Deriving test suites for software checking based on Extended Finite State Machine

Morozov Maxim
Ivannikov Institute for System Programming of RAS, 25 Alexander Solzhenitsyn st., Moscow, 109004,
National Research University Higher School of Economics
Russian Federation

***Abstract*—This paper presents a comparative analysis of two formal methods, HSI method and Transition Tour method, for deriving test suites for software checking. The main idea, is in evaluating fault-detection capabilities of the test suites derived by these methods, by using mutant testing on software implementations of five EFSMs. The results based on these experiments data were analyzed and made it possible to identify some classes of mutations which are detected equally by HSI and Transition Tour test suites and one class of mutations that is detected better by HSI method.**

***Keywords*—** Extended Finite State Machine, Formal Methods, Harmonized State Identifiers Method, Transition Tour Method, Mutant Testing

I. Introduction

Formal methods offer a structured framework for validating software systems through mathematical modeling, analysis, and verification techniques. These methods help identify errors, defects, and vulnerabilities in the software development process. By employing formal methods, software developers and testers can detect potential issues and ensure compliance with specified requirements [1], [2]. Finite State Machines (FSMs) serve as foundational models for representing sequential logic systems in various software applications [3]–[6]. When testing FSMs, ensuring the comprehensive coverage of states, transitions and behaviors is essential to validate system correctness and reliability. Three popular testing techniques are based on various derivatives of the W-method, such as Wp-method, H-method, and HSI-method [6]. These methods focus on identifying various test scenarios and generating test cases that cover different paths through the software application.

When deriving test suites, the above methods include a tour for covering all transitions checking a final transition state by a set of distinguishing sequences. The HSI method involves the use of distinguishing sequences when checking the final state of each transition but differently from the W-method, these sequences are related only to the final transition state. On the other hand, the Transition Tour method focuses on designing test sequences that traverse all possible transitions in the FSM, aiming to detect errors that may arise from transition dependencies and boundary conditions.

In this paper, we present a comprehensive comparative study of the HSI method and Transition Tour method for deriving FSM based test suites for software implementations, focusing on their effectiveness in achieving coverage and fault detection. We incorporate mutant testing using two programming languages, C++ and Python, to assess the resilience and fault-detection capabilities of the testing techniques across different programming languages. By combining the analysis of the HSI method and Transition Tour method with mutant testing in multiple programming languages, our research aims to offer an examination of testing strategies for FSMs. Through this investigation, we seek to provide valuable insights into the strength and weakness of these methods in ensuring the reliability of a system under test that passes an FSM based test suite.

The rest of the paper is structured as follows. In Section II, the necessary definitions are briefly given. In Section III, the process of research is described and experimental data is provided. In Section IV, research questions are stated while in Section V, the obtained experimental results are presented, Section VI concludes the paper.

II. Background

A. Extended Finite State Machine

An Extended Finite State Machine (EFSM) is a formalism used in software and system engineering to model complex systems with multiple states and transitions. In essence, an EFSM extends the traditional concept of a FSM by incorporating additional features like data variables, actions, and guards to represent more intricate system behaviors. In an EFSM, each state represents a specific condition or mode of the system, while transitions signify the movement between states triggered by certain events or conditions.

B. FSM abstraction

An FSM abstraction for an EFSM is an FSM that has the similar behavior to the EFSM but without conditional transitions and data variables. For that purpose every FSM state represents one EFSM state with specified values for data variables of EFSM. The reason for deriving an FSM abstraction is the knowledge that test suites with guaranteed fault coverage can be derived for the FSM model, while for EFSMs algorithms are mostly based on some properties' covering are proposed.

C. HSI method

The HSI method is a sophisticated testing technique that utilizes search algorithms to generate diverse and effective test inputs for software systems. It is a derivative of the well-known W-method, but allows to generate shorter test sequences. To generate test cases, it uses inputs sequences that reach all transitions and the distinguishing sets. The original W method concatenates each input sequence with the whole W set. The HSI method selects a HSI set for each input sequence, resulting in less test cases. [7]

D. Transition tour method

The Transition Tour Method is a technique used to systematically explore the possible transitions between states in a system model. It involves the generation and examination of all possible transitions that can occur within a state machine. All W-base methods contain a transition tour of the specification FSM, i.e., such a test suite traverses every transition of the specification FSM. In paper [8], the authors study the fault coverage of a transition tour for microcontrollers and conclude that for such devices the fault coverage is rather high. However, we are not aware of the publications where the fault coverage of a transition tour of the FSM abstraction of an extended FSM specification is studied for software applications.

III. Experimental evaluation

A. Experimental setup

In this series of experiments, five EFSM were utilized to evaluate two test derivation techniques. These EFSMs, serving as the core entities for the experiments, were designed to represent various complex states and transitions within a system. Each EFSM was meticulously crafted to capture different aspects of state-based behavior and logic.

To utilize the testing process, all five EFSMs were presented in the forms of FSM diagrams, offering visual representations of the state transitions and behaviors encoded within the models. FSM forms were constructed manually due to the lack of automatic instruments for this purpose. That FSM descriptions can be found at the project's git repository [9] in tests folders. Used EFSMs will be considered in "Experimental data" in details.

To generate tests for the extended FSMs, the web platform fsmtestonline.ru [10] was used, which provided valuable tools and resources for generating tests based on the HSI and Transition Tour methods.

In addition to designing and testing the FSM models, the programs for the EFSMs were developed in two programming languages, C++ and Python. These programs were crafted to implement the behavior defined by the EFSM models, enabling the execution of mutant tests and evaluation of the testing coverage in a programming context.

To explore the testing process of the experiments, mutants were created for the developed programs in both C++ and Python. These mutants are derived by inserting various faults and errors into the program code, simulating potential defects that could arise in real-world software systems.

Finally, both the original programs and their corresponding mutants were subjected to rigorous testing using the tests generated by the HSI and Transition Tour methods. By executing these tests on the programs and mutants, the researchers could evaluate the effectiveness of the testing techniques in detecting faults, uncovering issues, and ensuring the reliability and robustness of the EFSMs implemented in C++ and Python. It is also interesting to evaluate both approaches with respect to classes of mutations to see which mutations can be detected with high fault coverage. Through meticulous testing and analysis, the experiments aimed to provide valuable insights into the strengths and limitations of different testing methodologies in the context of EFSMs.

B. Experimental data

The experimental data for this study consist of five EFSMs. Among these EFSMs, four were specifically designed to represent distinct protocols [2], each with its own set of states, transitions, and behaviors. The fifth EFSM describes the behavior of a calculator.

The first EFSM describes the behavior of the so-called "Simple Connection" protocol [11], that EFSM has three states, five inputs, five outputs and two context variables, it also has seven transitions.

The EFSM was transformed into an FSM with nine states, each corresponding to state and context variables' values. Also certain ranges of values were assigned to input and output parameters which results in nine inputs and nine outputs in the derived FSM. In the derived FSM 81 transitions were defined, 31 of them correspond to the original EFSM, other 50 describe undefined transitions in the original EFSM. Additional transitions were done by next algorithm: a transition is defined to the same state with a new output that is the same for all undefined transitions. The table that represents FSM transitions is available as Table 6 in Appendix.

The second EFSM describes the "Time" protocol. That EFSM has two states, two inputs, two outputs and no context variables, EFSM has two transitions. The EFSM was transformed into a FSM with the same number of states, inputs and outputs. In the derived FSM four transitions were defined, two of them correspond to the original EFSM, other two correspond to undefined transitions in the EFSM.

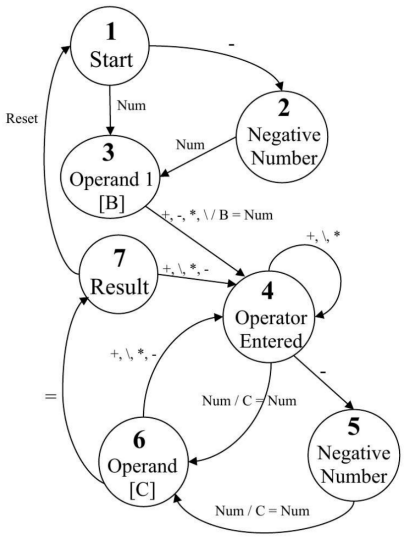

Fig. 1. A transitions diagram of the EFSM for a calculator

The third EFSM describes the "SMTP" protocol. That EFSM has two states, five inputs, seven outputs and one context variable, EFSM has eight transitions. The EFSM was transformed into a FSM with three states, corresponding to state and context variable value. Also certain ranges of values were assigned to inputs and outputs which results in seven inputs and seven outputs in the derived FSM. In the derived FSM 21 transitions were defined, 13 of them correspond to the original EFSM, other 8 correspond to undefined transitions in the EFSM.

The fourth EFSM describes the "POP3" protocol. That EFSM has four states, nine inputs, ten outputs and two context variables, EFSM has 16 transitions. The EFSM was transformed into an FSM with seven states, corresponding to state and context variables values. Also certain ranges of values were assigned to inputs and outputs which results in 14 inputs and ten outputs in the derived FSM. In the derived FSM 98 transitions were defined, 45 of them correspond to the original EFSM, other 53 describe undefined transitions in EFSM.

The fifth EFSM describes a simple calculator. That EFSM has seven states, seven inputs, six outputs and two context variables, EFSM has 11 transitions. The EFSM was transformed into a FSM with six states, seven inputs and seven outputs. In the derived FSM 42 transitions were defined, 20 of them correspond to the original EFSM, other 22 describe undefined transitions in the EFSM.

In our work, the following code mutations were considered.

- Changing IF conditions to constantly positive or negative expressions. (Group 1)
- Changing relationship signs: equal to not equal, less or greater and vice versa. (Group 2)
- Changing between each other: addition, subtraction, multiplication, division. (Group 3)
- Changing OR and AND operators between each other. (Group 4)
- Changing true to false and false to true. (Group 5)

A detailed analysis of various types of mutations is carried out in the following paper [12].

IV. Research questions

In the paper, three key research questions were identified to analyse the FSM based test derivation techniques applied to the corresponding software implementations.

1. Can a test suite built by HSI detect more errors than a transition tour? This fundamental question compares effectiveness of the HSI

method versus the Transition Tour method in terms of error detection within the FSMs. For hardware, the transition tour seems to be practical [8], [13], [14]. By analyzing the results of tests generated through these two techniques, our aim is to determine whether one method outperforms the other in detecting mutants' errors in associated programs.

2. Does the first property hold true for programs written in different languages? By examining whether the first property holds for programs written in different languages (C++ and Python), we can generalize and validity of this property across diverse programming languages. Programming languages are divided into different groups depending on typing: static or dynamic; and execution method: compiled or interpreted.

3. Does the first property hold true across mutations of different FSMs and different mutation classes? This research question focuses on the stability of the first property under various mutations applied to different FSMs.

By addressing these research questions, the experimental study aims to offer insights into the efficacy, reliability, and generalizability of test derivation techniques based on Finite State Machines, and study their applicability in real-world software testing scenarios.

V. Experimental results

For every EFSM, two programs were created: one in C++ language, another in Python language. C++ is a statically typed, compiled language, while Python is a dynamically typed, interpreted language. Next, for programs were generated a group of mutants; in C++ it was done via handmade framework, that substitutes terminal symbols in program for other terminal symbols, while in Python the mutatest [15] framework was used. Testing was performed for C++ in the handmade testing mode, for Python PyTest [16] was used.

TABLE I
Length of generated test suites

	Transition tour test suite length	HSI test suite length
Program 1	129	666
Program 2	4	8
Program 3	25	77
Program 4	126	1614
Program 5	60	278

TABLE II
Test results for C++ programs

	Total mutants	Detected mutants	Detection rate
Transition tour program 1	25	20	80%
HSI program 1	25	20	80%
Transition tour program 2	28	9	29%
HSI program 2	28	26	93%
Transition tour program 3	30	15	50%
HSI program 3	30	27	90%
Transition tour program 4	30	23	77%
HSI program 4	30	26	87%
Transition tour program 5	30	22	73%
HSI program 5	30	30	100%

TABLE III
Test results for Python programs

	Total mutants	Detected mutants	Detection rate
Transition tour program 1	24	21	88%
HSI program 1	24	21	88%
Transition tour program 2	53	39	74%
HSI program 2	53	39	74%
Transition tour program 3	52	34	65%
HSI program 3	52	37	71%
Transition tour program 4	73	68	93%
HSI program 4	73	68	93%
Transition tour program 5	61	29	48%
HSI program 5	61	45	74%

For every program developed in both C++ and Python, the HSI method and Transition Tour were applied to generate test sequences for detecting programs' mutants. Generated test suites are represented as a sequences of pairs. Every pair consists of an input symbol and expected output symbol from the FSM's transition. The length of the test suite is the total number of pairs of all sequences. Information about length of the generated test suites is provided in the Table I. The program number in Tables I, II, III corresponds to the number of the EFSM in Experimental data. It has been noted that test sequences generated using Transition tour method are usually significantly shorter — typically ranging from 2 to 12 times — than those generated by the HSI method. As mentioned above, the efficiency of Transition tour method is in producing high-quality test suites for hardware implementations [8], [13], [14].

Generated tests revealed that test suites generated by the HSI method consistently detected an equal or a higher number of mutants compared to those detected by the Transition Tour method. Moreover, every mutant that was detected by Transition Tour method was also detected by HSI method.

It has been noted that the weakest part of Transition tour method is the detecting of mutants where equal sign was changed to not equal, greater or less sign, these mutants refer to Group 1 mutants. At the same time, Transition tour is not worse than HSI in detecting other types of mutants. This can be explained by the fact that the HSI method creates more different states in the program, by having longer test suites, where a changed condition can lead to an unexpected result in the program. This result is the same for both languages C++ and Python.

Statistics for mutant detection for programs written in C++ language is available in Table 2, for Python programs is available in Table 3

TABLE IV
Language results comparison

	Total mutants	Detected mutants	Detection rate
Transition tour C++	143	89	62%
HSI C++	143	129	90%
Transition tour Python	263	191	73%
HSI Python	263	210	80%

TABLE V
TOTAL TEST RESULTS

	Total mutants	Detected mutants	Detection rate
Transition tour	406	280	69%
HSI	406	339	83%

Due to the final results, the HSI generated sequences which detect 83% of mutants for provided programs, when Transition Tour generated sequences detect only 69%. Statistics for final results is available in Table 5. These results are comparable with those in [17] where Java implementations were verified using test sequences generated by the W-method and its derivatives.

VI. CONCLUSION

The research of this study successfully answers all research questions, providing information about HSI method compared to the Transition Tour method in mutant detection.

1. The HSI method outperforms Transition tour method: The analysis conducted in this study demonstrates that the HSI method consistently identifies a higher number of errors compared to the Transition Tour method. In general, this was expected but for most mutant classes the Transition Tour did not lose.

2. Universal applicability across programming languages: Furthermore, the research findings reveal that the first property, holds true across all programs written in both languages. Despite their differing syntax and features, C++ and Python both yield similar results for the mutation testing. So the result seems to be applicable to both static languages like Java and Go, and dynamic languages like JavaScript.

3. Consistency across all kinds of FSM Mutations: Moreover, the first property holds true for all mutations across different EFSMs programs. The results indicate a consistent pattern, showing that the HSI method's efficiency in error detection remains reliable across the mutations of various FSMs compared to transition tour method. We also mention that for all mutations except of modifying the equal sign, the Transition Tour still provides the good fault coverage. We also mention that test suites derived by the Transition Tour method are shorter than those derived by the HSI method and the difference becomes more and more when the size of an EFSM increases.

In the future, that research can be explored across a wider array of programming languages to consider procedure, OOP and functional paradigms of programming. By increasing the number of tested EFSMs, research results can be refined. So, considering more systems can lead to confirmation or modification of the obtained results.

This study shows the consistent superiority of the HSI method over the Transition Tour method in error mutant detection. Irrespective of programming language or FSM mutations that shows powerful testing approach for software systems. Although it is worth considering that the length of the HSI test suites is many times greater, which can affect the performance of test frameworks and thus, for some mutation types the Transition Tour test suites possibly somehow augmented with distinguishing sequences can have the high-quality [8], [13], [14].

I would like to thank my supervisor Dr. Nina Yevtushenko for all her help and advice with this paper. Her immense knowledge and plentiful experience have encouraged me in all the time of my academic research.

REFERENCES

[1] Mathur, A. P. (2008) Foundations of Software Testing. Pearson Education, 689 p.

[2] Lai, R., (2002) A survey of communication protocol testing. The Journal of Systems and Software. 62:21–46.

[3] Gill, A. (1962) Introduction to the Theory of Finite-State Machines: McGraw-Hill.

[4] Lee D. and Yannakakis, M. (1996) Principles and methods of testing finite state machines- a survey. Proceedings of the IEEE, 84(8): 1090–1123.

[5] Petrenko, A. (1991) Checking experiments with protocol machines. Proc.4th Int. Workshop on Protocol Test Systems (IWPTS), 83–94.

[6] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, N. Yevtushenko, 2010 FSM-based conformance testing methods: a survey annotated with experimental evaluation, Information and Software Technology 52 (12)(2010) 1286–1297.

[7] Vanderson Hafemann Fragal, Adenilso Simao, Mohammad Reza Mousavi, Uraz Cengiz Turker (2018) Extending HSI Test Generation Method for Software Product Lines. The Computer Journal, Volume 62, Issue 1, pp. 109–129.

[8] Laputenko A. (2019) Logic Circuit Based Test Derivation for Microcontrollers. / A. Laputenko // The 20th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM 2019): Novosibirsk: NSTU publisher, 2019 – P. 70-73

[9] Morozov M. (2024) EFSM project GitHub. https://github.com/MaximMorozovMIPT/fsm-test/tree/main

[10] Yevtushenko N. (2024) Web tool for test suites generation. http://www.fsmtestonline.ru/

[11] Alcalde B., Cavalli A., Chen D., Khuu D., Lee D. (2004) Network Protocol System Passive Testing for Fault Management: A Backward Checking Approach. Proceedings of FORTE, pp. 150-166.

[12] A.V. Kolomeets. ALGORITHMS FOR SYNTHESIS OF CHECKING TESTS FOR CONTROL SYSTEMS BASED ON ADVANCED AUTOMATONS. Dissertation. for the Candidate of Science degree. tech. sciences in special 05.13.01, Tomsk State University, 2020, 108 p.

[13] Vinarskii E. (2018) Testing Digital Circuits: Studying the Increment of the Number of States and Estimating the Fault Coverage / E. Vinarskii, A.Laputenko, J. L´opez, N. Kushik // Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices: proceedings of 19th International Conference. Erlagol, Novosibirsk, p. 220-224.

[14] Lopez J. (2017) On the Fault Coverage of High-level Test Derivation Methods for Digital Circuits / J. L´opez, E. Vinarsky, A. Laputenko // 18th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM 2017) – Novosibirsk: NSTU publisher, P. 184-189.

[15] Evan Kepner (2024) Python mutation testing framework main page. https://pypi.org/project/mutatest/

[16] Holger Krekel (2024) Python unit tests framework main page. https://docs.pytest.org/

[17] K. El-Fakih, A. Alzaatreh, U. C. Turker, (2022) Assessing test suites of extended finite state machines against model- and code-based faults, Softw. Test. Verification Reliab. 32 (7)

APPENDIX

TABLE VI
FSM TRANSITIONS' FOR SIMPLE CONNECTION PROTOCOL EFSM

state/ input	S1(0,0,0)	S2(0,0,0)	S2(1,0,0)	S2(2,0,0)	S2(0,1,0)	S2(1,1,0)	S2(2,1,0)	S3(_,_,0)	S3(_,_,1)
ConReq qos = 0	S2(0,0,0)/ connect(0)								
ConReq qos = 1	S2(0,1,0)/ connect(1)								
ConReq qos = 2	S1(0,0,0)/ NonSupport(2)								
refuse		S2(1,0,0)/ connect(0)	S2(2,0,0)/ connect(0)	S1(0,0,0)/ CONcnf-	S2(1,1,0)/ connect(1)	S2(2,1,0)/ connect(1)	S1(0,0,0)/ CONcnf-		
accept qos = 0		S3(_,_,0)/ CONcnf(0)	S3(_,_,0)/ CONcnf(0)	S3(_,_,0)/ CONcnf(0)	S3(_,_,0)/ CONcnf(0)	S3(_,_,0)/ CONcnf(0)	S3(_,_,0)/ CONcnf(0)		
accept qos = 1		S3(_,_,0)/ CONcnf(0)	S3(_,_,0)/ CONcnf(0)	S3(_,_,0)/ CONcnf(0)	S3(_,_,1)/ CONcnf(1)	S3(_,_,1)/ CONcnf(1)	S3(_,_,1)/ CONcnf(1)		
accept qos = 2		S3(_,_,0)/ CONcnf(0)	S3(_,_,0)/ CONcnf(0)	S3(_,_,0)/ CONcnf(0)	S3(_,_,1)/ CONcnf(1)	S3(_,_,1)/ CONcnf(1)	S3(_,_,1)/ CONcnf(1)		
Data								S3(_,_,0)/ data(0)	S3(_,_,1)/ data(1)
Reset								S1(0,0,0)/ abort"	S1(0,0,0)/ abort