ISSN 0146-4116, Automatic Control and Computer Sciences, 2017, Vol. 51, No. 7, pp. 516–522. © Allerton Press, Inc., 2017.
Original Russian Text © A.D. Ermakov, N.V. Yevtushenko, 2016, published in Modelirovanie i Analiz Informatsionnykh Sistem, 2016, Vol. 23, No. 6, pp. 729–740.

Deriving Test Suites with the Guaranteed Fault Coverage for Extended Finite State Machines

A. D. Ermakov* and N. V. Yevtushenko**

Tomsk State University, Tomsk, 634050 Russia

**e-mail: antonermak@inbox.ru*

***e-mail: nyevtush@gmail.com*

Received September 2, 2016

Abstract—Extended finite state machines (EFSMs) are widely used when deriving tests for checking the functional requirements for software implementations. However, the fault coverage of EFSM-based tests covering appropriate paths, variables, etc., remains rather obscure. Furthermore, these tests are known be incapable of detecting many functional faults frequently occurring in EFSM-based implementations. In this paper, an approach is proposed for deriving complete tests with the help of a proper Java EFSM implementation. Since the software is based on a template, the faults turn directly into EFSM faults. The method proposed here makes it possible to derive test suites that can detect functional faults. In the first step, the EFSM-based test suite derived by a well-known method is checked for completeness with respect to the faults generated by the μJava tool. Then, each undetected fault is easily mapped into an EFSM mutant. In the next step, some FSM abstraction is used to derive a distinguishing sequence for two finite-state machines (if such a sequence exists), which is added to the current test suite. The test derived in this way is complete with respect to the faults generated by μJava. If the corresponding FSM derived by EFSM modeling is too complex or no such FSM can be derived, the resulting test suite can be incomplete. However, the experiments performed by us clearly show that the original test suite extended by distinguishing sequences can detect many functional faults in software implementations when the given EFSM is used as a specification for the system.

Keywords: mutation testing, extended finite state machine (EFSM), FSM abstraction, μJava

DOI: 10.3103/S0146411617070057

INTRODUCTION

Software implementations, including those used in critical systems, have become more sophisticated, and the completeness of their testing without the use of mathematical models is almost impossible. To derive tests with guaranteed fault coverage, researchers actively use models with a finite number of transitions [1, 2], for example, the model of an extended finite state machine (EFSM), which is sufficiently close to the software implementation in a high-level language. Despite the large number of publications on the automatic construction of this model, the examples described in these studies are usually limited to small programs, and often such EFSMs are built by test engineers manually from informal requirements for the software under test. Accordingly, it is almost impossible to compare the faults in the constructed model (the completeness of the test is to be guaranteed with respect to these faults) with the faults in the source program; therefore, to generate a test sequence detecting such a functional fault, one has to compare the specification software with the mutant program for constructing the corresponding distinguishing sequence [3, 4]. In this paper, we propose an EFSM-based method for deriving a set of test sequences that can detect many functional faults in the template implementation of the EFSM in Java.

An EFSM [5] extends the classical Finite State Machine (FSM) by internal (context) variables and input and output parameters. The EFSM-based tests are known to be sufficiently qualitative; however, there remain many functional faults in the tested software implementations that are not detected by the tests [4, 6]. Correspondingly, we propose to enhance the completeness of EFSM-based tests by considering the most common faults in software implementations on the basis of a "template" implementation of the EFSM in Java. Using the μJava tool [7], one can generate many mutants for a "template" software implementation; the initial test suite is applied to these mutants. The initial test can be derived by one of the existing methods, including a set of randomly generated sequences. If a mutant is not detected by the test, the corresponding fault is easily transformed into a fault in the EFSM, and thus the distinguishing sequence is constructed not for two software implementations which is known to be rather complicated [4] but for two EFSMs which is simpler [8]. The results obtained using this approach were partially published in [9]; in the current paper, we discuss the usability of the proposed approach for detecting functional faults in EFSMs.

The paper is organized as follows. Section 1 introduces the necessary definitions and notations. Section 2 discusses the techniques for deriving distinguishing sequences for two EFSMs (in our case, for an EFSM–specification and a mutant of interest). A proposed method for deriving the test is described in Section 3. Section 4 considers an example of testing the software implementation of the Simple Connection Protocol (SCP); we use this to illustrate the proposed approach. In conclusion, we briefly discuss possible directions for future research.

1. DEFINITIONS AND NOTATIONS

A *finite state machine* (FSM) is a quintuple $S = (S, I, O, T_S, s_0)$, where S is a nonempty finite set of *states* with the designated initial state s_0, I is a nonempty finite set of *inputs* (the input alphabet), O is a nonempty finite set of *outputs* (the output alphabet), and $T_S \subseteq I \times S \times S \times O$ is the transition relation [8]. An extended finite state machine (EFSM) extends an FSM with context (internal) variables, input and output parameters, and conditions that enable the transition. Formally [5], an EFSM M is the 6-tuple $M = (S, s_0, X, Y, T, V)$, where S is a nonempty finite set of states, X is a nonempty finite set of inputs, Y is a nonempty finite set of outputs, V is a finite (possibly empty) set of context variables, and T is the set of transitions between states of S. Each transition in an EFSM is a 7-tuple $(s, x, P, o_M, y, u_M, s')$, where s and s' are the initial and final transition states; $x \in X$ is an input and $D_{\text{inp}-x}$ denotes the set of input vectors the components of which are the values of the parameters that correspond to the input x (hereafter, *input parameters*); $y \in Y$ is an output and $D_{\text{out}-y}$ denotes the set of output vectors the components of which are the values of the parameters that correspond to the output y (hereafter, *output parameters*); and P, o_M, and u_M are functions defined over input parameters and context variables of V. The predicate P: $D_{\text{inp}-x} \times D_V \to \{0, 1\}$, where D_V is the set of context vectors (i.e., the vectors the components of which are the values of context variables), describes the conditions under which this transition can occur. The function o_M: $D_{\text{inp}-x} \times D_V \to D_{out-y}$ describes the values of the output parameters as the result of the transition. The function $u_M : D_{inp-x} \times D_V \to D_V$ describes the values of the context variables as the result of the transition.

Each pair (state, vector of values of context variables) is called a *configuration*, and the pairs (input, vector of values of input parameters) and (output, vector of values of output parameters) are called *parameterized* input and output, respectively. The initial configuration of an EFSM is usually assumed to be known. A transition in an EFSM can be executed if only the corresponding predicate takes the *true* value in this configuration for the given parameterized input. Thus, unlike classical systems with a finite number of states, not every transition in an EFSM can be executed in the current state (this is the well-known problem of the executability of a sequence of transitions in an EFSM). Quite possibly, the execution of a specific transition will require first the execution of a number of other transitions (for example, to reach a required value of the variable–counter) before the required transition can be executed.

By *functional faults* in an EFSM, we mean the faults of transitions/outputs, assignment of values to variables, errors in predicates, etc. The existing methods for deriving EFSM-based tests are focused on the coverage of paths, variables, conditions, etc. [4, 6]; however, S. Nica has showed in her thesis [4] that the completeness of these tests with respect to functional faults is very low (not exceeding 70%). Computer experiments with different protocol implementations [6] have revealed that the completeness of sufficiently long random tests with respect to functional faults is almost the same as the completeness of the tests constructed when covering the sets of different EFSM transitions.

As an example, we consider an EFSM describing the Simple Connection Protocol (SCP) [10, 11]. The EFSM (Fig. 1) has three states, which, generally speaking, describe different functioning modes. State S_1 describes the mode of waiting for a connection request, S_2 corresponds to the state of connection setup, and state S_3 corresponds to data transmission. The inputs describe the standard protocol commands: *Req* (request), *Conn* (connection), *Data* (data transmission), and *Reset* (reset). In addition, we use the input parameter *Support*, which equals 1 if the QoS level connection is ready and 0 otherwise. The input parameter *SysAvail* equals 1 if the system is free for connection and 0 if busy. The output parameters are *Nosupport*, *Error*, *Abort*, *Support*, *Refuse*, *Accept*, and *Ack*. The context variable *TryCount* corresponds to the counter of failed attempts to establish a connection. Although this is a somewhat "toy" protocol, it demonstrates rather well many aspects of protocol implementations.

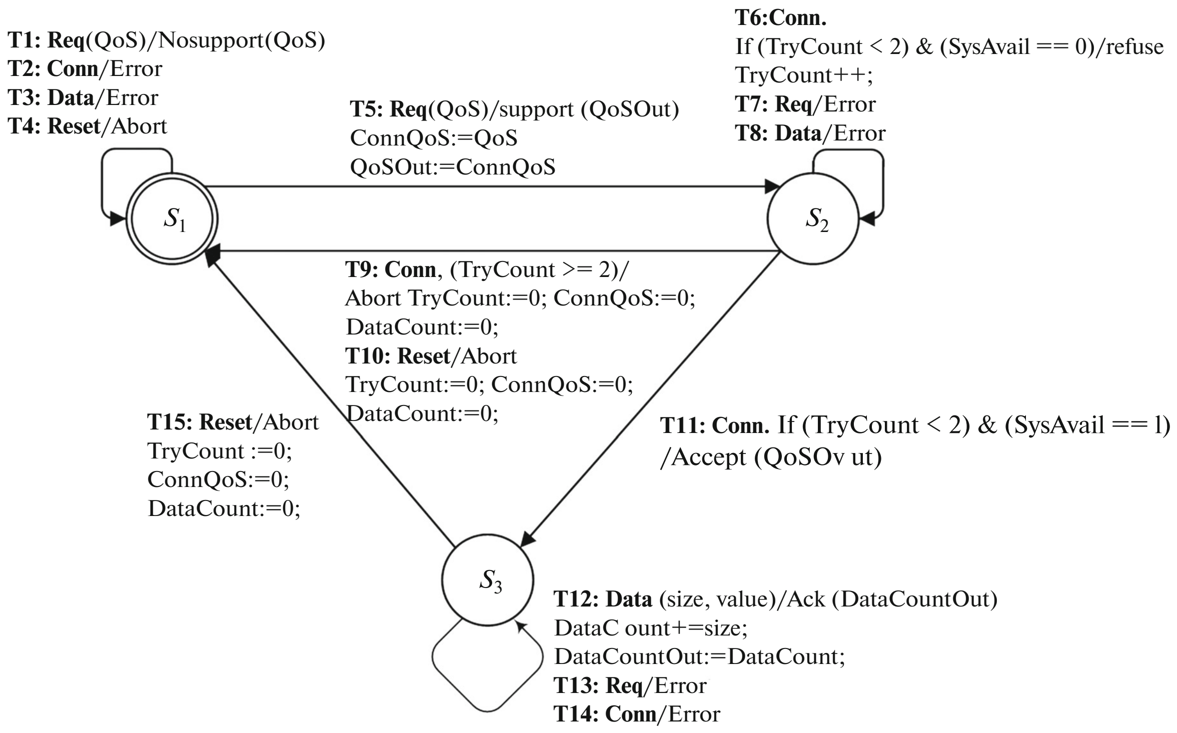

Fig. 1. EFSM for SCP.

It should be noted that, compared to distinguishing sequences for software mutants, the EFSM-based distinguishing sequences are derived in a much simpler way; in the next section, we briefly discuss the existing methods for deriving a distinguishing sequence for two EFSMs on the basis of different FSM abstractions, pointing out which abstractions are most suitable for which functional faults.

2. CONSTRUCTION OF DISTINGUISHING SEQUENCES FOR EFSMs

In this section, we discuss the methods for deriving a distinguishing sequence for two EFSMs. The general method described in [5], which guarantees the construction of a distinguishing sequence for two nonequivalent deterministic EFSMs, is rather cumbersome; moreover, the authors do not address the executability problem. All other methods developed for deriving distinguishing sequences for EFSMs are rather efficient heuristics based on FSM abstractions [11–15], for which executability is not an issue and a distinguishing sequence can be derived rather simply from the intersection of two FSMs (deterministic or nondeterministic, partial or complete).

Deriving a distinguishing sequence from the intersection of two EFSMs. The authors of [5] consider the problem of deriving a distinguishing sequence for two different EFSMs (in our case, a specification and a mutant). To describe all possible distinguishing sequences, a distinguishing FSM is constructed with a designated *fail* state that represents all sequences that can distinguish the initial configurations of an EFSM-specification and an EFSM-mutant undistinguishable with the specification by the test already derived. Since the distinguishability of two EFSMs is considered, where the sets of context variables do not intersect, the context variables of the mutant must be renamed. In this case, all distinguishing sequences are represented by the special state *fail*, and thus the derivation of a distinguishing sequence is reduced to the well-known problem of reachability. From the practical point of view, this distinguishing EFSM has some disadvantages. Firstly, only deterministic fully defined specifications are covered, which are not always available (especially, for protocol specifications). Secondly, not every sequence of transitions taking a distinguishing EFSM into the *fail* state is executable; i.e., the problem of executability is not addressed in the study. Thirdly, there remains the question of the equality of parameterized outputs when these symbols are even simple functions, rather than merely assigned to be equal to some constant. Thus, despite the generality of the proposed approach, the use of a distinguishing intersection–EFSM for deriving distinguishing sequences for two EFSMs is rather cumbersome.

Deriving a distinguishing sequence from FSM abstractions of two EFSMs. The best-known FSM abstractions for EFSMs are the following two. In the first case, the behavior of the EFSM in the initial configuration is simulated on input parameterized sequences until a given number of states is reached in the FSM or on all sequences of length of l or less. If the transition at which the FSM-specification and the FSM-mutant differ is known, the next transition can be chosen at each simulation step using some "greedy" algorithm, approaching as close as possible to the mutated transition. The experiments show that under the condition that the two EFSMs differ in a small number of transitions (one or two mutated transitions per specification), $l = 2$ or $l = 3$ will normally be sufficient to construct a distinguishing sequence. In addition, when the number of states is limited rather strongly, the traversal of the transition graph of the corresponding FSM reveals single errors in the predicates and functions for context variables and output parameters [13].

In the second case, all the predicates, input and output parameters, and corresponding functions are simply removed from the EFSM. In this case, the (adaptive) distinguishing sequence is constructed for two nondeterministic (possibly nonobservable) FSMs [14]. There are methods for deriving distinguishing sequences for these EFSMs [14, 15]; the experiments show that although the upper bounds on the length of these sequences are generally exponential, their length is mostly close to the number of FSM states. Further, the adaptive distinguishing sequences exist more often and are usually shorter; however, in this case, the tests should also be adaptive. Since all the predicates, context variables, and input and output parameters are removed, this abstraction can be effectively used only to derive distinguishing sequences for output and transition faults. For example, for the EFSM shown in Fig. 1, the error in the final state of transition T5 (state S_2 instead of S_1) is immediately detected with the input *Req*.

Deriving a distinguishing sequence from predicate abstractions of two EFSMs. Nondeterministic FSMs also appear for predicate abstractions of EFSMs [16]; for distinguishing sequences, these abstractions are proven to be most efficient when there are no input parameters.

Let β be the set of k predicates defined with respect to the context variables and input parameters of the EFSM M. If S is the set of states of the EFSM and D_W is the set of all configurations and parameterized input vectors, the predicate β-abstraction is defined as a_β: $SD_W \to S\{(0,1)\}^k$, where $a_\beta(s, \mathbf{w}) = (s, b_1, \ldots, b_k)$, $\mathbf{w} \in D_W$, $b_i \in \{0,1\}$, $i = 1,\ldots,k$. By definition, $b_i = 1$ if and only if the predicate $B_i(\mathbf{w})$ takes the "true" value. For a chosen set $\mathbf{P_x}$ of (parameterized) inputs, the set of inputs of the predicate abstraction contains pairs $(x, \mathbf{p_x})$, $\mathbf{p_x} \in \mathbf{P_x}$ and all nonparameterized inputs. For each configuration $(s, \mathbf{v})$, input $(x, \mathbf{p_x}) \in \mathbf{P_x}$, and predicate P of the transition (s, x, P, y, u_p, s') from state s, such that $(\mathbf{v}, \mathbf{p_x})$ turns P into truth and $u_M(\mathbf{v}) = \mathbf{v}'$, the set of transitions of the predicate abstraction involves the transition $((s, \mathbf{a}), x, y, (s', \mathbf{a}'))$, $(s, \mathbf{a}) = a_\beta(s, \mathbf{v})$ and $(s', \mathbf{a}') = a_\beta(s', \mathbf{v}')$. As an example, we consider the predicate abstraction for the EFSM shown in Fig. 1 with respect to the set of predicates (*TryCount* < 2) and (*SysAvail* = 0). If there is an error in the condition (*SysAvail* = 0), the predicate abstraction does not remain in state S_2 and goes from S_2 to state S_3, which can be easily detected with input *Data*.

In the general case, a predicate abstraction is a nondeterministic FSM where the degree of nondeterminism essentially depends on the chosen set of predicates and the set of parameterized inputs. Simplifying the proposition given in [16] for the case when the initial configurations of two EFSMs are distinguishable, we obtain that the two EFSMs are distinguishable by an (adaptive) input sequence α if the initial states of the corresponding predicate abstractions are separable by α (adaptively distinguishable); i.e., the sets of output responses to α in these states do not intersect (there exists an adaptive distinguishing test case [15]). Paper [16] discusses the choice of predicates for constructing a predicate abstraction in order to reduce the nondeterminism and increase the probability of the existence of a separating sequence. In addition, one can derive an adaptive distinguishing sequence; i.e., the two EFSMs can be adaptively distinguishable. As far as we know, there have been no studies of adaptive distinguishing sequences for EFSMs.

It should be noted that the construction of a predicate abstraction is rather cumbersome because the EFSM configurations are analyzed. However, if there is an error for only one transition (this is the case with μJava), one can simply mark the transition with the error in the predicate abstraction of the specification; i.e., there is no need to construct a new predicate abstraction for the FSM-mutant.

Deriving a distinguishing sequence from different slices of two EFSMs. A detailed discussion of this problem can be found in [13, 17]. The experiments conducted by the authors revealed that the tests constructed as transition tours of these slices can properly detect single functional faults, such as transition and output faults, as well as errors of assigning an incorrect value to a context variable/output parameter or the replacement of some arithmetic/logical operation.

3. METHOD FOR CONSTRUCTING A TEST USING THE μJAVA TOOL

The verification tests derived from an EFSM with the help of known (sufficiently simple) methods are supplemented with distinguishing sequences for the corresponding mutants, which are constructed using the special tool μJava. This tool has wide functionality and, according to the documentation, can generate 34 types of software code mutations, primarily the traditional faults (such as replacement of mathematical, logical, and comparison operators) and faults of object-oriented programming (such as inheritance and polymorphism faults), which fit sufficiently well with the software functional faults. The latter dictates the choice of this approach for increasing the completeness of tests derived from an EFSM. Single faults are known to be the most difficult in terms of detection; therefore, exactly these mutants of software implementations are considered. The mutations can be generated by running the μJava GUI and selecting the program project and the types of mutations to be generated. As a result, the directory *Results* includes all the mutants generated in subdirectories with their names corresponding to the mutation type and number. Using the unit-testing library JUnit [18], the test can be applied to all mutants simultaneously to determine which mutants cannot be detected by the test. Thus, the EFSM-based test suite with μJava consists of the following steps.

Step 1. The initial verification test suite *TS* is derived by one of the well-known methods on the basis of EFSM-specification *M*. One can use transition tour of the corresponding EFSM, a number of methods for covering the paths, variables, conditions, etc., as well as randomly generated tests of a certain length.

Step 2. Using a predefined template, we construct a software implementation of the EFSM-specification in such a way that the software implementation errors can be strictly linked to EFSM faults. Specifically, in the software implementation, the EFSM states are used as labels for describing the corresponding operation mode. The context variables and input and output parameters correspond to those in the software implementation. The predicates describe the conditions for the execution of instructions.

Step 3. The test suite *TS* is checked for completeness for the template software implementation using the errors introduced by the μJava generator. The undetected errors are introduced into the EFSM and the set *Mut* of EFSM-mutants undetected by *TS* is constructed.

Step 4. For each EFSM *Imp* from the set *Mut*, an appropriate FSM abstraction is constructed and a sequence is determined that distinguishes the FSM abstractions of the two EFSMs, the EFSM-specification, and the EFSM *Imp*. If it exists, this sequence is added to the *TS*; if there is no such sequence, the current mutant and the specification are concluded to be indistinguishable.

Proposition 1. If there exist deterministic FSMs modeling the behavior of the EFSM-specification and each constructed mutant, the algorithm consisting of the above-described steps returns a complete test with respect to quasi-equivalence; i.e., the constructed test suite detects any mutant with its behavior differing from the specification in some (parameterized) input sequence defined in the specification.

In some cases, for an EFSM-specification or a mutant, it is impossible to derive an FSM of this kind due to computational problems or, for example, an infinite set of admissible values for a context variable or input parameter. Accordingly, we cannot guarantee the distinguishability of a mutant and a specification if no distinguishing sequence is found; however, the experiments conducted with a number of EFSMs describing the behavior of protocols and technical systems revealed that these cases are sufficiently rare. If the resulting FSMs are nondeterministic, the completeness of the derived test is determined with respect to nonseparability or adaptive nondistinguishability (when an adaptive distinguishing sequence is used), rather than with respect to equivalence. It should be emphasized that for arbitrary EFSMs there are no necessary and sufficient conditions for checking the equivalence, reduction, separability, and adaptive distinguishability of two FSMs. In our experiments, we considered rather simple EFSMs; therefore, we used only sufficient conditions to check these relations. One of these conditions is the injection of an error leading to an additional connectivity component in the reference EFSM; in this case, no changes occur in the original connectivity component.

4. ANALYSIS OF THE EXPERIMENTS FOR THE CASE OF SCP

The experiments were conducted with EFSMs describing the protocols such as SCP, "Time," SMTP, POP3, TFTP, calculator, and Audio CD player. In almost all cases, except for the simplest protocols, we had to supplement the initial transition tour of the EFSM by distinguishing sequences. We illustrate the process in more detail for the simple connection protocol (SCP). For this protocol, we developed a software implementation of the EFSM and used the transition tour as the initial test suite. The μJava tool allowed us to generate 245 traditional (arithmetic) mutants and 7 object-type mutants (see the Table 1).

Table 1. Generated mutants

Name	Mutant description	Number of mutants
AOIS	Increment/decrement of random variable	96
AOIU	Negation of variable	5
LOI	Bitwise negation	24
ROR	Rational operator replacement >, <, =, <=, >=, ==	91
COR	Conditional operator replacement ^, ‖, &&, &, |&	4
COI	Conditional operator insertion !(true), !(false)	17
ASRS	Assignment operator replacement (short-cut) +=, /=, -=, %=	8
JSI	Insert static modifier	7
Total		252

The initial test suite *TS* run on these mutants showed that 62 mutants (24.6%) were identical to the original program. The use of FSM abstractions revealed that 9 (3.6%) of them are not equivalent to the specification. The remaining 53 (21%) mutants introduced no changes to the behavior of the EFSM-specification. Further, the return to the EFSM made it possible to identify the transitions where nonequivalent mutations had occurred. Due to this, the test suite was supplemented by adding three distinguishing parameterized input sequences of a total length of 11. Thus, the test length increased from 18 to 29 inputs. The test rerun distinguished 9 nonequivalent mutants from the specification. Thus, only 53 equivalent mutants remained undistinguished. As a result, the test completeness increased from 75.4 to 100% (with respect to the mutants generated by μJava).

5. CONCLUSIONS

In this study, we have proposed a method for increasing the completeness of EFSM-based tests through the use of mutation testing for the software implementation based on a special template. The tests constructed as a transition tour of the specification have proven to be incomplete with respect to the faults introduced by μJava; i.e., there were functional faults undetected by the test. This is primarily caused by the fact that the EFSM-specification is derived most often from some informally described requirements for the software implementation, rather than directly from the software implementation. To fully detect the faults introduced by μJava, the test suite was supplemented by different sequences for FSM abstractions of the specification and its mutant. The experiments show that this approach on the basis of FSM abstractions has made it possible to identify the mutants that were equivalent to the specifications, as well as to complete the verification test with distinguishing sequences for the nonequivalent mutants. It is of interest to analyze whether it is possible to use observability for context variables on the basis of this approach. If some mutations of the context variables are not detected with FSM abstractions, it is reasonable to make these variables observable during debugging, i.e., to declare them as output parameters. The construction of distinguishing sequences makes it possible to minimize the list of these observable variables. It is of special interest to derive distinguishing sequences for FSMs of special classes, such as tree and timed FSMs. Another direction of our future activities is the use of the results obtained in this study not only to detect, but also to localize the faults in software implementations.

ACKNOWLEDGMENTS

This work was supported by the Russian Science Foundation (RSF), project no. 16-49-03012.

REFERENCES

1. Kaur, M. and Singh, R., A review of software testing techniques, *Int. J. Electron. Electr. Eng.,* 2014, vol. 7, no. 5, pp. 463–474.
2. Jorgensen, P.C., *Software Testing: A Craftsman's Approach,* Auerbach Publications, 2008, 3rd ed.
3. Nica, M., Nica, S., and Wotawa, F., On the use of mutations and testing for debugging, *Software Pract. Exp.,* 2013, vol. 43, no. 9, pp. 1121–1142.
4. Nica, S., On the use of constraints in program mutations and its applicability to testing, *PhD Thesis,* Graz Technical University, 2013.
5. Petrenko, A., Boroday, S., and Groz, R., Confirming configurations in EFSM testing, *IEEE Trans. Software Eng.,* 2004, vol. 30, no. 1, pp. 29–42.
6. El-Fakih, K., Salameh, T., and Yevtushenko, N., On code coverage of extended FSM based test suites: An initial assessment, *Lect. Notes Comput. Sci.,* 2014, vol. 8763, pp. 198–204.
7. µJava Documentation. µJava home page, 2014. http://cs.gmu.edu/~offutt/mujava/. Accessed April 10, 2016.
8. Villa, T., et al., *The Unknown Component Problem: Theory and Applications,* Springer, 2012.
9. Ermakov, A. and Yevtushenko, N., Increasing the fault coverage of tests derived against Extended Finite State Machines, *Syst. Inf.,* 2016, vol. 7, 23–32.
10. Alcalde, B., et al., Network protocol system passive testing for fault management: A backward checking approach, *Proc. of the 24th IFIP WG 6.1 Intern. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'2004,* pp. 150–166.
11. Kushik, N., et al., Optimizing protocol passive testing through "Gedanken" experiments with finite state machines, *Proc. Int. Conf. Software Quality and Reliability, QSR'2016,* 2016.
12. Kushik, N. and Yenigün, H., Heuristics for deriving adaptive homing and distinguishing sequences for nondeterministic finite state machines, *Lect. Notes Comput. Sci.,* 2015, vol. 9447, pp. 243–248.
13. Kolomeets, A.V., Algorithms for the synthesis of checking tests for control systems based on extended automata, *Cand. Sci. (Eng.) Dissertation,* 2010.
14. Kushik, N., Yevtushenko, N., and Cavalli, A., On testing against partial non-observable specifications, *Proc. Int. Conf. Quality of Information and Communications Technology,* 2014, pp. 230–233.
15. Kushik, N., et al., On adaptive experiments for nondeterministic finite state machines, *Int. J. Software Tools Technol. Transfer,* 2016, vol. 18, no. 3, pp. 251–264.
16. El-Fakih, K., et al., Distinguishing extended finite state machine configurations using predicate abstractions, *J. Software Res. Dev.,* 2016.
17. Mikhailov, Yu.V. and Kolomeets, A.V., Checking the transitions in an extended automaton based on slices, *Vestn. Tomsk. Gos. Univ., Ser. Upr. Vychisl. Tekh. Inf.,* 2008, vol. 3, no. 4.
18. JUnit 4, documentation. http://cs.gmu.edu/~offutt/mujava/. Accessed April 10, 2016.

Translated by V. Arutyunyan