Software mutation testing: towards combining program and model based techniques

M. Forostyanova
Department of Information technologies
Tomsk State University
Tomsk, Russia
mariafors@mail.ru

N. Kushik
Department of Information technologies
Tomsk State University
Tomsk, Russia
ngkushik@gmail.com

Abstract — **The paper is devoted to the mutation testing technique that is widely used when testing different software tools. A short survey of existing methods and tools for mutation testing is presented in the paper. We classify existing methods: some of them rely on injecting bugs into a program under test while other use a formal model of the software in order to inject errors. We also provide a short description of existing tools that support both approaches. We further discuss how these two approaches might be combined for the mutation based test generation with the guaranteed fault coverage.**

Keywords — *Software testing, mutation testing, mutation operator, model based testing, fault coverage.*

I. INTRODUCTION

As the number of widely used information systems increases quickly, the problem of software testing becomes more important. Thorough testing is highly needed for software being used in critical systems such as telecommunications, banking, transportation, etc. [1]. An approach for mutation testing has been proposed around thirty years ago but there still remain some issues of this approach waiting for new effective solutions. Those are fault coverage, equivalent mutants, etc. that we further discuss. In order to solve these problems model based methods for mutation testing are now appearing. In this paper, we make an attempt to follow the chronology of mutation software testing. We start with the initial methodology of mutating programs and further turn to model based mutation testing techniques. In both cases, we provide a brief description of tools which are developed for program/model based mutating testing.

As mentioned in [2], the mutation testing has been introduced by a student Richard Lipton in 1971 [3] while the first publication in this field has been prepared by DeMille, Lipton and Sayward [4]. Meanwhile, the first tool for mutation testing has been developed by Timothy Budd ten years later, in 1980 [5]. Next twenty years the popularity of mutation testing techniques did not grow rapidly while after Millennium it became more and more popular. Moreover, in 2000 the first complete survey of existing methods for mutation testing has appeared [3]. During the last decade there appeared more than 230 publications on mutation testing [2] and almost all existing tools rely on injecting errors in a program under test. One may turn to [2] to find various papers, PhD theses, etc. combined together into one large repository [6], where the authors make an attempt to cover mutation testing evolution from 1977 till 2009. However, there exist much less publications on model based mutation testing and much less tools that support corresponding formal methods.

In this paper, we first discuss mutation testing technique when a program is mutated by injecting bugs into it. In this case, a program with an injected bug is called a *mutant*. If the behavior of the program is not changed after an injected bug then such injection leads to an *equivalent* mutant. Most of existing tools are developed for software that is written in high level language, and thus, mutation operators are often adapted to language operators. Moreover, ‘good’ tools usually inject those errors that programmer often ignores in his/her programs.

When deriving a mutant based test suite two ways are often used. The first way is to randomly generate test sequences and to check which mutants (errors) are detected by these sequences. Another option is to generate mutant based test sequences such that all the injected errors are detected. The first approach is mostly used when a model is the program itself while the second approach is used more rarely and deals with the formal specification of the program.

Considering the first approach there exist tools that are able to inject bugs into programs written in Fortran [see, for example 7], C/C++ [see, for example 8], Java [see, for example 9], and an SQL code [see, for example 10]. As for the second approach, there exist tools that are developed for injecting errors into software specifications on different abstraction level such as Finite State Machines [see, for example 11], State charts [see, for example 12], Petri Nets [see, for example 13], XML-specification [14]. In this paper, we discuss a number of methods and tools for mutation testing and divide the paper into two parts. The first part is devoted to the *program* based mutation testing where bugs are injected into programs and in the second part the *model* based mutation testing is discussed.

The rest of the paper is organized as follows. Section II contains the preliminaries. Section III is devoted to the program based mutation testing. This section contains the description of a testing method when a program is mutated and a short description of tools for mutating programs written in

Java, C and SQL languages. The model based mutation testing is discussed in Section IV. This approach is illustrated for different kinds of program specifications such as finite state machines, XML-specifications, etc. A number of tools developed for injecting faults into the program specifications are also described. Section V discusses an approach of combining the program based mutation testing with the model based one. Section VI concludes the paper.

II. Preliminaries

As mentioned above, software testing becomes more and more important and there appear new methods and techniques for this kind of testing. Nevertheless, all methods for software testing can be implicitly divided into two large groups. Methods of the first group rely on the informal program specification or the informal software requirements while methods of the second group require a formal model to derive a test suite for a given program. The main advantage of the first approach is the speed of testing that might be rather high because of short length of test sequences and because of a the cardinality of a test suite. However, the main problem of this technique is the fault coverage that is not guaranteed. This problem can be partially solved for model based testing techniques where a test suite is derived based on the formal specification of a given program. This formal specification may be finite transition model [15], pre-post conditions [16], etc. However, the speed of the software testing may fall down exponentially since a long time is needed for deriving formal specifications as well as for deriving a test suite on the basis of this specification. Thus, a good compromise might be to combine somehow methods of the first and the second groups in order to increase the testing speed and to guarantee the fault coverage at least for some classes of program bugs.

Mutant based software testing is not an exception of this tendency and methods for mutation testing can be also implicitly divided into those that rely on a program itself and those that are based on the formal model of a given program. Hereafter, we refer to methods of the first groups as methods of *program based* mutation testing while methods of the second group are called *model based* methods. The main idea of the mutation testing is to change a program or a model in such way that this change corresponds to possible errors in program implementation. Another nontrivial task is to derive a test sequence or a test suite such that all 'inappropriate' changes could be detected by applying test sequences.

Each tool for the program based mutation testing relies on a set of mutation operators and this set describes types of errors that can be detected in the source code by a corresponding test suite. The bigger is the set of mutation operators the more test properties can be verified by these mutants.

In this paper, when discussing the program based mutation testing we consider programs written in high level languages like C + + and Java. We further turn to the model based mutation testing and consider finite state machines (FSMs) and extended FSMs (EFSMs) as formal specifications that are widely used for the software test derivation.

Model based testing allows to detect those implementation bugs that cannot be detected by random testing or other techniques of program based testing. Thus, in this paper we discuss which methods and tools are developed for the program based mutation testing as well as for the model based mutation testing. In Section V we make a step towards combining those methods, i.e., we establish a correspondence between software bugs (program mutants) and formal specification errors (model mutants).

III. Program based mutation testing

In case of the program based mutation testing mutated programs (mutants) are often used for evaluating the quality of a given test suite, i.e., a mutant is used for checking whether corresponding types of program bugs can be detected by the test suite or not. If some mutants cannot be detected or *killed* the test suite is extended by corresponding test sequences. This approach is illustrated in Fig. 1. [2]. One may turn to [2] to find out more about the scheme presented in Fig 1.

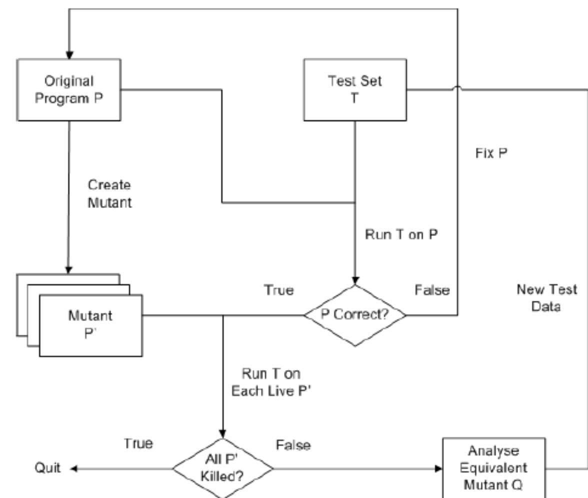

Fig. 1. Generic Process of Mutation Analysis

The program based mutation testing described above has been implemented as several software tools. Most of the tools are developed only for injecting bugs into a source code and only several tools support a test generation process. Moreover, almost every tool for program based mutation testing is commercial. We further provide a short description of existing tools for mutation testing of C/C++ and Java programs.

A. *Tools for C program Mutation testing*

Agrawal et al. [8] have proposed a comprehensive set of mutation operators for the ANSI C programming language in 1989. There are 77 mutation operators defined in this set. Moreover, Vilela et al. [17] proposed a number of mutation operators to represent bugs associated with static and dynamic memory allocations.

Now there exist a number of tools for injecting errors into C programs. Some of these tools are briefly presented in Table I.

TABLE I

A LIST OF TOOLS FOR PROGRAM BASED MUTATION TESTING FOR C++/C PROGRAMS

Name	Date of first release	Accessibility	Is improving currently	Features
PlexTest [18]	2005	The commercial product	+	Only an instruction removal is supported
Insure++ [19]	1998	The commercial product	+	Injects and detects bugs for identifiers, memory/stack bugs and bugs that concern to linking libraries
Proteum/IM 2.0 [20]	2000	The free download utility	-	Supports 71 mutation operators and calculates the number of mutants being killed
Certitude [21]	2006	The commercial product	+	Can be used for C/ C + + and HDL programs; Combines the mutation approach with the static code analysis; Allows to verify an environment of the program under test
MILU [22]	2008	The free download utility	+	Allows a user to choose the desired number of mutants and to specify their types; 77 mutation operators are supported

One may conclude from Table I that almost all existing tools are developed for injecting bugs, probably except of PlexTest, Insure++ and Certitude that also provide test generation. In spite of the fact that these products are commercial they do not guarantee the guaranteed fault coverage with respect to their specifications.

B. Tools for Java program mutation testing

As many Java programs support the object-oriented paradigm, tools that inject bugs into such programs are mainly concentrated on disturbing inheritance and/or polymorphism features.

Kim et al. [23] were the first to define mutation operators for Java programming language taking into account object-oriented paradigm. This team has proposed 20 mutation operators for Java programs. Moreover, Kim has introduced Class Mutations, which were divided into six groups: Types/Variables, Names, Classes/interface declarations, Blocks, Expressions and others.

Following the tendency from Section A we further describe several tools developed for Java program mutation testing. Differently from C based mutation testing most of the tools developed for the Java program mutation testing are distributed for free. A brief description of available tools is presented in Table II.

Looking at this table one may conclude that there exist tools that support injecting bugs concerned on encapsulation, polymorphism and inheritance. Those are MuJava, Javalanche, and Jester. Moreover, these tools support mutation testing based on corresponding mutation operators. However, the fault coverage of these tests remains unknown. One of the reasons could be the problem of equivalent mutants that are not automatically excluded from the mutants being generated. Therefore, the program based mutation testing needs to be extended with the formal specification of a program under test in order to provide the guaranteed fault coverage of a test suite.

TABLE II

A LIST OF TOOLS FOR PROGRAM BASED MUTATION TESTING FOR JAVA PROGRAMS

Name	Date of first release	Accessibility	Is improving currently	Features
Jester [24]	2001	The free download utility	+	Supports object-oriented mutation operators; Shows equivalent mutants to a user
MuJava [25]	2004	The free download utility	+	Supports 24 mutation operators which specify object-oriented bugs; Mutants are generated and executed automatically; Equivalent mutants have to be excluded manually
MuClipse [26]	2007	The free download utility (plugin for Eclipse)	+	This is the MuJava version developed for Eclipse
Javalanche [27]	2009	The commercial product	+	Detects around 10% of equivalent mutants; Allows a user to manipulate with a bytecode; Most mutants are concerned about the replacement of an arithmetic operator, constants,

				function calls; Can execute several mutations in parallel and might be used for testing parallel and distributed systems

IV. MODEL BASED MUTATION TESTING

The first steps in model based mutation testing have been made in 1983 by Gopal and Budd. They have proposed a technique for the software mutation testing describing software requirements taking into account the predicate structure of the program under test.

When generating a test using the model based mutation testing, errors are injected into the model, i.e. the model is mutated. Moreover, similar to the program based mutation testing equivalent mutants need to be deleted. The model based mutation testing has been studied for a number of formal models such as automata models [15], Petri nets, etc described in UML, XML, etc.

By the use of automata models such as FSMs, EFSMs, Petri nets, tree automata, labeled transition systems (LTS), etc. there were proposed a number of approaches for specifying informal software requirements. A number of mutation operators have been proposed for such finite state models. One may turn, for example, to [28] where the authors propose 9 mutation operators representing faults related to states, inputs and outputs of an FSM that is mutated. This set of mutation operators has been implemented in the tool PROTEUM [20]. In [29], the authors investigated an application of mutation testing for probabilistic finite automata (PFAs). They have defined 7 mutation operators and have specified a number of rules how to exclude equivalent mutants.

Mutation operators have been also defined for EFSMs in [30]. In this work, the author has discussed changing operators and/or operands in functions and predicates. Nevertheless, only some types of mutants are formally specified in this work. Thus, in our paper we make an attempt to classify EFSM mutants and to establish a correspondence between EFSM mutants and bugs in the corresponding software implementations.

Tree automata are also of a big help when dealing with software verification. Moreover, each tree automaton can be described as an XML document, and thus, a number of mutation operators is defined especially for XML-documents. One may turn to [14] where Lee and Offut discuss how to inject errors into XML-documents and how to apply this technique for mutation testing of web-servers. The authors have proposed 7 mutation operators and they have further extended their work in 2001 introducing a new approach to XML mutation. This work is based on deriving invalid XML-data using seven mutation operators. All the XML mutation operators introduced in [32] have been combined together and have been implemented in the tool XTM. XTM supports 18 mutation operators and allows to test XML-documents. Nevertheless, the authors of [31] ‘complain’ that only 60% of injected errors have been detected in their experiments.

V. ESTABLISHING A CORRESPONDENCE BETWEEN PROGRAM MUTANTS AND MODEL MUTANTS

In order to somehow combine methods and tools for the program based mutation testing with that based on formal models we are now interested in establishing a correspondence between bugs in a program under test and faults in a model of this program. We are planning to experimentally solve this problem and we focus on testing C/C++ programs. Moreover, we choose one of finite state models discussed above and define a number of mutation operators for this model. A model of an EFSM [32] is rather close to C/C++ implementation because it extends a classical FSM with input and output parameters and context variables. Predicates can be also specified in the EFSM model and a transition can be executed if a corresponding predicate is true. Thus, we establish a correspondence between bugs in C/C++ programs and EFSM faults. Such correspondence can be further used for deriving a test suite for C/C++ implementations based on program mutants but preserving the same fault coverage as if a test suite is derived based on a corresponding EFSM.

We first classify EFSM mutants and then establish to which C/C++ program errors they correspond to.

1. *Predicate* EFSM *mutant* is derived when a predicate formula is mistaken or the predicate is deleted, i.e. ,transition becomes unconditional.

2. *Transition* EFSM *mutant* is derived when a transition is deleted, unspecified transition is added to EFSM or the next state of some transition is wrong.

3. *Function* EFSM mutant occurs when changing a formula for calculating the next value of a context variable or an output parameter.

We now discuss which C/C++ bugs correspond to the above mutants.

A. *Predicate mutants*

Each EFSM predicate corresponds to a *switch/case* or *if/else* instruction of a corresponding C/C++ code, and thus, the following cases are possible.

1. An EFSM predicate is deleted and this fault corresponds to eliminating the *if/else* instruction from the C code.
2. An EFSM predicate consists of several conditions and one of these conditions is deleted. In this case, the corresponding C code contains a complex condition under *if* or *while* and one of its conditions is deleted.
3. Changing logical connectives of a predicate corresponds to a software implementation with an invalid condition.

4. An EFSM predicate can be also changed with respect to a corresponding formula, i.e., operators and/or operands may be changed. These changes correspond to the same changes under *if* or *while* conditions in the C code.

B. Transition mutants

This type of EFSM faults is rather difficult to correlate with C/C++ implementation changes. The reason is that this correspondence strongly depends on how states are defined in the program. If each EFSM state corresponds to one of special program state variable then EFSM transition mutant can correspond to changing the identifier of the next state in the program. If the transition is deleted in the EFSM then a corresponding instruction is deleted from the C/C++ code. Establishing such correspondence is much more difficult when state semantics is different in the program and by now this option is out of the scope of this paper.

C. Function mutants

When changing formula for calculating values of a context variable or output parameters corresponding C/C++ program is changed in the same way. Thus, EFSM function mutants correspond to those program mutants that are derived by changing corresponding operators and/or operands in the C/C++ instructions.

In Table III the correspondence between EFSM mutants and bugs in software implementations is presented.

TABLE III
A CORRESPONDENCE BETWEEN EFSM MUTANTS AND PROGRAM BUGS

Program bugs	EFSM mutants
Removal of instruction block **if / else**	Predicate mutant (a transition becomes unconditional)
Removal of a part of composite condition	Predicate mutant
Sign changing in an **if** condition	Predicate mutant
Operators and/or operands changes in an **if** condition	Predicate mutant
Change of identifier in an **if** condition	Predicate mutant
Return of a wrong variable from a function	Output mutant
Changing a sign in an arithmetic operation	Function mutant
Removal of instruction block after **if** (or **else**)	Transition mutant

VI. Conclusiuon

In this paper, we have discussed different methods and tools developed for the software mutation testing. The paper clearly shows that there exists a list of tools that support program based mutation testing when a bug is injected into the original program. Much less tools are developed for model based software testing in spite of the fact that this technique allows to guarantee the fault coverage of a test suite. As a result, we are planning to combine the program based mutation testing with the model based one in order to derive tests with the guaranteed fault coverage rather fast. For this purpose we have tried to establish a correspondence between program bugs and model mutants. Such correspondence can be further used for deriving a test suite for C/C++ implementations based on program mutants but preserving the same fault coverage as if a test has been derived based on corresponding EFSM. Developing such testing method based on this correspondence is an open problem for a future work.

References

[1] О. Г. Степанов. Методы реализации автоматных объектно-ориентированных программ. Диссертация на соискание ученой степени кандидата технических наук, СПбГУ ИТМО: 2009, 115 с.

[2] Yue Jia, M. Harman, IEEE, "An Analysis and Survey of the Development of Mutation Testing", pp 33

[3] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal,"In Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00), published in book form, as Mutation Testing for the New Century. San Jose, California, 6-7 October 2001, pp. 34–44

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, vol. 11, no. 4, pp. 34–41, April 1978

[5] T. A. Budd, "Mutation Analysis of Program Test Data," PhD Thesis,Yale University, New Haven, Connecticut, 1980

[6] Repository: [web-site]. URL: http://www.dcs.kcl.ac.uk/pg/jiayue/repository/, 2010

[7] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward,"Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," in Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'80), Las Vegas, Nevada, 28-30 January 1980, pp. 220–233

[8] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of Mutant Operators for the C Programming Language," Purdue University, West Lafayette, Indiana, Technique Report SERC-TR-41-P, March 1989.

[9] S. Kim, J. A. Clark, and J. A. McDermid, “Investigating the effectiveness of object-oriented testing strategies using the mutation method,” in Proceedings of the 1st Workshop on Mutation Analysis (MUTATION’00), published in book form, as Mutation Testing for the New Century. San Jose, California, 6-7 October 2001, pp. 207–225.

[10] SQLmutation : [web-site]. URL: http://in2test.lsi.uniovi.es/sqlmutation/, 2005

[11] S. S. Batth, E. R. Vieira, A. R. Cavalli, and M. U. Uyar, “Specification of Timed EFSM Fault Models in SDL,” in Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE’07), ser. LNCS, vol. 4574. Tallinn, Estonia: Springer, 26-29 June 2007, pp. 50–65.

[12] G. Fraser and F. Wotawa, “Mutant Minimization for Model-Checker Based Test-Case Generation,” in Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION’07), published with Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART’07). Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 161–168.

[13] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and W. E. Wong, “Mutation Testing Applied to Validate Specifications Based on Petri Nets,” in Proceedings of the IFIP TC6 8th International Conference on Formal Description Techniques VIII, vol. 43, 1995, pp. 329–337.

[14] S. C. Lee and A. J. Offutt, “Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis,” in Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE’01), Hong Kong, China, November 2001, pp. 200–209.

[15] N.Shabaldina, Khaled El-Fakih, N. , "Testing Nondeterministic Finite State Machines with Respect to the Separability Relation", TestCom/FATES, 2007, pp:305-318

[16] S. S. Batth, E. R. Vieira, A. R. Cavalli, and M. U. Uyar, “Specification of Timed EFSM Fault Models in SDL,” in Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE’07), ser. LNCS, vol. 4574. Tallinn, Estonia: Springer, 26-29 June 2007, pp. 50–65.

[17] P. Vilela, M. Machado, and W. E. Wong, “Testing for Security Vulnerabilities in Software,” in Software Engineering and Applications, 2002.

[18] PlexTest : [web-site]. URL: http://www.itregister.com.au/products/plextest, 2005

[19] Insure++: [web-site]. URL: http://www.parasoft.com/jsp/products/insure.jsp?itemId=63, 1998

[20] M. E. Delamaro, J. C. Maldonado, "Proteum/IM 2.0: An Integrated Mutation Testing Environment", Univ. de Sao Paulo, Sao Paulo, Brazil, 2001, pp. 91 - 101

[21] Cetress, “Certitude,”: [web-site]. URL: http://www.certess.com/product/ , 2006

[22] Y. Jia and M. Harman, “MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language,” in Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART’08). Windsor, UK: IEEE Computer Society, 29-31 August 2008, pp. 94–98.

[23] S. Kim, J. A. Clark, and J. A. McDermid, “The Rigorous Generation of Java Mutation Operators Using HAZOP,” in Proceedings of the 12th International Cofference Software and Systems Engineering and their Applications (ICSSEA 99), Paris, France, 29 November-1 December 1999.

[24] Jester : [web-site]. URL: http://jester.sourceforge.net/, 2001

[25] MuJava : [web-site]. URL: http://cs.gmu.edu/~offutt/mujava/, 2004

[26] B. H. Smith and L. Williams, “An Empirical Evaluation of the MuJava Mutation Operators,” in Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION’07), published with Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART’07). Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 193–202.

[27] Schuler, D., Dallmeier, V., and Zeller, A. 2009. Efficient mutation testing by checking invariant violations. In Proceedings of the 18th international Symposium on Software testing and Analysis (2009), pp. 69–80.

[28] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. Masiero, “Mutation Analysis Testing for Finite State Machines,” in Proceedings of the 5th International Symposium on Software Reliability Engineering, Monterey, California, 6-9 November 1994, pp. 220–229.

[29] R. M. Hierons and M. G. Merayo, “Mutation Testing from Probabilistic Finite State Machines,” in Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION’07), published with Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART’07). Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 141–150.

[30] А. В. Коломеец. Алгоритмы синтеза проверяющих тестов для управляющих систем на основе расширенных автоматов. Дис. … канд. техн. наук. Томск: 2010, 129 с.

[31] Ledyvania Franzotte and Silvia Regina Vergilio, “Applying Mutation Testing to XML Schemas”, Computer Science Department, Federal University of Parana (UFPR), Brazil, pp 6

[32] El-Fakih K., Prokopenko S., Yevtushenko N., Bochmann G. "Fault diagnosis in extended finite state machines",In Proc. of the IFIP, 15th Intern. Conf. on Testing of Communicating Systems, France -2003.