Mutation Testing Applied to Hardware: the Mutants Generation

Mutation Testing Applied to Hardware: the Mutants Generation

T. B. Nguyen, C. Robach
LCIS-ESISAR, BP 54, 50 rue B. de Laffemas, 26902 Valence, France
Binh.Nguyen-Thanh@esisar.inpg.fr
Chantal.Robach@esisar.inpg.fr

Abstract

The Al-Hayek's testing approach consists in adapting the mutation testing, a software test method, to hardware. To apply this approach, in this paper, we present a mutants generator and a test data generator for circuit descriptions in VHDL. In the mutants generator, we implement the enhancement process that allows design validation to be efficiently reused for hardware testing, and we implement the selective mutation to reduce the test cost. We also present both random and deterministic test data generator.

1. Introduction

During the design process of complex systems, the validation is an expensive and difficult problem. So it is necessary to develop methods and tools that allow these validation costs to be reduced. The validation particularly lies on the test activities, in which the aim is to reveal specification and coding faults that are possibly present in the system.

In hardware systems, the testing methods aim at detecting physical or logical faults. However, the functional level allows us to aim at other objectives such as the design validation: this is very important as it is very expensive and difficult to find a design fault when the system is already operational. Many research works proposed methods to generate test data from the behavioral description of a circuit. Hansen and Hayes [1] [2] proposed the SWIFT algorithm to generate test data from the RTL (Register Transfer Language) descriptions. This algorithm guarantees a full low-level fault coverage when covering functional faults. However, it depends on the implementation and the technology of the circuit under test. Lin and Su [3] proposed an approach for test generation of RTL descriptions. The main advantage of this approach is the use of a symbolic execution technique that permits to generate test data of RTL descriptions. However, it suffers from being computationally complicated on VLSI circuits. Levendel and Menon [4] proposed an extension of the D-algorithm to generate test data for CHDL (Computer Hardware Description Language) descriptions. The considered fault modes are function variables stuck at 0 or 1, control faults (which affect conditional transfers in the CHDL description), and general function faults (which cannot be modeled by stuck variables or control faults). The advantage of this technique is the adaptation of the D-algorithm to higher level descriptions. But it remains impractical on complex VLSI circuits because of the cost of test generation. Barclay and Armstrong [5] proposed an approach similar to Levendel and Menon's approach for test generation of VHDL descriptions. The considered faults are the micro-operations faults (which affect logic, relational and arithmetic operations) and the control faults (which affect VHDL control constructs). This approach has the advantage of using a simple fault model and artificial intelligence techniques of goal trees to implement the test generation algorithm. However, it still has a high cost for complex VLSI circuits. Most of these methods are not very efficient when dealing with complex VLSI circuits. For this reason, Al-Hayek [6] proposed to adapt the *mutation testing* technique, which is originally used in software testing, to hardware. Mutation testing involves injecting simple faults in an original program in order to produce a set of erroneous versions, called *mutants*, and comparing the results of these mutants to the results of the original program executed on a set of test data. The mutation testing method can be applied on VHDL descriptions to generate the set of test data. When the circuit is implemented, this same set of test data is used to detect possibly present physical/logical faults in the circuit. In this paper, we present a tool whose aim is to implement mutation testing on hardware circuits.

This paper is organized in six sections. The following section presents the mutation testing technique. Section 3 presents the adaptation of this technique to hardware. The mutants generation and the data test generation are described in sections 4 and 5 respectively. Finally, a conclusion is given in section 6.

2. Mutation testing overview

The fundamental idea of this method is to find a set of test data that reveals present faults in a given program by supposing that a fault is a single modification in the "correct" program.

Mutation testing helps the tester to create and to improve a set of test data. In the process of mutation testing, the faults are injected in the original program to create erroneous versions called *mutants*. A mutant is

VLSI-SOC 2001
11th International Conference on Very Large Scale Integration
december 3-5, Montpellier, France
2-9517461-0-5

generated by only one *mutation operator*. A mutation operator is a single syntactic modification that is made to a statement of the original program. A mutant is said to be *killed* by a test data if it produces a result that is different from the result of the original program when executed on that test data. If it is not killed, it is *live*: in this case, either the set of test data is not sufficient to kill it, so it is necessary to rebuild the set of test data; or it is said to be *equivalent* to the original program. This method is based on the three following hypotheses [7]:

- The *competent programmer hypothesis*: a competent programmer tends to write programs that are "close" to being correct.
- The *coupling effect*: a set of test data that distinguishes all programs with simple faults is so sensitive that it will also distinguish programs with more complex faults.
- The *oracle hypothesis*: the assumption is that there is an oracle to determine the program correctness.

Figure 1 shows a function with three mutations. The lines preceded by the Δ symbol are the statements mutated by the mutation operators. Note that each mutated statement represents a mutant. In function MIN written in the C language, the mutation operator replaces each operand by another legal operand (Δ3) and modifies the expression by replacing an operator by the other legal operators (Δ1 and Δ2). Note that the mutant created by Δ2 is always equivalent to the original program whatever the set of test data.

```
MIN(int M, int N)
{
    int MinVal;
    if (M > N)
Δ1  if (M < N)
Δ2  if (M >= N)
            MinVal = N;
    else
            MinVal = M;
Δ3          MinVal = N;
    return MinVal;
}
```

Figure 1. Function MIN with three mutations

Demillo and *al.* first introduced this method in [8] and it is called *strong mutation testing*. They developed a system, called Mothra, [9] which is based upon 22 mutation operators. This system builds up the set of mutants and it computes the set of test data necessary to kill the mutants. But its complexity is high, since the number of mutants that can be generated is very large (this number is on the order of O(*Lines***Lines*), where *Lines* is the number of lines in the program [10]). So the methods proposed to overcome this cost include *weak mutation* and *selective mutation*. All of these methods are variations of strong mutation testing. Weak mutation [11] does not consider the behaviour of the whole program but it only considers the behaviour of components in the program; these components consist of variable reference, variable assignment, arithmetic expression, relational expression and boolean expression.

Selective mutation is based on the idea that the mutation operators, which produce the highest number of mutants, are omitted. First, Mathur [12] proposed to delete two mutation operators SVR (Scalar Variable Replacement) and ASR (Array reference for Scalar variable Replacement); then Offutt and *al.* [13] extended this approach by eliminating more mutation operators: *N-selective mutation*, *i.e. N* mutation operators ($N = 2, 4, 6$) are deleted. Both Mathur and Offutt and *al.* suggested that the two mutation operators, which should be omitted, are SVR and ASR. Empirical studies in [12] [13] [14] show that selective mutation reduces the testing cost considerably but still ensures a very high mutation score.

3. Mutation testing for hardware

In this section, we present the mutation testing adaptation to hardware proposed by Al-Hayek [6]. First we present the analogy between the VHDL hardware description language and usual programming languages.

3.1. Analogy between VHDL and usual programming languages

VHDL is a hardware description language. It was developed to describe electronic components. However, it has characteristics that are close to the characteristics of usual programming languages such as Pascal or Ada. In VHDL, an electronic component can be described as a module (a program) that consists of two parts: the inputs and outputs declaration, and functional or the behavioural description. The VHDL description uses types (bit, bit_vector, boolean, integer...), structures (if-then-else, case...), operators (arithmetic, relational...) ... The following example (Figure 2) shows this analogy:

```
entity MIN_ENT is
      port (M, N : in INTEGER;
            MIN : out INTEGER);
end MIN_ENT;
architecture BEHAVIOR of MIN_ENT is
begin
   process (M, N)
   begin
         if N<M then
               MIN <= N;
         else
               MIN <= M;
         end if;
   end process;
end BEHAVIOR;
```

Figure 2. The entity MIN_ENT

This analogy allows the distinction between hardware and software testing to be reduced. Therefore, software testing can be used to detect faults on VHDL functional descriptions and this also permits to detect hardware faults of electronic components.

3.2. Mutation testing for VHDL descriptions

Al-Hayek [6] states that VHDL functional descriptions are written, verified and corrected like usual programs, so that an error in a VHDL functional description has the same causes and effects than one in software. Then, he proposes to use software mutation testing to test VHDL functional descriptions. In mutation testing, the fault model is represented by the mutation operators set. Each mutation operator defines a fault class. Al-Hayek defines the fault model for VHDL functional descriptions by choosing a mutation operators subset used for software testing. This subset consists of the ten mutation operators shown in Table 1.

Table 1. Mutation operators for VHDL

Type	Description
AOR	Arithmetic operator replacement
ABS	Absolute value insertion
CR	Constant replacement
CVR	Constant for variable replacement
LOR	Logical operator replacement
ROR	Relational operator replacement
NOR	No operation replacement
VCR	Variable for constant replacement
VR	Variable replacement
UOI	Unary operator insertion

To test a VHDL description, mutation testing lies on faults injection to generate mutants of the description and a set of test data that kills a maximum of these mutants. Fault injection is realized by applying a mutation operator to the description. Furthermore, the mutation method must consider hardware characteristics such as the bit-width of data, which has an impact on the hardware area and thus the test size. So, Al-Hayek introduced an enhancement process taking into account this characteristic. Mutation operators are divided into simple and complex ones (arithmetic, relational and logical operators). We illustrate the mutation method in Figure 3.

```
entity MIN_ENT is
    port (M, N : in INTEGER;
          MIN : out INTEGER);
end MIN_ENT;
architecture BEHAVIOR of MIN_ENT is
begin
    process (M, N)
    begin
            if N<M then
Δ1          if N>M then
                    MIN <= N;
Δ2                  MIN <= abs(N);
            else
                    MIN <= M;
Δ3                  MIN <= N;
            end if;
    end process;
end BEHAVIOR;
```

Figure 3. Some mutants of entity MIN_ENT

To validate this approach, Al-Hayek proposed the validation process shown in Figure 4. In this validation process, a synthesis tool is used to generate the gate level structure from the VHDL description. To measure the quality of the set of test data generated by mutation testing, it is necessary to have a gate level fault simulator. This fault simulator calculates the Mutation Fault Coverage (MFC) of hardware faults ensured by a set of test data.

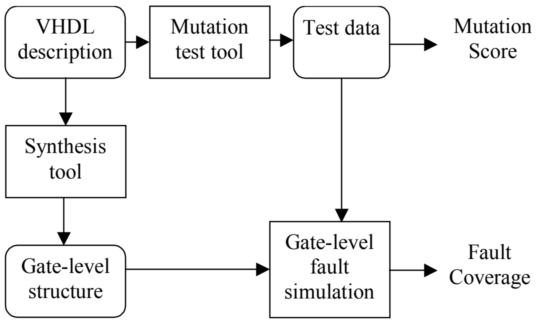


Figure 4. The validation scheme

To apply Al-Hayek's approach, we present the design of the mutants generation and the random test data generation for VHDL descriptions.

4. Mutants generation system

This mutants generation system aims at being used for several languages, such as C/C++, VHDL... In this paper, we particularly consider mutants generation for VHDL. In addition, to reduce the cost, selective mutation will be applied to this system.

4.1. System architecture

We propose a general architecture of the mutants generation system which is independent from the used language. In this architecture (Figure 5), we distinguish two parts: the *lexical/syntactic analyzer* and the *mutants generator*.

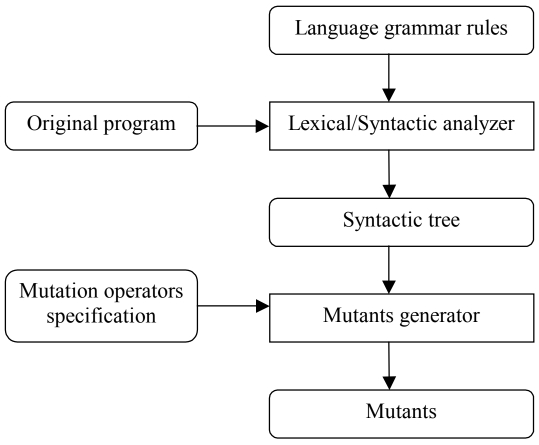


Figure 5. General architecture

We illustrate this architecture by the following simple example (Figure 6). Let us suppose that our program is: “a = b + c * d” and two mutants are created: “a = b - c * d” and “a = b + c / d”.

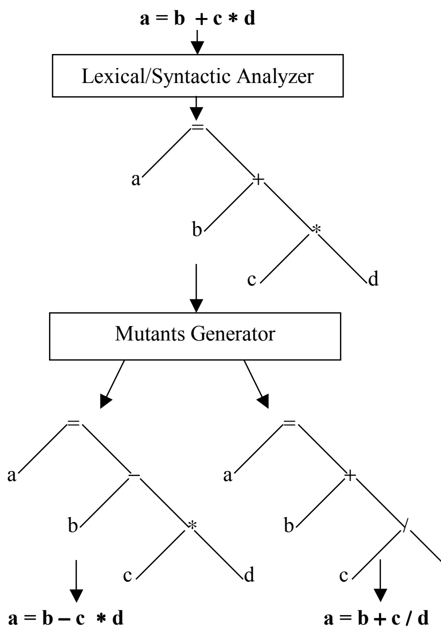


Figure 6. An example of the mutants generation

4.2. Lexical/syntactic analyzer

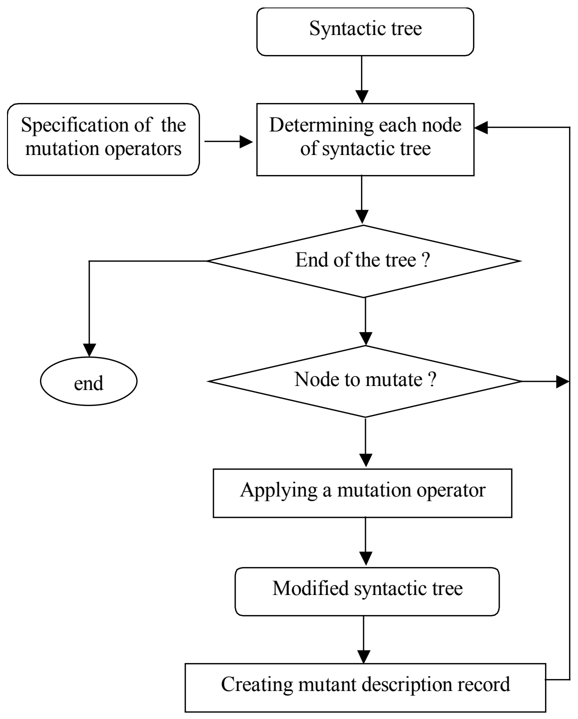
The inputs of the lexical/syntactic analyzer are the original program and the language grammar rules. It is necessary to use Flex [15] and Bison [16] tools, originally used to build compilers, to produce a syntactic tree of this original program. Flex is a tool that generates lexical analyzer or scanner: this program recognizes strings that match patterns in text and converts the strings to tokens. Bison is a tool that generates syntactic analyzer or parser : it uses grammar rules to analyze tokens from Flex and create a syntactic tree. Therefore, these two tools are used to build a syntactic tree from a VHDL functional description.

Fortunately, there is the FreeHDL project [17] that develops a VHDL simulator whose source is open. In fact, it provides the FIRE (Feeble Intermediate Representation with Extensibility) library that helps to transform a VHDL description into a corresponding syntactic tree. So we will take advantage of this library to generate a syntactic tree of a VHDL description.

4.3. Mutants generator

After having analyzed VHDL description, we get the corresponding syntactic tree. This tree is then mutated by applying mutation operators. Mutants are generated from mutated syntactic trees.

The algorithm of the mutants generator is given in Figure 7.



Syntactic tree
Specification of the mutation operators
Determining each node of syntactic tree
End of the tree ?
end
Node to mutate ?
Applying a mutation operator
Modified syntactic tree
Creating mutant description record

Figure 7. Algorithm of mutants generation

The inputs of the mutants generator are a syntactic tree and the specification of the mutation operators. As the system’s goal is to tackle several languages, the specification of mutation operators depends on the used language and on the chosen type of mutation (strong, weak, selective). In this paper, we only consider VHDL descriptions, and the specification of mutation operators consists of the ten mutation operators for VHDL defined in Table 1.

The syntactic tree is a hierarchical structure; we traverse each node of the tree, and we verify if it is a node for applying a mutation operator; if so, we apply the mutation operator in order to generate the mutant. This process is continued until the end of the syntactic tree.

In addition, when generating the mutants, we consider the enhancement process. For that, we have to compute the operands size for logical, relational and arithmetic operators that are complex operators, which influence the hardware implementation of the component. To kill a complex mutant generated by a complex operator, we have to generate N test cases (and not only one). This number N essentially depends on the operands size.

To reduce the cost, we propose to apply selective mutation. This is easily integrated by modifying the specification of mutation operators. The user can define the specification of mutation operators according to the VHDL description.

4.4. Results

This mutants generator was applied to the descriptions of sequential circuits: b01.vhdl, b02.vhdl, b03.vhdl from the ITC'99 benchmarks of Turino Polytechnic Institute. We obtain the following results:

Table 2. Results of the application of the mutants generator

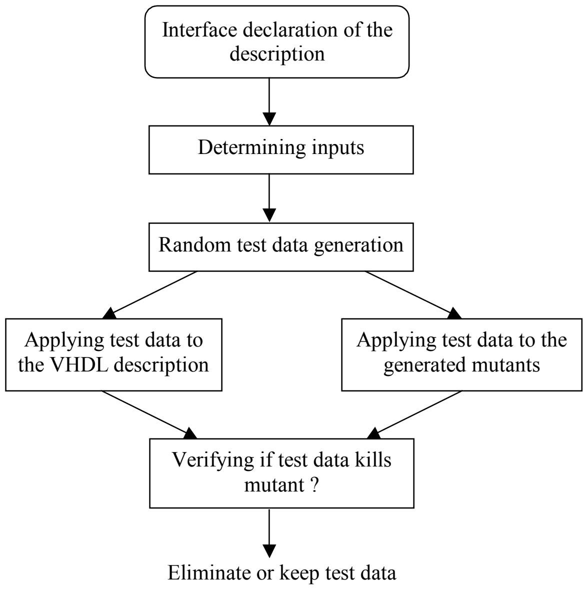
Circuit descriptions	Number of generated mutants	Number of generated complex mutants
b01.vhdl	682	158
b02.vhdl	106	29
b03.vhdl	814	71

5. Test data generation

One of the difficulties of the mutation testing application is the generation of effective test data (*i.e.* killing a maximum of generated mutants for a program). In this section, we propose two approaches: random and deterministic test data generation.

5.1. Random test data generation

This first approach is to randomly generate test data of a VHDL description in its inputs domain. This approach does not consider the internal structure of the VHDL description, so the generated test data are not enough to kill all the mutants. This approach is illustrated in Figure 8.



Interface declaration of the description
Determining inputs
Random test data generation
Applying test data to the VHDL description
Applying test data to the generated mutants
Verifying if test data kills mutant ?
Eliminate or keep test data

Figure 8. Random test data generator

A VHDL description contains the interface declaration of its inputs and outputs. To generate test data of a description, we only consider its inputs (*i.e.* type and domain of each input). After randomly generating test data, we use a filter to select only the effective test cases that kill at least one mutant. For that, we have to execute the original VHDL description and each of its mutants on each test case, and then we compare their final results to decide which test case is ineffective.

5.2. Deterministic test data generation

The second approach is to generate test data effectively by considering the structure of the mutants of the VHDL description. This approach proposed in [18] is based on constraints to generate test data that makes a difference in the mutant's behaviour. We present this approach to generate test data for a VHDL description.

The first constraint is that, for a test case to kill a mutant, it is necessary that the states of the mutant and the VHDL description differ after the last execution of the mutated statement. This characteristic is said the *necessity condition*. For a test case to kill a mutant, it must create an incorrect output, in which case the final state of the mutant differs from that of the VHDL description. This second constraint is said the *sufficiency condition*.

These constraints are represented by the algebraic expression, which is composed of variables, parentheses and operators. Expressions are taken directly from the test program.

The generator architecture is represented in Figure 9. For each statement, the *path analyzer* uses path-coverage techniques to build the *path expression* constraint such that the test case reaches that statement. The *constraints generator* aims at building up constraints for each mutant. The *constraints satisfier* generates a test case, which satisfies the constraints.

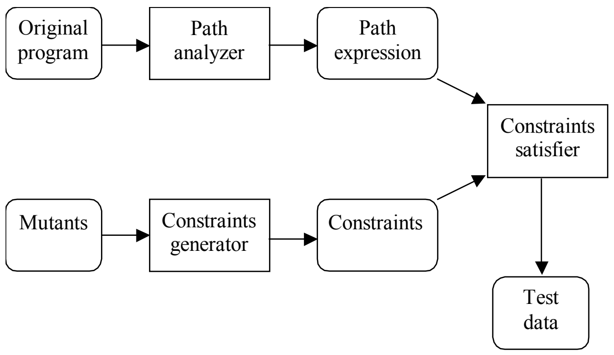


Figure 9. Deterministic test data generator

In [18], the tool, called Godzilla, was implemented to generate test cases for Fortran 77 programs. But the general concept is independent from the used language; so we will implement a tool that uses this approach to generate test data for a VHDL description in a future work.

6. Conclusion

In this paper, we reviewed the method of mutation testing, developed for the software, and also the way in which this technique may be applied to hardware. Indeed, by using high-level design tools, a functional description of an electronic system in a hardware description language such as VHDL can be considered as a software piece. So detecting logical faults in a circuit may be reached by detecting software faults in its functional / behavioural description.

To apply this approach, we developed two tools: a mutants generator and a test data generator. The mutants generator is rather a "compiler" which generates mutants (VHDL descriptions) by applying mutation operators (strong, weak, selective mutation) to the VHDL description under test. Our ultimate goal is to apply this to several languages.

We also developed a random test data generator for a VHDL description that must be complemented by a deterministic test data generator using algebraic constraints.

7. References

[1] M.C. Hansen and J.P. Hayes, "High-level test generation using physical-included faults", *IEEE VLSI Test Symposium (VTS'13)*, Princeton, NJ (USA), pp. 20-28, 1995.

[2] M.C. Hansen and J.P. Hayes, "High-level test generation using symbolic scheduling", *International Test Conference (ITC'95)*, Washington, D.C. (USA), pp. 586-595, 1995.

[3] T. Lin and S.Y.H. Su, "Functional Test Generation of Digital LSI/VLSI Systems Using Machine Symbolic Execution Techniques", *IEEE International Test Conference*, pp. 660-668, 1984.

[4] Y. Levendel and P. Menon, "Test Generation Algorithms for Computer Hardware Description Language", *IEEE Transactions on Computers*, vol. C-31, No 7, pp. 577-588, 1982.

[5] D. Barclay and J. Armstrong, "A heuristic chip-level test generation algorithm", *23rd Design Automation Conference*, pp. 257-262, 1984.

[6] Ghassan Al-Hayek, "Vers une Approche Unifiée pour la Validation et le Test de Circuits Intégrés Spécifiés en VHDL", *Thesis*, Institut National Polytechnique de Grenoble, 1999.

[7] A. Jefferson Offutt "The Coupling Effect: Fact or Fiction", *3rd Symposium on Software Testing Analysis and Verification*, ACM-SIGSOFT, Keywest (Florida US), pp. 131-140, 1989.

[8] Richard A. DeMillo and al., "Hints on test data selection: help for the practising programmer", *IEEE Computer*, Vol. 11, No. 4, pp. 31-41, 1978.

[9] K. N. King and A. Jefferson Offutt, "A Fortran Language System for Mutation based Software Testing", *Software-Practice and Experience*, Vol. 21(7), pp. 685-718, 1991.

[10] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Mutation Analysis", Georgia Institute of Technology, *Technical Report*, GIT-ICS-79/08, 1979.

[11] W.E. Howden, "Weak Mutation Testing and Completeness of Program Test Sets", *IEEE Transaction on Software Engineering*, Vol. SE-8, No. 4, pp. 162-169, 1982.

[12] M.P. Mathur, "Performance, effectiveness, and reliability issues in software testing", *15th Annual International Computer Software and Application Conference*, Tokyo, Japan, pp. 604-605, 1991.

[13] A. Jefferson Offutt, Gregg Rothermel and Christian Zapf, "An Experimental Evaluation of Selective Mutation", *In Proceeding of the 1993 International Symposium on Software Testing*, IEEE Computer Society Press, Los Alamitos Calif., pp. 100-107, 1993.

[14] A. Jefferson Offutt and *al.*, "An Experimental Determination of Sufficient Mutant Operators", *ACM Transaction on Software Engineering and Methodology*, Vol 5, No. 2, pp. 99-118, 1996.

[15]Vern Paxson, "*Flex, version 2.5*", delivered with software, 1995.

[16] Charles Donnelly et Richard Stallman, "*Bison*", delivered with software, 1999.

[17] Marius Vollmer, "*FreeHDL*", http://www.freehdl.seul.org.

[18] Richard A. DeMillo and A. Jefferson Offutt, "Constraint-Based Automatic Test Data Generation", *IEEE Transaction on Software Engineering*, Vol. 17, No. 9, pp. 900-910, 1991