UDC 004.415.53 - 519.7

Increasing the fault coverage of tests derived against Extended Finite State Machines

Anton Ermakov (Faculty of Radiophysics, Tomsk State University)
Nina Yevtushenko(Faculty of Radiophysics , Tomsk State University)

Abstract. Extended Finite State Machines (EFSMs) are widely used when deriving tests for checking whether a software implementation meets functional requirements. These tests usually are derived keeping in mind appropriate test purposes such as covering paths, variables, etc. of the specification EFSM. However, it is well known that such tests do not detect many functional faults in an EFSM implementation. In this paper, we propose an approach for increasing the fault coverage of test suites initially derived against the specification EFSM. For this reason, the behavior of the specification EFSM is implemented in Java using a template that is very close to the EFSM description. At the next step, the fault coverage of an initial test suite derived against the specification EFSM is calculated with respect to faults generated by μJava tool. Since the EFSM software implementation is template based, each undetected fault can be easily mapped into a mutant EFSM of the specification machine. Thus, a distinguishing sequence can be derived not for two programs that is very complex but for two machines and there are efficient methods for deriving such a distinguishing sequence for Finite State Machine (FSM) abstractions of EFSMs. As an FSM abstraction, an *l*-equivalent of an EFSM can be considered that in fact, is a subtree of the successor tree of height *l* that describes the EFSM behavior under input sequences of length up to *l*. Such *l*-equivalents are classical FSMs and if *l* is not large then a distinguishing sequence can be derived simply enough. The initial test suite augmented with such distinguishing sequences detects much more functional faults in software implementations of a system described by the specification EFSM.

Key words: *Extended Finite State Machine (EFSM), test derivation, fault coverage, mutation testing, μJava.*

1. Introduction

Model based test derivation is now widely used for deriving functional (conformance) tests for software implementations [1, 2] which nowadays are used everywhere including various critical systems. When deriving tests with the guaranteed fault coverage finite state models such as Finite State Machines (FSMs) and their extensions are widely used [2], since these models have the

natural reactivity and there are no races between inputs and outputs. However, traditional FSMs are too big for real-life software and extracting such a model from informal description of functional software requirements is rather difficult. Despite the big number of publications about automatic derivation of an FSM from informal behavioral restrictions, most authors consider small examples and very often such model is manually derived by a test engineer. The Extended Finite State Machine (EFSM) [3] extends the classical FSM with input and output parameters, context variables, update functions and predicates defined over context variables and input parameters. For test derivation for telecommunication protocol implementations the EFSM model is often extracted from the protocol RFC specification [2]. As a result, it is nearly to impossible to find the correlation between model and software faults with respect which we are going to guarantee the fault coverage. Often enough faults are injected into constructed software and then corresponding mutants are distinguished [4, 5]. In this paper, we propose to derive distinguishing sequences not for code but for two EFSMs injecting faults into the template Java implementation of the specification EFSM. We note that the same approach can be applied when using other programming languages for which an automatic mutation tool exists.

There exist a number of EFSM based test derivation methods. Tests usually are derived keeping in mind appropriate test purposes such as covering paths, variables, etc. of the specification EFSM. The derived tests have a good quality but it is well known [6, 7] that such tests do not detect many functional faults in a software implementation of a system described by the EFSM. Correspondingly we propose to increase the fault coverage of EFSM based test suites constructing a Java template implementation of the specification EFSM. Using the tool μJava [8] a number of mutants is generated for the template EFSM implementation which are tested using the initial test suite derived by covering appropriate paths in the specification EFSM. If a mutant is not detected by the initial test suite then a corresponding fault is easily mapped into an EFSM fault and a distinguishing sequence is derived not for two software programs that is known to be a very complex task [5] but for two finite state models that is known to be much simpler [9, 10]. First results have been published in [11]; in this paper, we extend a proposed approach to arbitrary EFSMs.

The rest of the paper is structured as follows. Section 2 contains preliminaries. A proposed approach is described in Section 3. In Section 4, we apply a proposed approach to a Simple Connection Protocol SCP that being a ‘toy example’ has many features that are presented in real protocol descriptions. Section 5 concludes the paper.

1. Preliminaries

A *Finite State Machine* (*FSM*) [12] is a 5-tuple $S = (S, I, O, h_S, s_0)$, where S is a nonempty finite set of states with the designated initial state s_0, I and O are nonempty finite *input* and *output* alphabets, $h_S \subseteq I \times S \times S \times O$ is a *behavior* or a *transition* relation. Extended FSM (EFSM) extends the classical FSM by *context* (internal) variables, *input* and output parameters and conditions when a transition can be fired. Formally [3], an EFSM M is a 5-tuple $M = (S, s_0, X, Y, T, V)$, where S is a nonempty finite set of states of the EFSM, X and Y are nonempty finite *input* and *output* alphabets, V is a finite possibly empty set of context variables, T is a set of transitions between states of S. Every transition of the EFSM is a 7-tuple $(s, x, P, o_M, y, u_M, s')$, where s and s' are the initial and final states of the transition; $x \in X$ is an input, along with $D_{inp\text{-}x}$ denoting the set of input vectors, i.e., vectors with all possible values of input parameters which correspond to x (*input parameters*); $y \in Y$ is an output, along with $D_{out\text{-}y}$ denoting the set of output vectors, i.e., vectors with all possible values of output parameters which correspond to y (*output parameters*); P, o_M and u_M are functions over input parameters and context variables from V. The predicate P: $D_{inp\text{-}x} \times D_V \rightarrow \{0, 1\}$ where D_V is the set of context vectors, describes the conditions when a corresponding transition can be fired; the function $o_M : D_{inp\text{-}x} \times D_V \rightarrow D_{out\text{-}y}$ updates the values of output parameters after firing the transition, while the function $u_M : D_{inp\text{-}x} \times D_V \rightarrow D_V$ updates the context variables.

The *configuration* is a pair «state, context vector»; a *parameterized input* (*parameterized output*) is a pair «input, vector of input parameter values» («output, vector of output parameter values». The *initial* configuration is usually denoted $(s_0, \mathbf{v_0})$. A transition of an EFSM can be *fired* if the corresponding predicate is 'True' for the current parameterized input and configuration. Thus, differently from classical FSMs not each transition at a current state can be fired and this is the well known problem of transition execution [3]. It is possible that in order to fire a given transition we have to execute a number of other transitions first, for example, in order to reach a predefined value of a counter.

Well known test for an EFSM is a transition tour which is widely used for detecting functional faults in various systems' implementations [7]. A *transition tour* is a parameterized input sequence that traverses each transition of the EFSM. As mentioned above, it is not simple to construct such a sequence for an EFSM; however, there exist methods [13] for the transition tour construction. Other methods construct the set of input sequences that cover critical paths, conditions, variables but as it is shown in PhD thesis of S. Nika [6], the fault coverage of such tests with respect to functional software faults is very low, around 70 %; when considering functional faults such as transition

and/or predicate faults, variable and/or parameters updating, etc. On the other hand, in [7], it is shown that the transition tour also is not very efficient with respect to such functional faults, since a transition tour covers only appropriate paths. Accordingly in [7] it is shown that the fault coverage of random tests of appropriate length is almost the same as that of a transition tour.

As an example, we consider an EFSM describing the behavior of Simple Connection Protocol (SCP) [14, 15], that has three states; every state describes an appropriate operating mode. State S_1 describes a mode when a protocol implementation is waiting for connection, S_2 corresponds to establishing the connection while state S_3 is related to data transmission. Inputs correspond to standard protocol commands: *Req* (request), *Conn* (connection), *Data* (data transmission) and *Reset*. We also use an input parameter *Support* that equals 1 when the connection of the predefined quality QoS can be established and 0, otherwise. Input parameter *SysAvail* equals 1 if the system is available and 0, otherwise. Output parameters are also very natural: *No support*, *Error*, *Abort*, *Support*, *Refuse*, *Accept*, *Ack_1*. Context variable *TryCount* corresponds to the number of attempts when the connection has failed. Despite the fact that this protocol is somehow a “toy protocol”, it illustrates many aspects of protocol implementations.

As mentioned above, differently from deriving a distinguishing sequence against software mutants the derivation of such sequences for finite state machines is much simpler. Distinguishing sequences for two FSMs are constructed based on the product of these machines [11]; for EFSMs it is a bit more complex, since appropriate FSM abstractions are derived first [16, 17, 18]. Such abstractions can be derived in various ways, for example, we can simply delete all predicates, context variables, input and output parameters and updating functions. As shown in [19], in this case, a distinguishing sequence will be constructed for two nondeterministic FSMs. It is also the case when predicate abstractions are considered when deriving a distinguishing sequence [20]. One of simple ways is to use *l*-equivalents of an EFSM which describe the EFSM behavior under critical (parameterized) input sequences of length up to *l*. In the paper [10], it is experimentally shown that when two EFSMs differ in a small number of transitions (the specification EFSM and a mutant EFSM with one or two mutation transitions) usually it is enough to consider $l = 2, 3$ when deriving a distinguishing sequence.

2. Test derivation when using μJava

An initial test suite is derived against the specification EFSM using one of known approaches. It can be a transition tour or a set of randomly derived test cases of appropriate length. This initial test suite will be then augmented with distinguishing sequences for mutants derived by μJava for a

template Java implementation of the specification EFSM. The last version of this tool (μJava, v.4) appeared in June, 2013 [8]. The μJava has good functional abilities and according to the documents can generate 34 types of code mutations. There are traditional faults such as the operator or variable replacement and object-oriented faults for inheritance, polymorphism, etc.; there is the high correlation between these mutants and functional software faults. For this reason, we have selected this approach in order to increase the fault coverage of EFSM based test suites. It is known that single faults are most hard detecting faults [6], while test suites complete with respect to single faults detect a big number of other (multi) faults. Correspondingly, μJava injects exactly various types of single faults into software implementations. For correct use of μJava it is necessary to perform appropriate pre-settings which can be found in the official developer site [6]. For mutant generation, the μJava graphic shell should be installed and the code project and mutation types have to be selected. As a result, in the *Results* folder all the generated mutants will appear; subfolders will have titles corresponded to a mutation type. By the use of the JUnit library for module testing [21] a current test suite can be applied to all mutants simultaneously in order to determine mutants which are not detected by the test suite, i.e., have the behavior that cannot be distinguished with the template Java implementation. For those mutants, corresponding faults will be injected into the specification EFSM and a distinguishing sequence will be derived for two machines, a mutant and the specification. Therefore, EFSM based test derivation strengthened with μJava includes the following steps.

Step 1. An initial EFSM based test suite *TS* is derived using one of well known methods. This test suite can be a transition tour of the specification EFSM *M*, or a test suite can cover some critical transitions, conditions, paths, etc., or a test suite can be a random test of appropriate length.

Step 2. A Java template implementation of the specification EFSM is derived. The template is very close to the EFSM notion and thus, there is the strong correlation between faults in the specification EFSM and template implementation. In particular, EFSM states in the template implementation are the values of a corresponding variable (for describing an appropriate mode, for example). Context variables and input and output variables correspond to those in the template implementation; predicates describe the conditions for instruction execution.

Step 3. The fault coverage of the test suite *TS* is checked with respect to faults injected by μJava generator into the template implementation and the set *Mut* of the specification EFSM mutants corresponded to undetected faults is constructed.

Step 4. For each EFSM *Imp* of the set *Mut*, an appropriate FSM abstraction is derived keeping in mind mutated transitions. At the next step, a distinguishing sequence for the specification and mutant FSM abstractions is constructed if such a sequence exists. In this case, a derived

distinguishing sequence is added to *TS*. If such a sequence is not found then the conclusion is drawn that the mutant is *indistinguishable* with the specification EFSM.

Remark. We note that when a distinguishing sequence is not found, we cannot guarantee the equivalence of the mutant and specification but our experiments with protocol implementation show that such situations occur very rare. In other words, according to performed experiments, the specification and indistinguishable mutant EFSMs always were equivalent. Nevertheless, we underline that for EFSMs there are no necessary and sufficient conditions of the two EFSMs' equivalence and in our experiments when checking the equivalence, we used only some sufficient conditions. One easily checked condition is that the fault injection creates instructions which do not influence the specification EFSM behavior.

3. Analyzing performed experiments for simple connection protocol

We performed experiments with EFSMs which are used for describing protocols Simple Connection Protocol, Time, SMTP, POP3, TFTP, Audio CD player [11]. Almost in all cases, save for the simplest protocols, a transition tour needed to be augmented with distinguishing sequences obtained after using the μJava tool. The augmentation process in more details is illustrated for the Simple Connection Protocol. A template Java implementation has been obtained for this protocol and a transition tour was used as an initial test suite. After applying μJava, 245 traditional (arithmetic) mutants have been generated, along with seven object oriented mutants (Table 1).

Table 1. Generated mutants

Name	Mutant description	Number of mutants
AOIS	Variable increment/decrement	96
AOIU	Inserting a unary operator (arithmetic "-") before a variable	5
LOI	Operand bit based inversion	24
ROR	Logic operator replacement >,<,=,<=,>=,==	91
COR	Logic operand replacement ^,||,&&,&,|	4
COI	Injecting logic inversion into conditions	17
ASRS	Arithmetic operator modification: +=, /=, -=, %=	8
JSI	Adding the "*Static*" modifier to instance variables	7
Overall		**252 mutants**

When applying the initial test *TS* to all generated mutants, it occurred that 62 mutants (24,6%) produced expected outputs to all test cases. Using FSM abstractions we determined that 9 (3,6%) are distinguishable with the specification EFSM. Other 53 (21%) mutants were indistinguishable with the EFSM specification. Based on the EFSM specification the paths were determined where nonequivalent mutations occurred and a test suite has been augmented with three additional parameterized distinguishing sequences of the total length 11; correspondingly, the test suite length was increased from 18 up to 29 parameterized inputs. We also checked 53 indistinguishable mutants using simple sufficient conditions and all of them were found to have injected faults which do not influence the specification behavior. Therefore, the fault coverage of the initial test suite has been increased from 75,4% up to 100 % (with respect to mutants generated by μJava tool).

4. Conclusion

In this paper, we proposed how to increase the fault coverage of EFSM based test suites when testing software implementations, since tests based on covering appropriate paths, variables, etc. are known to be incomplete with respect to functional software faults. We develop a template Java implementation of the EFSM specification such that implementation faults can be easily mapped into the EFSM faults. The μJava tool is used to inject faults into the template implementation and the set of corresponded EFSM faults undetected with the initial test suite is constructed. Thus, a distinguishing sequence is derived not for two Java programs that is very complex but for two machines and there are efficient methods for deriving such a distinguishing sequence for FSM abstractions of EFSMs. As the performed experiments show this approach allows to eliminate mutants which are equivalent to the EFSM specification and to augment the initial test suite with appropriate distinguishing sequences for non-equivalent mutants. We plan more experiments with real protocol software implementations in order to reveal which functional faults still are not detected with constructed test suites. Another direction of our future work includes the study how to use the obtained results for fault localization in software implementations.

References

1. Kaner C., Falk J., Nguyen H.Q. Testing Computer Software, 2nd Ed. New York, et al: John Wiley and Sons, Inc, 1999. 480 p.
2. Proceedings of the International Conference on testing software and systems (former workshop on protocol testing). Kluwer, Springer, 1991 – 2015.

3. Petrenko A., Boroday S., Groz R. Confirming Configurations in EFSM Testing. IEEE Trans. Software Eng. 30(1), 2004.
4. Nica M., Nica S., Wotawa F. On the use of mutations and testing for debugging. Software: practice & experience. 43(9), 2013. 1121-1142.
5. Nica S., Wotawa F. Using Constraints for Equivalent Mutant Detection. Proceedings of WS-FMDS, 2012. pp. 1-8.
6. Nica S. On the Use of Constraints in Program Mutations and its Applicability to Testing, PhD thesis, Graz Technical University, 2013.
7. El-Fakih K., Salameh T., Yevtushenko N. On Code Coverage of Extended FSM Based Test Suites: An Initial Assessment. Lecture Notes in Computer Science, LNCS 8763, 2014. pp. 198-204.
8. μJava documentation // μJava Home Page. 2014. [WEB]. URL: http://cs.gmu.edu/~offutt/mujava/ (accessed: 10.04.2016).
9. Gill A. Introduction to automata theory, N.Y., 1964.
10. Kushik N., Forostyanova M., Prokopenko S., Yevtushenko N. Studying the optimal height of the EFSM equivalent for testing telecommunication protocols. Proceedings of CCIT, 2014.
11. Ermakov A. Deriving Conformance Tests for Telecommunication Protocols Using μJava Tool // 15th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM 2014). Novosibirsk: Nobosibirsk State Technical University, 2014. pp. 150-153.
12. Villa T., Yevtushenko N., Brayton R., Mishchenko A., Petrenko A., Sangiovanni-Vincentelli. The Unknown Component Problem. Springer, 2012.
13. Burdonov I.B., Kossachev A.S. Studying transition graph by interacting automata. Vestnik of Tomsk State University: series on control systems and informatics. 2014, № 3 (28), pp. 67-75 (in Russian).
14. Alcalde B., Cavalli A., Chen D., Khuu D., Lee D. Network Protocol System Passive Testing for Fault Management: A Backward Checking Approach. Proceedings of FORTE, 2004, pp. 150-166.
15. Kushik N., Lopez J., Cavalli A., Yevtushenko N. Optimizing Protocol Passive Testing through ‘Gedanken’ Experiments with Finite State Machines. Proccedings of QSR, 2016, (accepted for publication).
16. Kolomeez A.V. Algorithms for Extended Finite State Machine based test derivation for control systems. PhD thesis, Tomsk State University, 2010 (in Russian).
17. Kushik N., Kolomeez A., Cavalli A., Yevtushenko N. Extended Finite State Machine based Test Derivation Strategies for Telecommunication Protocols. Proceedings of SYRCOSE, 2014. pp. 108-113.
18. Kushik N., Yenigün H., Heuristics for Deriving Adaptive Homing and Distinguishing Sequences for Nondeterministic Finite State Machines. Lecture Notes in Computer Science, LNCS 9447, 2015. pp. 243-248.
19. Kushik N., Yevtushenko N., Cavalli A. On Testing against Partial Non-observable Specifications. Proceedings of QUATIC, 2014. pp. 230-233.

20. El-Fakih K., Yevtushenko N., Bozga M., Bensalem S. Distinguishing extended finite state machine configurations using predicate abstractions. Journal on Software Research and Development, published online March, 31, 2016.
21. JUnit 4, documentation. [web] // JUnit Home Page. – URL: http://junit.org/junit4/ (accessed: 10.04.2016).