j and f_i can be very efficiently done.

2. Critical path tracing for fault collapsing in logical circuits [2]. In [10], a method for increasing the efficiency of backtracking by using AGs is given.

3. Reverse implication through functional blocks [8].

4. Hierarchical fault collapsing by combining fault simulation with reverse implication. It is a new approach presented here for hierarchically checking the dominance between faults which are determined at different levels. Suppose the faults $f_i \in F(m_i)$ at m_i are given through a domain of arithmetical differences $z(m_i)^* + \varepsilon$, where $z(m_i)^*$ is the expected value of the reference variable $z(m_i)$. Suppose also that m_j is labeled by an expression $E(m_j)$ representing a component in the system whose implementation details are known. In such a case, the Fault List $F(m_j)$ can be derived based on the low-level description. This list can be represented as set of erroneous values of $E(m_j)$ found by local simulation for all the given low-level faults. By using the values in $F(m_j)$, the given domain $z(m_i)^* + \varepsilon$ can be updated to a new domain $z(m_i)^* + \varepsilon'$ which is covering the values in $F(m_j)$. The new domain $z(m_i)^* + \varepsilon'$ is used further on in the procedure of reverse implication.

In Tab.2 the results of fault collapsing for the example of Fig.2 are presented. In column 3, initial fault sets $F(m)$ are given. At 5,6,14,15 no faults are listed, as they have to be determined by reverse implication. The sets R at 3 and 10 refer to fault lists to be derived from the lower level implementation. Column 5 contains the reduced fault lists as results of the local simulation. The analysis affects nodes 4,7,8,11. E.g., when simulating the faults at $m=7$, it comes out that none of these faults can cause an error at $j' \equiv z(4)$. As a result, $MF(7)$ will be empty, and continuing the tree from the node 7 on is not needed. So, $MF(11)$ will be empty too. Column 6 contains results which reveal that all the faults in $F(m)$ for m = 2,4,8,9 are dominated by faults at successor nodes. All these fault lists are therefore collapsed. Column 7 contains results where nodes 1,5,6, 10,14,15 have been affected. Hierarchical analysis is carried out at 3 and 10. For R, equivalent faults are deter-

#	Node labels	Fault lists F(m)	Faults in F(m)	MFL after local simulation	MFL after fault collapsing	MFL after reverse implication	Control faults in MFL	Data faults in MFL
1	t = 20	20 + ε	| ε |			-		
2	j' ≤ 1	0	1	0	-			
3	t'+in(1)	R	| R |			-		
4	j' = 1	0,2	2	2	-			
5	t' = 10				-	-		
6	in(1)=10					10 + ε		| ε |
7	j' ≤ 1	0	1	-				
8	j' + 1	0,2	2	2	-			
9	j' ≤ 1	0	1	0	-			
10	t'+in(0)	R	| R |			-		
11	j' = 0	1,2	2	-				
12	j' = 0	1,2	2	1,2	1,2		2	
13	j' = 0	1,2	2	2	2		1	
14	t' = 0					ε		| ε |
15	in(0)=10					10 + ε		| ε |
Σ_{CF} Σ_{DF}			13 7| ε |				3	3| ε |

Tab. 2: Fault collapsing application experiment.

mined and included, correspondingly, into *F(5)* and *F(6)* (for the node 3), and into *F(14)* and *F(15)* (for 10). As a result of fault collapsing, the numbers of faults in the control and data parts were reduced, correspondingly, from 13 to 3 (reduction of 76%) and from $7^*|\varepsilon|$ to $3^*|\varepsilon|$ (reduction of 57%). The value of ε is left open. Nodes 6,12,13,14,15 (the sites to be fault injected) are colored.

6. Translating collapsed FL into low-level FL

In this phase, the main problem is to maintain a close correspondence between the high-level and the low-level descriptions, from the following points of view.

Injection time translation. It is necessary to find a good correspondence between the *time* defined at the high-level and the *time* defined at the low-level.

Injection space translation. It is necessary to collect all the possible information about mapping between high-level variables and the lower-level implementation. For example, at the assembly level, allocation and scheduling table should be provided. Therefore all the information about how the synthesizer manages variables (*compile rules*, *name-mapping*, etc.) has to be obtained from the tool (or person) that performed the synthesis step.

7. Conclusion

In this paper a new technique is proposed to select malicious faults for dependability validation of fault-tolerant systems. A high-level fault analysis method based on alternative graphs is applied. Thanks to the generality of AGs, the present approach is independent from the fault model. This independence allows easy introduction of hierarchy and multilevel fault handling.

The main idea of the paper is to carry out fault analysis and malicious fault list generation at a higher behavioral level where the complexity of the model is very low. The high-level reduced fault list will be translated then into a lower-level fault list. In such a way, low-level fault analysis in low-level circuit descriptions can be avoided.

The method proposed in this paper allows explicit handling in a uniform way of both data and control faults. Instead of expanding the full explicit data flow whose size can explode, fault tracing is carried out on the compressed high-level AG-model to build up a fault tree. A new result is also integration of different methods for fault collapsing. On a simple high-level example it was shown that the new method allowed to reduce the malicious fault candidate list more than 4 times for control faults and more than twice for data faults.

8. References

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, D. Powell, *Fault injection for Dependability Validation: A Methodology and Some Applications*, IEEE Transactions on Software Engineering, Vol.16, No. 2, February 1990, pp. 166-182

[2] M. Abramovici, P.R. Menon, D.T. Miller, *Critical Path Tracing: An Alternative to Fault Simulation*, IEEE Design & Test of Comput., Vol.1, No. 1, pp. 83-93

[3] J. Karlsson, U. Gunneflo, P. Liden, J. Torin, *Two Fault Injection Techniques for Test of Fault Handling Mechanisms*, IEEE ITC, 1991, pp. 140-149

[4] G.A. Kanawati, N.A. Kanawati, J.A. Abraham, *FERRARI: A Flexible Software-Based Fault and Error Injection System*, IEEE Transactions on Computers, Vol 44, N. 2, February 1995, pp. 248-260.

[5] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson, *Fault injection into VHDL Models: the MEFISTO Tool*, Proc. FTCS-24, Austin (USA), 1994, pp. 66-75

[6] J. Laprie, *Dependable Computing and Fault Tolerance: Concepts and Terminology*, Proc. IEEE FTCS-15, pp.2-11, June 1985

[7] V. Pitchumani, P. Mayor, N. Radia, *Fault Diagnosis Using Functional Fault Model for VHDL Descriptions*, Proc. of the IEEE ITC, Nashville, 1991, pp.327-337

[8] D. Todd Smith, B. Johnson, J. Profeta III, D. Bozzolo, *A Fault-List Generation Algorithm for the Evaluation of System Coverage*, Proc. Reliability and Maintainability Symposium, 1995, Washington, pp. 425-432.

[9] R. Ubar, *Alternative Graphs and Technical Diagnosis of Digital Devices*¸ Electronic Technique, Vol. 8, No. 5, May 1988, pp. 33-57 (in Russian).

[10] R. Ubar, *Fault Diagnosis in VLSI Devices*, Proc. Estonian Acad. Sci. Eng., 1995, Vol.1., No 1, pp. 51-67.

[11] R. Ubar, *Test Synthesis with Alternative Graphs*, IEEE Design & Test of Computers, 1996 Spring, pp. 48-57.