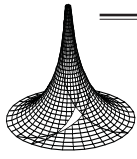


18/1 (2020), 1–17
DOI: 10.5485/TMCS.2020.0469
tmcs@science.unideb.hu
http://tmcs.math.unideb.hu

Teaching
Mathematics and
Computer Science

Teaching model-based testing

Gábor Árpád Németh

Abstract. Different testing methodologies should play an important role in the education of informatics. In the model-based testing (MBT) approach, the specification of the system is described with a formal model. This model can be used to revise the correctness of the specification and as a starting point for automatic test generation. The main problem with MBT is however, that there is a huge gap between theory and practice and that this approach has a high learning curve. To cope with these problems, current paper shows, how the MBT approach can be introduced to students through a small scale example.

Key words and phrases: model-based testing, test design, teaching.

ZDM Subject Classification: P50.

Introduction

Testing plays a vital role both in software and hardware development. The complexity of hardware and software products is continuously increasing, while the time frame between the releases of different product versions becomes shorter. Unsurprisingly, the fast-paced development increases the probability of faults. Although quality assurance is essential, limited resources are allocated for testing compared to the complexity of the problem. To cope with this challenge, the execution of test cases are done automatically in most big software companies. The next level of automation is the automation of test design. If the requirements of the product are described in a formal model specification, then the test cases

can be generated automatically from this model to fulfill given testing goals. This area of testing is called model-based testing (MBT).

Several formal models exist for system specifications. This paper focuses on Finite State Machine (FSM) formal models, which have been extensively used in diverse areas such as telecommunication software and protocols (Holzmann, 1990), software related to lexical analyzes and pattern matching (Ammann & Offutt, 2008) and embedded systems (Bringmann & Krämer, 2008).

Although MBT would be virtually suitable for all type of problems, its practical application is limited. One of the main obstacles that has kept MBT from becoming widespread, is its huge entry cost. This approach requires a different way of thinking than other testing methodologies; competence build-up is a slow and hard process. Another problem is that there is a huge gap between theory and practice. Different terminologies are used in theory and in practice with different focuses. For example, MBT theory focuses on good coverage, while in practice the length of the test set is even more important. While theory concentrates on the efficiency of test generation algorithms, nowadays the interfaces for (G)UI, existing programming languages and tools are more important for the test engineer, as he would like to integrate this new approach to an existing toolchain. Theory mostly concentrates on rigid models, but in practice the systems evolving continuously and the testing should follow it step-by-step.

To cope with this problem, a course in ELTE Computer Science MSc has been launched, called "Modeling and testing". The curriculum is built on 3 pillars:

(1) State-of-the-art MBT theory research with their usual assumptions

(2) Typical problems and usual assumptions in industry related to MBT

(3) MBT tools

In the current paper, some of the main cornerstones of this MBT teaching material is shown through one of the practical, small-scale examples presented during the course.

Models

Finite State Machines

A Mealy Finite State Machine (abbreviated as 'FSM' in the rest of the paper) M is a quadruple $M = (I, O, S, T)$ where I, O, and S are the finite and nonempty sets of *input symbols*, *output symbols* and *states*, respectively. T is the finite and

nonempty set of *transitions* between states. Each transition $t \in T$ is a quadruple $t = (s_j, i, o, s_k)$, where $s_j \in S$ is the start state, $i \in I$ is an input symbol, $o \in O$ is an output symbol and $s_k \in S$ is the next state.

An FSM can be represented with a *state transition graph*, which is a directed edge-labeled graph whose nodes are labeled with the state symbols and whose edges correspond to the transitions between the states. Each edge is labeled with the input and the output, written as i/o, associated with the transition.

The number of states, transitions, inputs and outputs of an FSM is denoted by $n = |S|$, $m = |T|$, $p = |I|$ and $q = |O|$, respectively.

FSM M is *deterministic*, if for each (s_j, i) state-input pair there exists at most one transition in T, otherwise it is *non-deterministic*. If there is at least one transition $t \in T$ for all state-input pairs, the machine is said to be *completely specified*, otherwise it is *partially specified*.

In case of deterministic and completely specified FSMs, each (s_j, i) state-input pair defines a transition, which can be given as $t = (s_j, i, \lambda(s_j, i), \delta(s_j, i))$, where λ: $S \times I \to O$ denotes the *output function* and δ: $S \times I \to S$ denotes the *next state function* (Lee & Yannakakis, 1996). In this case $m = p \cdot n$.

Two states, s_j and s_l of FSM M are *distinguishable*, iff there exists an $x \in I^*$ input sequence – called a *separating sequence* – that produces different output for these states, i.e. $\lambda(s_j, x) \neq \lambda(s_l, x)$. Otherwise we say that states s_j and s_l are *equivalent*, i.e. $s_j \cong s_l$, iff for all input sequences $x \in I^*$, $\lambda(s_j, x) = \lambda(s_l, x)$. A machine is *reduced*, if no two states are equivalent.

Example FSM: Battery-Operated Signal: The battery-operating toy signal from the Märklin company[1] has the following functionalities:

- The device has 2 LEDs (red and green), a 2-way switch and a press button.
- The 2-way switch turns the device off or on. When the device is turned on, the green LED lights up.
- The button can be used to change which LED lights; red or green.
- If no button is pressed for 7 secs the device changes the lights.

The state transition table and the state transition graph of this FSM is given in Table 1 and Figure 1, respectively (double circle denotes the initial state).

[1]Märklin my world – Battery-Operated Signal 72201, `https://www.maerklin.de/en/products/details/article/72201/`. Accessed: 2019-11-27

	Switch on	Switch off	Press button	Wait for 7 secs
s_{off}	s_{green} / green	-	s_{off} / -	s_{off} / -
s_{green}	-	s_{off}/-	s_{red} / red	s_{red} / red
s_{red}	-	s_{off}/-	s_{green} / green	s_{green} / green

Table 1. The FSM of Battery-Operated signal

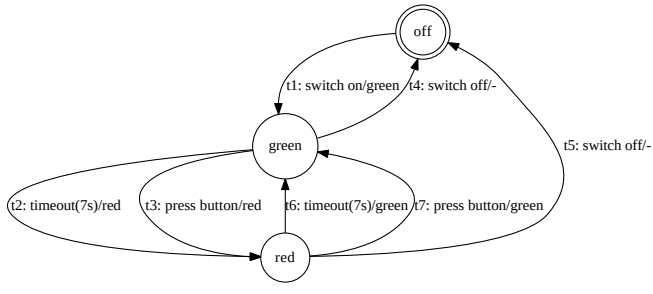


Figure 1. The FSM of Battery-Operated signal

Extended Finite State Machines

The Extended Finite State Machine (EFSM) is an extension of the FSM formalism with variables and operations based on variable values.

Formally, an EFSM M is a quintuple $M = (I, O, S, V, T)$ where I, O, S, V and T are the finite and nonempty sets of *input symbols*, *output symbols*, *states*, *variables* and *transitions*, respectively. Note that inputs are sometimes referred to as events. Each transition $t \in T$ is a sextuple $t = (s_j, i, o, s_k, G_x(V), A_x(V))$, where $s_j \in S$ is the start state, $i \in I$ is an input symbol, $o \in O$ is an output symbol, $s_k \in S$ is the next state, $G_x(V)$ is a guard condition on current variable values and $A_x(V)$ is an action on current variable values.

Example EFSM: Battery-Operated Signal: Extend the functionalities of the previous signal with the following: if the button is pressed 3 times, then the device will play the MÁV music signal, but won't change its state. The counter of button presses is set to 0 when the music signal ends, or when the device is switched on. The EFSM is given in Table 2 and Figure 2. Note that because the guarding condition on variable value *counter* precedes the action on this variable value, the guarding condition is set to 2 instead of 3.

	Switch on	Switch off	[$counter < 2$] Press button	[$counter == 2$] Press button	Wait for 7 secs
s_{off}	s_{green} / green	-	s_{off} / -	s_{off} / -	s_{off} / -
s_{green}	-	s_{off}/-	s_{red} / red	s_{green} / green, play MÁV music signal	s_{red} / red
s_{red}	-	s_{off}/-	s_{green} / green	s_{red} / red, play MÁV music signal	s_{green} / green

Table 2. The EFSM of Battery-Operated signal

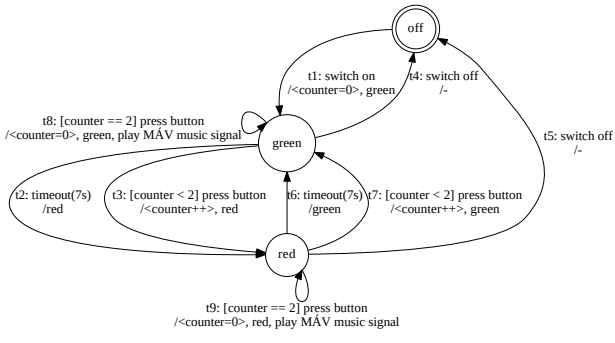
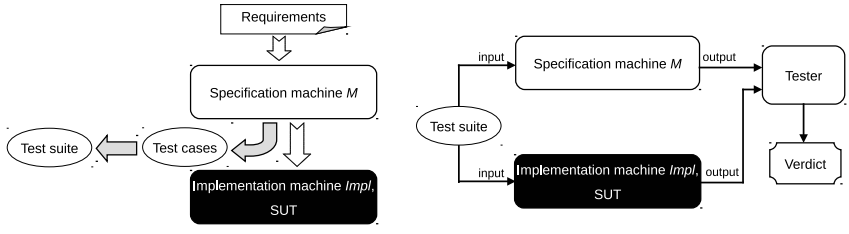


Figure 2. The EFSM of Battery-Operated signal

Conformance testing

The structure of model-based test generation and testing is shown in Figure 3. From the requirements a formal specification, an abstract model is created, denoted by M. M can be considered a white-box with a known internal structure: its state transition graph is given. Based on M, an implementation is created, denoted by $Impl$. $Impl$ is considered a black-box with unknown internal structure: we can only observe its output responses upon given inputs. *Test cases* are derived from M; these are the pairs of input sequences and expected output sequences of M. A set of test cases form a *test suite* – see Figure 3(a). This test suite is applied to the System Under Test (SUT) $Impl$ and the tester checks if the observed output sequences of $Impl$ are equivalent to the expected results derived from M – see Figure 3(b). This type of testing is called *conformance testing*.



(a) Test generation (b) Conformance testing

Figure 3. Model-based testing

FSM Fault Models

Fault models describe the assumptions of the test engineer about the implementation machine he is about to test. For completely specified and deterministic FSMs the following three types of faults was proposed (Chow, 1978):

I. Output fault: for a given state-input pair FSM $Impl$ produces an output that is different from the one that is specified in FSM M.

II. Transfer fault: for a given state-input pair FSM $Impl$ goes into a state that differs from the state specified in FSM M.

III. Missing state or extra state

For non-deterministic and non-completely defined FSMs, the fault model of (Chow, 1978) was extended with the following faults (Bochmann et al., 1992):

IV. Additional or missing transitions

Analogously for EFSMs the following additional faults may exist:

V. Guarding condition fault: different guarding condition than specified.

VI. Action fault: different action or action on different variable than specified.

VII. Additional or missing variables

A usual approach made in literature is that the faults do not increase the number of the states of the machine (Lee & Yannakakis, 1996), thus the FSM fault model of (Chow, 1978) and (Bochmann et al., 1992) are typically restricted to output and transfer faults (Lee & Yannakakis, 1996).

Assumptions About Specification and Implementation Machines

For (E)FSM-based testing, the subset of the following assumptions are usually made about specification and implementation machines M and $Impl$ (Yannakakis & Lee, 1995; Lee & Yannakakis, 1996; Broy et al., 2005):

1. M is completely specified. Our example FSM and EFSM are partially specified as *switch_off* and *switch_on* functions are not defined for all states, however, they can easily be converted to completely specified machines by adding loop transitions for these undefined state-input pairs.
2. M is deterministic. Our example FSM and EFSM are deterministic.
3. M is strongly connected. Otherwise we would not able to reach all states. Our example FSM and EFSM are strongly connected.
4. M is reduced[2]. The reason for requiring a reduced specification machine M is that by observing only the output sequences of $Impl$ upon given input sequences, one can not distinguish between equivalent machines. Our example machines are reduced (*button* input produces a different output for all states).
5. $Impl$ does not change during the test experiment and has the same input I and output O alphabet as M.
6. An upper bound is required for the number of states of $Impl$. Otherwise, it would be possible that the faulty part of $Impl$ is never reached by our test. It is usually assumed that $Impl$ has no more states than M (Yannakakis & Lee, 1995; Broy et al., 2005).
7. M and $Impl$ have a *reset message*. The reset message is a special input symbol that takes the machine from any state back to the initial state. For EFSMs it also initializes variables. The *reset is reliable* if it is guaranteed to work properly in any implementation machine $Impl$ of M. In our examples *switch_off* works as reset as it takes the machines back to the s_{off} initial state.

Coverage criteria

Structural coverage criteria checks the machine systematically based on preset structural roles. The most important are the following:

[2]Note that if M is not reduced, then it can always be transformed to an equivalent reduced machine (for example with the minimization method described in (Gill, 1962)).

- **all states:** Each state is visited at least once by the test suite. It has $O(n)$ complexity[3]. In our examples s_{off}, s_{green}, s_{red} states need to be covered, what can be done with $t_1.t_3$, thus the test sequence is *switch_on.button*.
- **all events:** Each input (event) is called at least once by the test suite. It has $O(p) \leq O(m)$ complexity[3]. In our FSM and EFSM examples *switch_on*, *switch_off*, *press_button*, *timeout* events should be covered, what can be done with $t_1.t_3.t_6.t_4$, thus the test sequence is *switch_on.button.timeout.switch_off*.
- **all outputs** Each output is checked at least once by the test suite. It has $O(q) \leq O(m)$ complexity[3]. In our FSM example -, *green*, *red* outputs should be covered, what can be done with $t_1.t_3.t_5$, thus the test sequence is *switch_on.button.switch_off*. In our EFSM example -, *green*, *red*, *play MÁV music signal* outputs should be covered, what can be done with $t_1.t_3.t_7.t_8.t_4$, thus the test sequence is *switch_on.button.button.button.switch_off*.
- **all transitions:** Each transition is traversed at least once by the test suite. It has $O(m)$ complexity[3]. In the FSM example $t_1 - t_7$ transitions should be covered, what can be done with $t_1.t_2.t_6.t_3.t_5.t_1.t_3.t_7.t_4$, thus the test sequence is *switch_on.timeout.timeout.button.switch_off.switch_on.button.button.switch_off*. In the EFSM example $t_1 - t_9$ transitions should be covered, what can be done with $t_1.t_3.t_7.t_8.t_4.t_1.t_2.t_6.t_2.t_7.t_3.t_9.t_5$, thus the test sequence is *switch_on.button.button.button.switch_off.switch_on.timeout.timeout.timeout.button.button.button.switch_off*.

  Note that beside structural coverage criteria, other criteria exist:

- **functional criteria:** Go through scenarios, use cases or user profiles. In the examples above $t_1.t_2.t_6$ transitions check the timeout functionality, thus the test sequence of this functionality is *switch_on.timeout.timeout*.
- **stochastic criteria:** Given transitions from given states are selected with a given probability. The test coverage should reflect on these probabilities with weighted random walks. This criteria can be used for risk based testing, where more weight is assigned to more important features.
- **fault coverage criteria:** If some assumptions hold, then some algorithms (see next section) are able to show the absence of given type of faults.

Example: Battery-Operated Signal: We investigate with all transitions coverage the case, when instead of *2*, *3* is used as a parameter for guarding

[3]Note that this complexity is true for FSM models. In case of EFSMs, the complexity can be greater due to the actions and guarding conditions of some variables on given transitions.

	State	Event (Input)	Next state	Expected result	Observed result
Step 1	s_{off}	Switch on	s_{green}	green	green
Step 2	s_{green}	[0 < 2] Press button	s_{red}	red	red
Step 3	s_{red}	[1 < 2] Press button	s_{green}	green	green
Step 4	s_{green}	[2 == 2] Press button	s_{green}	green, play MÁV music signal	red
Step 5	s_{green}	Switch off	s_{off}	-	-
Step 6	s_{off}	Switch on	s_{green}	green	green
Step 7	s_{green}	Timeout	s_{red}	red	red
Step 8	s_{red}	Timeout	s_{green}	green	green
Step 7	s_{green}	Timeout	s_{red}	red	red
Step 9	s_{red}	[0 < 2] Press button	s_{green}	green	green
Step 11	s_{green}	[1 < 2] Press button	s_{red}	red	red
Step 12	s_{red}	[2 == 2] Press button	s_{red}	red, play MÁV music signal	green
Step 13	s_{red}	Switch off	s_{off}	-	-

Table 3. All transitions coverage: Expected vs. Observed results

condition to *button* due to an implementation error. The results are shown in Table 3. The difference from the expected and the observed output at Steps 4 and 12 clearly indicates that the implementation is faulty.

Test generation algorithms

Many methods have been introduced to create a test set from an FSM model (Yannakakis & Lee, 1995; Lee & Yannakakis, 1996; Broy et al., 2005). These algorithms can be divided into the following two subclasses:

- Algorithms that fundamentally consist of one stage, which checks all the transitions of the given implementation machine $Impl$.
- Algorithms that fundamentally consist of two stages. The first stage (state identification) checks for each state of the specification whether it exists in the implementation as well. The second stage (transition testing) checks all remaining transitions of the implementation by observing whether the output and the next state conform to the specification.

The methods also differ in the way they check whether the machine is in the expected state. The following preset sequences exists for state verification:

- **Separating family of sequences**. A *separating family of sequences* (Yannakakis & Lee, 1995) or *family of Harmonized State Identifiers (HSI)* (Petrenko, Yevtushenko, Lebedev, & Das, 1994)) of FSM M is a collection of sets $Z_i, i = 1, \ldots, n$ of sequences (one set for each state), which satisfies the following two conditions: for every non-identical pair of states s_i, s_j:

(I) there exists an input sequence x that separates them, i.e. $\exists x \in I^*$, $\lambda(s_i, x) \neq \lambda(s_j, x)$; (II) x is a prefix of some sequence in Z_i and some sequence in Z_j. Such a family may be constructed for every deterministic, reduced FSM: for any pair of states s_i, s_j a sequence z_{ij} is generated that separates them using a minimization method (Gill, 1962) for example. Then the separating sets are defined as $Z_i = \{z_{ij}\}, j = 1 \ldots n$ for each state s_i.

- **Characterizing set**. A *Characterizing Set (CS)* W of FSM M is a set of input sequences such that every non-identical pair of states s_l, s_x can be distinguished by at least one member of W, i.e. $\forall s_l, s_x \in S, l \neq x : \exists w_q \in W : \lambda(s_l, w_q) \neq \lambda(s_x, w_q)$. The CS is a special case of the separating family of sequences: the sets Z_i are identical. For every completely specified, deterministic, reduced FSM, a CS can be generated (Gill, 1962).
- **UIO sequence**. A *Unique Input Output (UIO) sequence* u_l for state s_l of machine M is a sequence that verifies the given state s_l of M, i.e. $\lambda(s_l, u_l) \neq \lambda(s_x, u_l)$, $\forall s_x \in S, l \neq x$. Only completely specified, deterministic reduced FSMs may have UIO sequences for all states, but not all of them do (Broy et al., 2005).
- **Distinguishing Sequence**. A *Distinguishing Sequence (DS)* of machine M is an input sequence d that gives different output for every state of FSM M, i.e. $\forall s_j, s_l \in S, j \neq l : \lambda(s_j, d) \neq \lambda(s_l, d)$. Thus, a DS of FSM M is able to *identify* any state of M. Only completely specified, deterministic reduced FSMs may have DS, however, very few real specification FSMs have DS (Ural, 1992).

If the output observed by the implementation machine can be looped back to the test cases, it is possible to use *adaptive* state verification sequences and sets. In this case the extra information – the observed output of the FSM – may result in a shorter state verification process. Note that the preset/adaptive test path generation is referred to as *offline/online* MBT in industry.

The most important test generation algorithms are summarized in Table 4.

Method	Structure	Required assumptions	State Identification	Complexity of test generation	Length of test	Comments
TT	1 stage	2, 3, 5, 6	-	$O(n^3 + m)$	$O(m)$	Guarantees to find output faults only
D	2 stages	1-6, DS exists	DS	PSPACE	Exp.	Applicable only if DS exists (it rarely exists)
UIOv	2 stages	1-7, UIO seqs. exists	UIO seq.	PSPACE	Exp.	Applicable only if UIO sequences exists for each state
W	1 stage	1-7	CS	$O(p \cdot n^3)$	$O(p \cdot n^3)$	-
Wp	2 stages	1-7	CS	$O(p \cdot n^3)$	$O(p \cdot n^3)$	Improved version of W
HSI	2 stages	2-7	sep. family of seqs.	$O(p \cdot n^3)$	$O(p \cdot n^3)$	Similar to Wp
H	2 stages	2-7	adaptive selection from sep. family of seqs.	no preset test	$O(p \cdot n^3)$	Adaptive version of HSI

Table 4. The differences between FSM-based test generation algorithms

Note that the Transition Tour (TT) method (Naito & Tsunoyama, 1981) generates the shortest possible test sequence, which provides 100% transition and 100% state coverage. The TT is equivalent to the Directed Chinese Postman Problem mathematical problem (Edmonds & Johnson, 1973) with unit costs for each edge. The TT-method has the least assumptions about specification and implementation FSMs, but it only guarantees to find output faults. For continuously evolving FSMs, the incremental TT-method is proposed in (Németh & Pap, 2014) that updates the TT test sequence after changes applied to the model.

Although both the D-method (Hennine, 1964) and the UIOv-method (Vuong, Chan, & Ito, 1989) guarantee to show the existence of all output and transition faults, their applications are very limited: DS and UIO sequence may not exist for all machines.

The very similar W(Chow, 1978)/Wp(Fujiwara, v. Bochmann, Khendec, Amalou, & Ghedamsi, 1991)/HSI(Yannakakis & Lee, 1995; Petrenko et al., 1994)-methods guarantee to find all output and transfer faults. The most general approach of this triple is the HSI method that can be used for partially specified FSMs also. For continuously evolving FSMs, the incremental HSI-method is proposed in (Pap, Subramaniam, Kovács, & Németh, 2007), which identifies the effects of changes in the test suite and only updates those parts that are necessary.

The state verification process of the HSI-method can be shortened with its adaptive version called the H-method (Dorofeeva, El-Fakih, & Yevtushenko, 2005).

Testing with GraphWalker

For the course, GraphWalker[4] was selected to illustrate MBT, because it is a documented, well supported, easy-to-use, free and open source tool.

The working process of GraphWalker consists of the following steps:

1. Creating an (E)FSM model.
2. Creating the adaptation code (in C#, python or perl) related to the graph that interacts with the SUT.
3. Executing the test sequences. Test sequences are generated mainly with different random walks and with different stopping conditions.

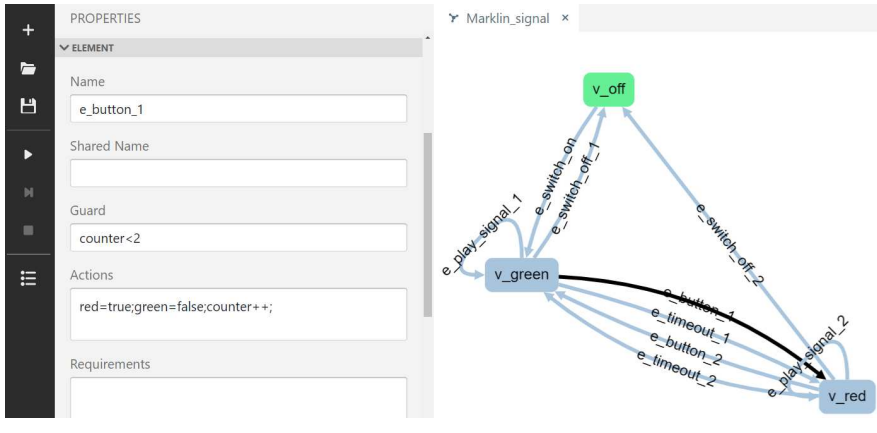


Figure 4. EFSM of the battery-operated signal in GW studio

The FSM can be edited in a GUI (graphical user interface) with GraphWalker studio. During the course, the model of the battery-operated signal is built step-by-step starting from a simple FSM model (introduced in Figure 1) and finished with its EFSM extension (introduced in Figure 2), which contains variables and guard conditions. The final version edited in GW studio is shown in Figure 4.

The different generators and stopping conditions can also be set via GUI:

- *random* path generator always selects a transition originating from a state by random, and repeats this process in the next state till the stopping condition is fullfilled. This may result in a large test sequence even in small models. For

[4]GraphWalker. Model-based testing tool, `https://graphwalker.github.io/`. Accessed: 2019-11-27

example for our FSM model (Figure 1), this path generator with 100% edge coverage stopping condition resulted in 10, 21, 22, 22 and 34 edge traversals in our 5 conducted experiments, while the optimal solution generated by the TT-method (Naito & Tsunoyama, 1981) required only 9 edge traversals. The 100% edge coverage stopping condition for the EFSM extension (Figure 2) resulted in 54, 73, 113, 116 and 275 edge traversals in our 5 conducted experiments, while the optimal solution (presented previously in transition coverage) required 13 only edge traversals.

- *quick_random* path generator is capable of generating a relative short path (by always selecting a previously non-visited edge by random, and finding the shortest path to that edge using Dijkstra's algorithm (Dijkstra, 1959)), but it does not deal with guarding conditions, i.e. it selects a new edge despite of these conditions. For our FSM model (Figure 1) this path generator traversed 9, 9, 11, 11 and 12 edges in our 5 conducted experiments, respectively, to fulfill 100% edge cover condition, but it can not be used for our EFSM model.
- *weighted_random* path generator works similarly as *random*, but it uses edge weights, which represent the probability of an edge getting chosen. For our EFSM example (Figure 2) we set edge weights for transitions originating from states s_{red} and s_{green}. Input symbols *button*, *timeout*, *switch_off* get 0.6, 0.3 and 0.1 weights, respectively. The relative high weight of *button* compared to the relative low weight of *switch_off* provides enough probability to reach guard condition *counter* == 2 and thus the traversals of t_8 and t_9 edges. This path generator with 100% edge coverage stopping condition resulted 21, 45, 48, 70, 97 edge traversals in our 5 conducted experiments.

During the curse, different stopping conditions are also investigated such as relaxing the edge coverage to 50% (*edge_coverage (50)*), 100% state coverage (*vertex_coverage(100)*), reaching a given state (*reached_vertex(v_red)*), 10 second time duration (*time_duration(10)*), 70% requirement coverage (*requirement_coverage (70)*)...etc.

Note that most of the MBT tools generate test codes only for symbolic execution, which run on implementation models rather than on actual implementations. GraphWalker does the same. Thus, a *glue code* or *adaptation code* is required to "adapt" the MBT generated test case to the SUT to be executable. GraphWalker documentation gives few example projects (like PetClinic, Amazon Shopping Cart...etc.) that show, how this adaptation code can be made for example to test a web page using Selenium, but this is beyond the scope of this article.

The path generators of GraphWalker create random sequences[5], thus the tests are not repeatable. For this reason, it is not suitable for regression testing, i.e. to retest a code after a bugfix. GraphWalker is however suitable to exploratory testing, or to check the strength of a given coverage by executing random walks.

Didactic of teaching model-based testing

During the course, theoretical information is accompanied by slides to help comprehension; the algorithms are explained through step-by-step animation examples, the references for the most important state-of-the-art research papers and books are given. The students work in groups or individually to solve some simple real-life problems (like the presented toy signal, some basic functionalities of the SIP (Session Initiation Protocol) (*RFC 3261*) telecommunication protocol...etc.). These tasks help to ensure that the students have understood how models are created and tests generated. Invited guests from diverse domains of the industry show how the actual testing process is done in practice; what the typical problems are and which toolchains can be used. Note that besides functional testing we also discuss how some type of models can be used in performance testing.

During the term, the performance of the students is evaluated with project home works. First, they create requirements for a selected problem, then they create and refine an EFSM specification model step-by-step based on the requirements. The requirement engineering and modeling process involves close communication with the stakeholder (consultations with the lecturer in our case), to have a common understanding of the problem, to identify misunderstandings and to avoid defects (such as incomplete, inconsistent and unclear requirements) at early stages. This part also helps them to focus their attention on typical problems of modelling process and to cope with them (such as selecting the appropriate level of details for the model). If an easier problem is selected as a project (like testing a simple webportal or testing the main functionalities of a route planning application), then the students also create an adaptation code for the model, to be able to test an actual implementation. If a more complicated problem is selected (for example controlling the measurements of a big telescope depending on weather conditions, conference call functionality of SIP (*RFC 4579*)...etc.), then they will create a dummy code for symbolic test execution. The students then generate

[5]Although *a_star* generator creates paths systematically, it is only capable of finding the shortest path to a given state/transition.

and execute tests with GraphWalker or with an other free MBT tool with various coverage criteria. Finally they create a test report and conclude their findings in a document.

Conclusion

Even for students with some testing experience it is difficult to switch from the approach of designing test cases manually to the approach of creating a formal model thoughtfully, understanding the advantages and disadvantages of different coverage criteria and test generation algorithms to select the appropriate one that will generate tests automatically. The different terminologies and focuses on academia and industry makes it more difficult the enter the field of MBT.

During the course, the students learn the basics of MBT, the challenges and possible approaches to solve these problems both from the academic and industrial perspective. Given the syllabus of the course, the presented small scale real examples and the practical experiences gained from the project home works, students become capable of using MBT for the testing of simple applications at the end of the term.

Acknowledgements

The project is supported by the Hungarian Government and co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies).

References

Ammann, P., & Offutt, J. (2008). *Introduction to Software Testing* (1st ed.). New York, NY, USA: Cambridge University Press.

Bochmann, G. v., Das, A., Dssouli, R., Dubuc, M., Ghedamsi, A., & Luo, G. (1992). Fault Models in Testing. In *Proceedings of the IFIP TC6/WG6.1 fourth international workshop on protocol test systems iv* (pp. 17–30). Amsterdam, The Netherlands: North-Holland Publishing Co. Retrieved from `http://dl.acm.org/citation.cfm?id=648126.747577`

Bringmann, E., & Krämer, A. (2008). Model-based testing of automotive systems. In *Proceedings of the 2008 international conference on software testing, verification, and validation* (pp. 485–493). Washington, DC, USA: IEEE Computer Society. doi: 10.1109/ICST.2008.45

Broy, M., Jonsson, B., Katoen, J. P., Leucker, M., & Pretschner, A. (2005). *Model-Based Testing of Reactive Systems*. Springer.

Chow, T. (1978, May). Testing software design modelled by finite-state machines. *IEEE Transactions on Software Engineering*, *4*(3), 178–187.

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, *1*(1), 269–271.

Dorofeeva, R., El-Fakih, K., & Yevtushenko, N. (2005). An Improved Conformance Testing Method. In F. Wang (Ed.), *Formal techniques for networked and distributed systems – forte 2005* (Vol. 3731, pp. 204–218). Springer, Berlin, Heidelberg. Retrieved from `http://dx.doi.org/10.1007/11562436_16` doi: 10.1007/11562436_16

Edmonds, J., & Johnson, E. L. (1973). Matching, Euler tours and the Chinese postman. *Mathematical Programming*, *5*(1), 88–124.

Fujiwara, S., v. Bochmann, G., Khendec, F., Amalou, M., & Ghedamsi, A. (1991). Test selection based on finite state model. *IEEE Transactions on Software Engineering*, *17*(6), 591–603. doi: 10.1109/32.87284

Gill, A. (1962). *Introduction to the theory of finite-state machines*. McGraw-Hill. Retrieved from `http://books.google.hu/books?id=2IzQAAAAMAAJ`

Hennine, F. C. (1964). Fault detecting experiments for sequential circuits. In *Proceedings of the fifth annual symposium on switching circuit theory and logical design* (pp. 95–110). Los Alamitos, CA, USA: IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/SWCT.1964.8

Holzmann, G. J. (1990). *Design and Validation of Protocols*. Prentice-Hall.

Lee, D., & Yannakakis, M. (1996). Principles and Methods of Testing Finite State Machines – A Survey. *Proceedings of the IEEE*, *84*(8), 1090–1123.

Naito, S., & Tsunoyama, M. (1981). Fault detection for sequential machines by transition-tours. In *Proceedings of the 11th IEEE Fault-Tolerant Computing Conference (FTCS 1981)* (pp. 238–243). IEEE Computer Society Press.

Németh, G. A., & Pap, Z. (2014, July). The Incremental Maintenance of Transition Tour. *Fundam. Inf.*, *129*(3), 279–300. doi: 10.3233/FI-2014-972

Pap, Z., Subramaniam, M., Kovács, G., & Németh, G. A. (2007). A Bounded Incremental Test Generation Algorithm for Finite State Machines. In *Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th*

International Conference on Testing of Software and Communicating Systems (pp. 244–259). Berlin, Heidelberg: Springer-Verlag.

Petrenko, A., Yevtushenko, N., Lebedev, A., & Das, A. (1994). Nondeterministic State Machines in Protocol Conformance Testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI* (pp. 363–378).

Ural, H. (1992). Formal methods for test sequence generation. *Computer Communications*, *15*(5), 311–325. Retrieved from `http://www.sciencedirect.com/science/article/pii/014036649290092S` doi: 10.1016/0140-3664(92)90092-S

Vuong, S. T., Chan, W. W. L., & Ito, M. R. (1989). The UIOv-Method for Protocol Test Sequence Generation. In J. de Meer, L. Machert, W. Effelsberg, & North-Holland (Eds.), *Proceedings of the IFIP TC6 2nd International Workshop Protocol Test Systems* (pp. 161–175).

Yannakakis, M., & Lee, D. (1995). Testing finite state machines: fault detection. In *Selected papers of the 23rd annual acm symposium on theory of computing* (pp. 209–227).

GÁBOR ÁRPÁD NÉMETH
EÖTVÖS LORÁND UNIVERSITY, FACULTY OF INFORMATICS, DEPT. OF COMPUTERALGEBRA
H-1117 BUDAPEST, PÁZMÁNY PÉTER SÉTÁNY 1/C.

E-mail: `nga@inf.elte.hu`

(Received November, 2019)