MTR Model-Based Testing Framework

Gábor Árpád Németh and Máté István Lugosi

***Abstract*—In this article we propose a novel, free and open-source model-based testing framework for finite state machine specifications. The various model handling and test generation options make the framework suitable for testing complex systems and provide a solid background for investigating different automated test design methodologies. The complexity and fault detection capabilities of the available algorithms are investigated through analytical analyses and simulations applying randomly injected faults.**

***Index Terms*—model-based testing, conformance testing, finite state machine, test generation algorithm**

I. INTRODUCTION

In software development, testing is a critical, but often resource-intensive process. Although test execution is automated in most big software companies, test design is typically done manually, which – considering the rapidly changing complex products – tends to be an ad-hoc, error-prone and time-consuming approach. Model-based testing (MBT) turns this costly and labour intensive task into an automated process. In MBT, the requirements of the product are described as a formal model and the test cases are derived automatically from this model.

This article focuses on the MBT of Finite State Machine (FSM) specifications [9], [18], [22], which have been extensively used in different problem domains such as telecommunication software and protocols [16], [17], pattern matching [3], hardware design [26], and embedded systems [8]. A number of academic and commercial tools are developed to support MBT [6], [19]. Commercial products for FSM-like specifications include Conformiq Designer[1] and Reactis Tester[2], but these are not open-source. GrapWalker (GW)[3], fMBT[4], and Modbat[5] are free and open-source FSM-based tools that are actively under development. GW has an easy-to-use graphical user interface (GUI), but test generation is mainly done by random traversals; it lacks efficient systematic routing algorithms [30]. fMBT generates test cases from converted Extended FSMs using random and other heuristics to fulfill a given coverage (such as permutations of consecutive elements). Modbat is specialized to testing the application programming interface (API) of a software [4], test cases can be generated by heuristic search.

In this article we present a new, free and open-source model-based testing framework – called $Model \gg Test \gg Relax$ (MTR) – for FSM specifications. With MTR the test engineer can import specification models from GW, apply different conversions on the model and generate tests. The variety of systematic test generation algorithms and their different settings provide the potential to the test engineer to balance between quality aspects and the resources required for testing.

The body of the article is organized as follows. Section II discusses related terms regarding FSMs and MBT. Section III overviews the main functionalities of the MTR framework, Section IV describes its working process. The different test generation strategies are summarized in Section V, our new algorithm created for N-Switch Coverage is also briefly discussed here. Section VI presents simulations for test generation algorithms investigating the complexity of automated test creation, the size and the fault coverage of the resulting test suites. Possible future directions are discussed in Section VII, the main results of the paper are concluded in Section VIII.

II. PRELIMINARIES

A. Finite State Machines

A Mealy Finite State Machine (FSM) M is a quadruple $M = (I, O, S, T)$ where I, O, S and T are the finite and non-empty sets of *input symbols*, *output symbols*, *states* and *transitions* between states, respectively. 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. The number of states, inputs and transitions are denoted by n, p and m, respectively.

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

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 FSMs the output and the next state of a transition can be given as a function of the start state and the input of a transition, where λ: $S \times I \rightarrow O$ denotes the *output function* and δ: $S \times I \rightarrow S$ denotes the *next state function*. Let us extend δ and λ from input symbols to finite *input sequences* I^* as follows: for a state s_1, an input sequence $x = i_1, \ldots, i_k$ takes the machine successively to states $s_{j+1} = \delta(s_j, i_j)$, $j = 1, \ldots, k$ with the final state $\delta(s_1, x) = s_{k+1}$, and produces an *output sequence* $\lambda(s_1, x) = o_1, \ldots, o_k$, where $o_j = \lambda(s_j, i_j)$, $j = 1, \ldots, k$. An FSM M is *strongly connected* iff for each pair of states $(s_j\ ,\ s_l)$, there exists an input sequence which takes M from s_j to s_l.

[1] Gábor Árpád Németh and Máté István Lugosi are with the Department of Computer Algebra, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary, (e-mail: nga@inf.elte.hu, mate.lugosi@gmail.com)

[1] Conformiq Designer, https://www.conformiq.com/products/

[2] Reactis Tester, https://www.reactive-systems.com/products.msp

[3] GraphWalker, https://graphwalker.github.io/

[4] fMBT, https://github.com/intel/fMBT

[5] Modbat, https://gitlab.com/cartho/modbat

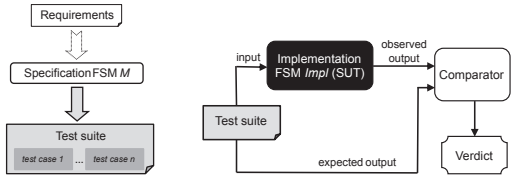
DOI: 10.36244/ICJ.2024.2.2

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 states s_j and s_l are *equivalent*. A machine is *reduced*, if no two states are equivalent.

An FSM M has a *reset message*, if there exists a special input symbol $r \in I$ that takes the machine from any state back to the s_0 initial state: $\exists r \in I : \forall s_j : \delta(s_j, r) = s_0$. The *reset is reliable* if it is guaranteed to work properly in any implementation machine $Impl$ of M; otherwise it is *unreliable*. A machine with reset capability is strongly connected, iff each state $s_j \in S$ is reachable from s_0.

The Extended Finite State Machine (EFSM) is an extension of the FSM formalism with variables, actions and guarding conditions over variables.

B. Model-based testing



(a) Test generation (b) Conformance testing

Figure 1. FSM model-based test generation and testing

The process of FSM model-based test generation is illustrated in Figure 1(a): From the requirements, an FSM M specification model is created. *Test cases* – which are the pairs of input sequences and expected output sequences of M – are generated automatically from this model. A set of test cases forms a *test suite*. The resulting test suite can be applied to the System Under Test (SUT) which can be considered as an $Impl$ implementation machine of specification M with an unknown internal structure, thus one can only observe its output responses upon a given input sequence – see Figure 1(b). *Conformance testing* checks if the *observed output sequences* of $Impl$ are equivalent to the *expected output sequences* derived from M – i.e. it determines if $Impl$ *conforms* to M.

Note that to connect the specification model to an actual SUT, a source code, called *adaptation code* needs to be created, that adapts the specification model to the SUT[6]. The adaptation code implements each element of the specification model as *keywords*. Such keywords are created for each transition of the specification model. Utilizing the adaptation code, one can transform abstract test cases into executable ones to effectively test the SUT.

C. FSM Fault Models

Fault models describe the assumptions of the test engineer about the implementation machine (s)he is about to test. For completely specified, deterministic FSMs the following three types of faults were proposed [10]:

I. Output fault: for a given state-input pair, FSM $Impl$ produces an output that differs from the one specified in machine M.
II. Transfer fault: for a given state-input pair, $Impl$ goes into a state that differs from the one specified in M.
III. Missing state or extra state

For non-deterministic and partially defined FSMs, the fault model above was extended with the following [7]:

IV. Missing or additional transitions

A usual assumption made in literature is that the faults do not increase the number of the states of the machine [18], thus the fault models of Chow [10] and Bochmann et al. [7] are typically restricted to output and transfer faults [18].

III. Overview of the Model ≫ Test ≫ Relax Framework

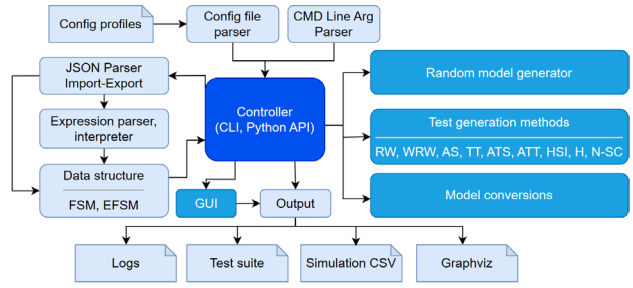


Figure 2. High level overview of $Model \gg Test \gg Relax$ framework

Figure 2 presents a high level overview of the architecture of the $Model \gg Test \gg Relax$ (MTR) model-based testing framework[7]. The user can import existing FSM or EFSM models or generate random ones, and can also make conversions on models (see Section IV-A). A wide range of algorithms can be utilized for test suite generation (see Section V) and an interface file can also be created that can be used as a skeleton for adaptation code creation (see Section IV-C). The parameters of the tool can be set by Command Line Interface (CLI) or by configuration profiles. Note that three different configuration profiles are delivered with the framework, optimized for testing, research and education purposes, respectively. Besides the generated test suite, the tool provides the following files to evaluate the results:

- *logs*: Six verbosity levels can be selected.
- *CSV file*: Comma-separated values summarize the main parameters of the model, in addition to the parameters and the results of the selected test generation algorithm.
- *Graphviz*[8] *file*: The models and the results of the applied test generation method can be visualized using this file.

The framework was implemented in C++ using the LEMON[9] library.

[6]Sometimes it is also referred as "glue code" as it glues the model and SUT together. In some cases – based on the abstraction level of the specification model and the applied testing tools – this adaptation code can be partially or completely generated.

[7] *Model ≫ Test ≫ Relax*. https://modeltestrelax.org/, https://gitlab.inf.elte.hu/nga/ModelTestRelax

[8] Graphviz. Graph visualization software. https://graphviz.org/

[9] Library for Efficient Modeling and Optimization in Networks (LEMON), http://lemon.cs.elte.hu

IV. WORKING PROCESS OF THE MODEL ≫ TEST ≫ RELAX FRAMEWORK

A. Model making and manipulations

1) Model import: The specification model is defined in a JSON[10] format that is similar[11] to the one used by GraphWalker (GW). Thus, the user is able to create a model in the GUI of GW Studio and import it to our framework.

2) Model generation: It is possible to generate random FSMs with different parameters for simulation purposes.

3) Model conversions and manipulations: MTR provides the following conversion options to manipulate models:

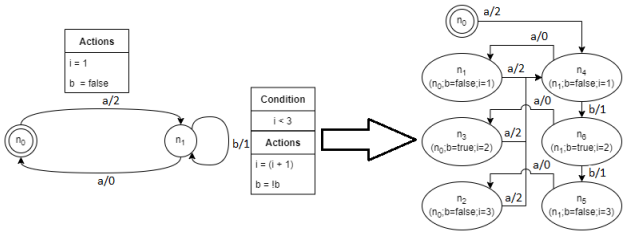


Figure 3. EFSM → FSM model conversion

- *EFSM → FSM model conversion:* For each possible state-variable value combination (that can be reached within the EFSM from the initial state considering the actions and guarding conditions of transitions[12]) a distinct state will be created in the converted FSM – see Figure 3. The conversion results in the well-known state explosion problem [18], but one can limit the range of variables. FSM test generation methods can be applied on the converted model and the adaptation keywords need to be implemented only once for each transition of the EFSM specification (parameterized by variables)[13].
- *Partially specified → completely specified conversion:* For each undefined state and input symbol pair a loop transition is added without an output symbol.
- *State minimization:* Helps the design phase of the formal specification by converting non-reduced machines into reduced ones merging equivalent states.
- *Add/remove reset*: Add/remove reliable or unreliable reset transitions to the model in one step.
- *Error injection:* Model-based mutation testing (MBMT) [5] can be applied by injecting given number of random transfer, output, missing or additional transition faults to the model. With this functionality one can investigate the fault detection capabilities of different test generation methods with simulations that apply the test suites generated from unmodified models to modified models.

[10] JSON. https://www.json.org/

[11] There are some differences in the model handling of GW and MTR as GW does not follow the (E)FSM formalism completely. Thus, some conversion is required if one would like to import the model of GW into MTR, but this is described in detail in the ”5.1.2. Editing models using GraphWalker Studio” section of MTR User Guide.

[12] If some states cannot be reached – thus they are not added to the converted model – then MTR displays a warning message.

[13] Note that an application example (OpenIddict) is delivered with MTR that tests the main functionalities of the Oauth 2.0 [2] protocol using an EFSM model and shows how the EFSM → FSM conversion and the testing on the converted model works.

B. Test generation

The framework contains numerous algorithms for test suite derivation with varying complexities and fault coverages enabling the test engineer to find an appropriate trade-off between resources allocated for the generation and execution of tests and the quality of the SUT. The available methods are described in Section V.

C. Test execution

MTR generates an interface file that contains the elements of the model and can be used when writing the adaptation code. The adaptation code should contain the following steps:

- STEP 1: Parse the next element of the test suite – that consists of an input/output list – generated by MTR.
- STEP 2: Execute actions corresponding to the given element of the input list.
- STEP 3: Check the result with assert if it corresponds to the one that can be expected from the given element of the output list.

Note that sample test projects are also delivered with the framework[14] which can be utilized as a base to understand the modelling and adaptation code creation processes before one creates an own test project. Each project contains the following parts:

- An FSM or EFSM model that describes the specification of the SUT.
- An adaptation code which implements each element of the specification model.
- A SUT that is provided by an external link.

Also note that MTR provides an option to export the generated test suites into GW and thus write the corresponding adaptation code there.

V. TEST GENERATION ALGORITHMS

Table I summarizes the available test generation algorithms in MTR and their main properties. A brief description is given for all algorithms in the following subsections and simulation results are presented in Section VI.

A. Random Walk (RW)

Random Walk (RW) starts from the initial state of the machine and in each step a transition leading from the current state is selected randomly and traversed entering a new state. The former step is executed until a given stop condition – the preset percentage of states or transitions have been visited – is fulfilled. MTR also provides an option to set selection probabilities for each transition of the model.

Although RW is unsuitable for the functional testing of large-scale software (as the length of the test sequence is not bounded and thus can be much longer than the optimal solution) and for regression testing (due to the randomness of transition traversals), it can be useful for exploratory testing of a new functionality.

[14] These projects can be accessed in folder *sample_models / applications*

TABLE I
THE MAIN ASPECTS OF TEST GENERATION ALGORITHMS

Algo.	Model	Complexity of test generation	Complexity of test suite	Structure of test suite	Coverage and other notes
RW	FSM / EFSM	Not bounded	Not bounded	1 test sequence	Given percentage of state/transition coverage (based on stop condition).
AS	FSM	$O(n^2)$	$O(m)$	1 test sequence	100% state coverage
TT	FSM	$O(n^3 + m)$	$O(m)$	1 test sequence	100% state and transition coverage. Guarantees to find output faults.
ATS	FSM	ATS0 (standard): $O(n^3 + m)$, ATSa/x (iterative): $O(\eta(n^3 + m))$, $\eta < 2 \cdot n$	ATS0: $O(m)$, ATSa/x: $O(\eta \cdot m)$, $\eta < 2 \cdot n$	1 test sequence (with subparts)	100% state and transition coverage. Guarantees to find output faults.
ATT	FSM	$O(m(n^3 + m))$	$O(m^2)$	1 test sequence (with subparts)	100% state and transition coverage. Guarantees to find output faults.
HSI	FSM	$O(p^{\theta+1} \cdot n^3)$	$O(p^{\theta+1} \cdot n^3)$	Structured test suite with multiple test sequences	Guarantees to discover output and transfer faults and θ extra states.
H	FSM	$O(p^{\theta+1} \cdot n^3)$	$O(p^{\theta+1} \cdot n^3)$	Structured test suite with multiple test sequences	Guarantees to discover output and transfer faults and θ extra states. Improvement of the HSI-method
N-SC	FSM	$O((N+1) \cdot m^{(k+1)(N+1)}, k = 0..N$	$O((N+1) \cdot m^{N+1})$	1 test sequence	Covers all topologically possible, consecutive $N+1$ transitions.

B. All-State (AS)

The All-State (AS) test generation method produces a test sequence that visits every state of a deterministic, strongly connected[15] FSM at least once. It applies the Nearest Neighbour (NN) heuristic [15] which searches in each step for the closest unvisited state until such state exists.

The AS test generation has $O(n^2)$ time complexity, the length of the generated test sequence is $O(m)$.

C. Transition Tour (TT)

The Transition Tour (TT) [23] algorithm generates the shortest test sequence that visits all transitions of a deterministic, strongly connected[15] FSM at least once, then returns to the initial state.

The problem of creating the test sequence above was reduced to the Directed Chinese Postman Problem (DCPP) [12] with unit costs for the edges of graph G (where G corresponds to specification machine M). The related algorithm [12], [20], [25] consists of the following two parts:

I. Augmenting the original graph G by duplicating some edges to make it Eulerian graph G_E.
II. Finding an Euler tour over G_E.

The time complexity of TT test generation and the length of the resulting test sequence is $O(n^3 + m)$ and $O(m)$, respectively. The generated test sequence guarantees to discover all output faults, but does not guarantee to find transfer faults.

D. All-Transition-State (ATS)

The All-Transition-State (ATS) algorithm [31] creates a test suite for deterministic, strongly connected[15] FSMs that fulfills the first two formal conditions of the ATS criteria [13]:

I. For all t transitions: The test suite should cover at least one walk that contains t and then reaches all states of M.
II. There has to be at least one walk to all states which does not include transition t (if feasible).

The ATS algorithm uses a preamble part responsible for traversing all transitions of FSM M first, then a postamble part responsible for traversing all states of M to fulfill both conditions, but on different graphs. These different graphs include the state transition graph of the specification FSM M and its subgraphs, where some t transitions are filtered out. The preamble part is realized using the TT algorithm without returning to the initial state at the end and the postamble part applies the NN heuristic [15] which searches in each step for the closest unvisited state until such state exists.

There are 3 different versions of the ATS algorithm:

1. Standard version (ATS0),
2. Iterative version without iteration limit (ATSa),
3. Iterative version with iteration limit (ATSx).

The three versions of the algorithm differ in how condition II can be fulfilled. The user has the choice to find a trade-off between coverage and the overall length of the test suite. ATS0 has a total complexity of $O(n^3 + m)$ and an $O(m)$ overall length for the test suite [31]. ATSa and ATSx have a total complexity of $O(\eta(n^3 + m))$, where $\eta < 2 \cdot n$ and the total length of the resulting test suite is $O(\eta \cdot m)$ [31]. The generated test sequence guarantees to discover all output faults and to find most of transfer faults [31].

E. All-Transition-Transition (ATT)

This algorithm is the naive implementation of the first two conditions of the All-Transition-Transition (ATT) criteria [13]:

I. For all t transitions: The test suite should cover at least one walk that contains t and then reaches all transitions of the FSM.
II. There has to be at least one walk to all transitions which does not include transition t (if feasible).

For condition I, the ATT algorithm uses a preamble part that traverses all transitions of the FSM, then a postamble part that traverses all transitions of the machine again. Condition II can be fulfilled in a similar way, but the preamble part is applied on different filtered graphs of the specification.

The complexity of ATT test generation and the length of the test sequence is $O(m \cdot (n^3 + m))$ and $O(m^2)$, respectively.

F. Harmonized State Identifiers (HSI)

In this algorithm, the Harmonized State Identifiers (HSI) state verification method [21], [27] is applied to create a structured test suite for reduced, deterministic, strongly connected FSMs with reliable reset[16] capability [29]. The algorithm contains the following main parts:

- A *state cover set (SCS)* $Q = \{q_0, \ldots, q_{n-1}\}$ that is used for reaching all states; the problem was reduced to create a spanning tree from the s_0 initial state.

[15]If reset transitions exist, MTR applies them in the test suite only if the strongly connected assumption cannot be fullfilled without them.

[16]If the model has unreliable resets, then MTR generates a distinct test suite first, that checks if all reset transitions are implemented in the SUT properly.

- A *separating family of sequences* of Z responsible for verifying end states. The Z set is a collection of sets $Z_j, j = 0, \ldots, n-1$ of sequences (one set for each state) where for every non-identical pair of states s_j, s_l there exists a separating sequence. In our implementation, the Z set is represented with a spanning forest over an auxiliary state pair graph, the edges of which are directed to state pairs that have a separating input.

Based on the parts discussed above, the algorithm consists of two stages. The first stage identifies all states of the FSM and the second stage checks all remaining transitions. The resulting algorithm is the generalization of the W [10] and Wp [14] methods and it guarantees to find all output and transfer faults of FSM $Impl$.

The total length of the resulting test suite and the complexity of test generation is $O(p \cdot n^3)$ [29]. By extending the method above it will also guarantee to find θ given number of extra states in the implementation, resulting $O(p^{\theta+1} \cdot n^3)$ complexity.

G. H-method

The H-method [11] creates a test suite for reduced, deterministic, strongly connected FSMs with reliable reset[16]. The resulting test suite guarantees to discover all output and transfer faults and preset θ number of extra states in $Impl$.

Just like the HSI-method, the H-method also uses a Q SCS to travel to states that need to be verified. It also uses Harmonized State Identifiers for state identification and transition checking, but instead of using predetermined state identifiers, it selects the appropriate ones on-the-fly, thus shortening the test suite.

The algorithm consists of 4 stages:

- STAGE 1: Let the test suite be the SCS sequences, extended by every possible $\theta + 1$ long permutation of the input symbols.
- STAGE 2: For each two sequences u and v of the SCS Q, check if the test suite has sequences uw and vw such that w distinguishes the states $\delta(s_0, u)$ and $\delta(s_0, v)$. If there are no such sequences, select an appropriate w and add uw and vw to the test suite.
- STAGE 3: For each sequence $u\alpha$ where u is in the SCS Q, and α is a sequence of the input symbols with a length up to $\theta+1$, and a v sequence which is also in the SCS Q, check if the test suite has sequences $u\alpha w$ and vw such that w distinguishes the states $\delta(s_0, u\alpha)$ and $\delta(s_0, v)$. If there are no such sequences, select an appropriate w and add $u\alpha w$ and vw to the test suite.
- STAGE 4 (if $\theta > 0$): For each sequence $u\alpha$ where u is in the SCS Q, and α is a sequence of the input symbols with a length up to $\theta + 1$, and for each $u\beta$ where β is a prefix of α, check if the test suite has sequences $u\alpha w$ and $u\beta w$ such that w distinguishes the states $\delta(s_0, u\alpha)$ and $\delta(s_0, u\beta)$. If there are no such sequences, select an appropriate w and add $u\alpha w$ and $u\beta w$ to the test suite.

The complexity of the test generation and the resulting test suite is $O(p^{\theta+1} \cdot n^3)$ [28].

Note that although the original paper [11] mentioned that the length of the test suite depends on the order in which transitions are checked, no corresponding method is described for this. MTR proposes different strategies for processing transitions. We found that the most effective solution is that when the transitions are sorted by input symbols that produce the most diverse output symbols (i.e. the algorithm prefers input symbols in processing that are able to separate more states), the results presented in Section VI apply this approach.

H. N-Switch Coverage (N-SC)

N-Switch Coverage (N-SC) [10] covers all topologically possible, consecutive $N + 1$ transitions of reduced, deterministic, strongly connected FSMs. Note that article [10] only introduced the criteria that need to be fulfilled, but no corresponding algorithm had been given and to the best of our knowledge there still hasn't been published any.

Thus, we created a new heuristic algorithm for N-SC, that takes N and a k iteration limit (scaling from 0 to N) as input parameters and briefly works as follows:

STEP 1: Initialization:

- Set test sequence ts as empty: $ts := \{\}$.
- Set current state to the initial state: $s_c := s_0$.
- Create an ordered list $L = \zeta_1 ... \zeta_K$ for all possible, consecutive $N+1$ transitions (including loop transitions) in FSM M. Mark all $\zeta_1 ... \zeta_K \in L$ elements as *uncovered*.
- Initialize the next element of L to be covered: $\zeta^n := \{\}$.

STEP 2: Covering $N + 1$ transitions: Repeat until all elements of L are marked as *covered*:

- STEP 2.1: Select an ζ_x element of L that is marked as *uncovered* and for which its s_y start state is the nearest[17] from the s_c current state: $\zeta^n := \zeta_x$.
  - If $s_c \neq s_y$: Add the $s_c \rightarrow s_y$ path into ts. $s_c := s_y$.
- STEP 2.2: Add the next transition $t = (s_j, i, o, s_l)$ of ζ^n to ts. Set s_c to the end state of t: $s_c := s_l$.
- STEP 2.3: Check if ζ^n is covered:
  - If yes:
    - STEP 2.3.1: Mark the element in L corresponding to ζ^n as *covered*.
    - STEP 2.3.2: Check with k iteration limit if a ζ_i element of L marked as *uncovered* is partially covered with the last k steps, i.e. the first k elements of ζ_i is covered with the last k transitions in ts:
      - If yes: $\zeta^n := \zeta_i$ and continue with STEP 2.2.
      - Otherwise: continue with STEP 2.
  - If no:
    - Continue with STEP 2.2

Note that by changing iteration limit k, one can create a trade-off between the length of the resulting test sequence and the test generation time.

List L contains maximum $O(m^{N+1})$ elements each one with a length of $N + 1$, thus the length of the resulting test sequence is $O((N + 1) \cdot m^{N+1})$. STEP 2 iterates over all elements of L and for each element, STEP 2.2 adds test sequences with $N + 1$ length. STEP 2.3.2 checks partially covered elements with $O(m^{k \cdot (N+1)})$ worst case complexity,

[17] This can be found by breadth-first search from s_c.

resulting in $O((N+1)\cdot m^{(k+1)\cdot(N+1)})$ test generation complexity[18].

VI. SIMULATION RESULTS

The simulations were executed on servers running an Ubuntu 22.04.2 LTS operating system with 4 GB memory and one core of a shared AMD EPYC 7763 64-core CPU with 2445 MHz clock frequency.

Strongly connected, reduced, completely specified, deterministic random FSMs with reliable reset capability were generated with MTR in different Scenarios to investigate the performance of the algorithms – see Table II.

TABLE II
INVESTIGATED SCENARIOS

ID	Number of states			Density / $\lvert I\rvert$	$\lvert O\rvert$	simulation goal
	min.	max.	size of step			
Scenario 1	5	2000	5	5	10	complexity
Scenario 2	5	100	5	5	2	fault cov.
Scenario 3	5	100	5	5	10	fault cov.

A. Complexity investigations

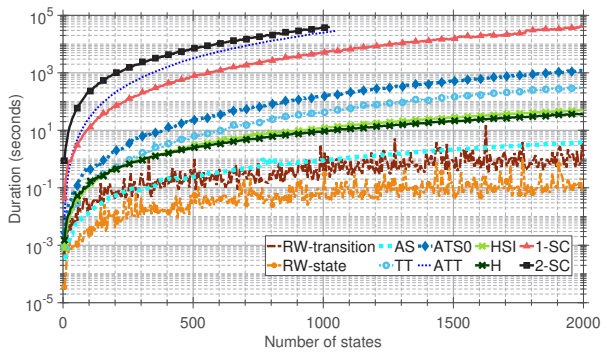


Figure 4. Scenario 1: Test generation time

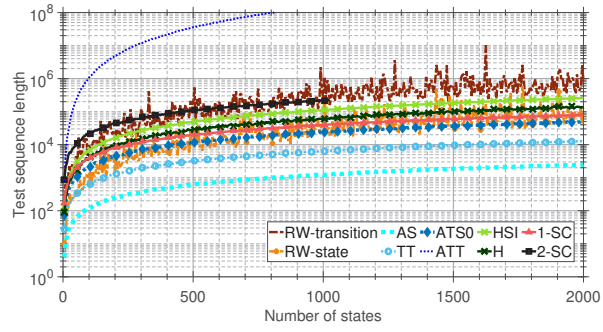


Figure 5. Scenario 1: Test sequence length

[18]In case of completely specified, deterministic FSMs, L contains $O(p^{N+1})$ elements, thus the complexity of the test sequence and test generation is $O((N+1)\cdot p^{N+1})$ and $O((N+1)\cdot p^{(k+1)\cdot(N+1)})$, respectively.

Scenario 1 investigates the time required for test generation and the overall length of the test sequences in function of the number of states in the specification machine.

Figure 4 shows the test generation time for RW (both with 100% state and 100% transition coverage), AS, TT, ATS (with ATS0 standard version), ATT, HSI (with $\theta = 0$), H (with $\theta = 0$), 1-SC (with $k = 1$) and 2-SC (with $k = 2$) algorithms[19]. Figure 5 shows the overall length of the resulting test sequences for the same settings.

As expected, the test generation time of 1-SC, 2-SC and ATT is the longest. The complexity of TT and ATS0 test generation is around the cubic function of the number of the states. The test generation complexity is less than the theoretic cubic upper limit in case of the HSI and the H method because each member of the separating family of sequences typically consists of a test sequence with a length of 1 or 2 instead of the theoretical worst case of $n-1$ length. The H performs better as it is an improvement of the HSI. AS solves a much easier problem to visit all states with NN that is reflected in its complexity. The test generation time of RW is the least as it only selects a transition randomly and checks the stop condition at each step.

The length of the test suite is the shortest in case of AS, that only visit states with NN. The size of the test suite is a linear function of the number of states in case of TT and ATS0. The ones generated by the HSI and H are significantly bigger as in this case the test suite systematically checks all states and verifies the end states of the remaining transitions, although they are much shorter than the theoretic upper limit due to the reason discussed previously. Its size is between 1-SC and 2-SC and the improvement of the H over HSI can be clearly seen. The ATT performs the worst as in each step it tries to create a transition adjacent walk before visiting all transitions.

B. Fault coverage investigations

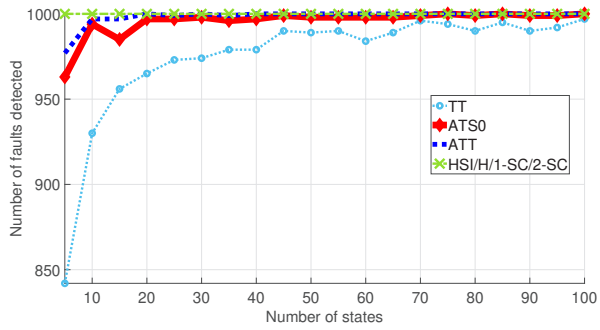


Figure 6. Scenario 2: Number of discovered faults

In Scenarios 2 and 3 the fault coverage of test suites generated by different algorithms is investigated with randomly injected transfer faults[20]. Each data point in these scenarios

[19]ATT runs out memory above 1050 states, 2-SC is investigated only up to 1025 states as its execution time grows rapidly.

[20]Transfer faults are selected for investigations because of the reasons described in Section II-C and the fact that output faults are guaranteed to be found by algorithms that traverse all transitions of the specification FSM.

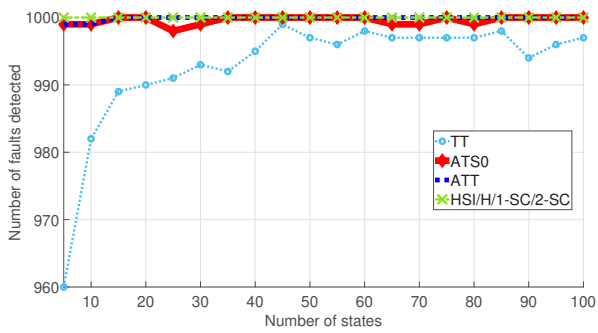


Figure 7. Scenario 3: Number of discovered faults

is obtained by 1000 simulation runs; in each simulation a single transition fault is injected into a distinct random FSM with given parameters and it is observed how many times from these 1000 distinct cases do the algorithms discover the fault. The results for FSMs with 2 and 10 output symbols are presented in Figures 6 and 7, respectively[21]. As expected H, HSI, 1-SC and 2-SC discovered all 1000 transfer faults regardless of the number of states. The ATS0 and ATT algorithms perform just a little worse and TT gives results that can still be acceptable depending on the application domain. In Scenario 3 there are more possible output symbols than in Scenario 2, thus the fault coverage of the test generation algorithms increases, but the trends are similar.

C. SIP UAC registration example

Simulations were also performed to investigate the overall length and the fault coverage of the generated test suites for the specification machine presented in Figure 8 which describes the SIP (Session Initiation Protocol) [1] registration process of the User Agent Client[22].

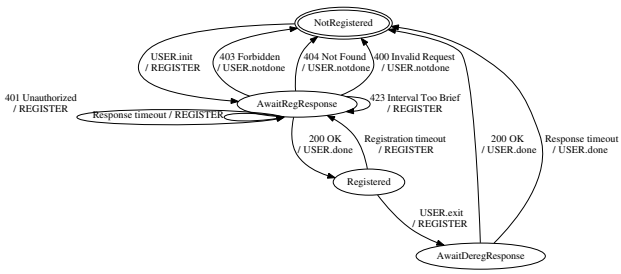


Figure 8. FSM for the registration process of the SIP user agent client

The overall length of the test sequence generated by AS, TT, ATS0, ATT, HSI (with $\theta = 0$), H (with $\theta = 0$), 1-SC (with $k = 1$) and 2-SC (with $k = 2$) is 3, 19, 47, 65, 32, 32, 76 and 372 transitions, respectively.

[21]Note that the results of AS algorithm are not presented in the figures to be able to discover the fault coverage of more robust algorithms precisely. In Scenario 2 AS discovers only 85-184 faults at and below 20 states and 206-256 faults at and above 25 states. In Scenario 3 AS finds 136-222 faults at and below 20 states and 202-261 faults at and above 25 states.

[22]Here only the signaling level was considered; a description about how this FSM can be constructed from the related call-flows is presented in [24].

As the machine contains 4 states and 12 transitions, $12 \cdot (4-1) = 36$ diverse transition faults are possible. Thus, 36 faulty models were created and the fault coverage of the different test suites was investigated. The AS and TT were able to find 6 and 33 faults, respectively. The ATS0, ATT, HSI, H, 1-SC and 2-SC were able to discover all 36 possible faults.

VII. FUTURE WORKS

We have the following plans for future MTR enhancements.

First, we would like to introduce incremental test generation algorithms which identify the effects of changes in the test suite derived from a previous system specification and only update those parts that are necessary. Thus, test generation time can be significantly reduced and different testing goals (such as regression testing, sanity testing) can be clearly isolated from each other.

We also plan to extend our framework to handle Communicating Finite State Machine models and timers which are essential in reliable communication protocols.

As a long term goal, we would like to apply some upper level logic which based on input data – the structure of the specification model, the problem domain, the testing goals and the resources allocated for testing – can automatically propose a test suite that best suits the needs of the test engineer.

VIII. CONCLUSION

In the current article, we introduced a novel model-based testing framework that can be used in the systematic testing of complex software in diverse problem domains such as infocommunications. The framework offers a wide range of model conversion and test generation options.

The test criteria and test coverage can be fine-tuned by selecting a given test generation algorithm and its parameters. The related algorithms were summarized, and a new heuristic test generation algorithm for the N-Switch Coverage Criteria [10] has also been presented. The complexity of test generation and the size of the resulting test suite for the implemented test generation algorithms were investigated via analytical worst case complexity calculations and by empirical analyses. The fault coverage of the generated tests was also observed by simulations. The results let the test engineer find an appropriate trade-off between sources allocated for test execution and the coverage of tests depending on testing goals.

ACKNOWLEDGEMENTS

The first author was supported by the project "Software and Data-Intensive Services" Nr. 2019-1.3.1-KK-2019-00011 financed by the Hungarian National Institute of Science and Innovation.

The authors would like to thank the students who took part in implementation of the following part of the framework: Zsolt Csáky for EFSM handling and EFSM → FSM transformation, Tódor Dávid Nyeste for H and HSI-methods, Tomás Varga for N-SC test generation. The authors would also like to express their gratitude to Levente Hunyadi for his valuable technological advice.

REFERENCES

[1] RFC 3261: SIP: Session Initiation Protocol, 2002. https://tools.ietf.org/html/rfc3261 Accessed: 2023-07-04.

[2] RFC 6749: The OAuth 2.0 Authorization Framework, 2012. https://datatracker.ietf.org/doc/html/rfc6749 Accessed: 2024-01-09.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2nd edition, 2016. DOI: 10.1017/9781316771273.

[4] Cyrille Artho, Martina Seidl, Quentin Gros, Eun-Hye Choi, Takashi Kitamura, Akira Mori, Rudolf Ramler, and Yoriyuki Yamagata. Model-based testing of stateful APIs with Modbat. In *Proc. 30th Int. Conf. on Automated Software Engineering (ASE 2015)*, pages 858–863, Lincoln, USA, 2015. IEEE. DOI: 10.1109/ASE.2015.95.

[5] Fevzi Belli, Christof J. Budnik, Axel Hollmann, Tugkan Tuglular, and W. Eric Wong. Model-based mutation testing—approach and case studies. *Science of Computer Programming*, 120:25–48, 2016. DOI: 10.1016/j.scico.2016.01.003.

[6] Maicon Bernardino, Elder M. Rodrigues, Avelino F. Zorzo, and Luciano Marchezan. Systematic mapping study on MBT: tools and models. *IET Software*, 11(4):141–155, 2017. DOI: 10.1049/iet-sen.2015.0154.

[7] Gregor von Bochmann, Anindya Das, Rachida Dssouli, Martin Dubuc, Abderrazak Ghedamsi, and Gang Luo. Fault Models in Testing. In *Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV*, pages 17–30, Amsterdam, The Netherlands, 1991. North-Holland Publishing Co.

[8] Eckard Bringmann and Andreas Krämer. Model-based testing of automotive systems. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, ICST* '08, pages 485–493, Washington, DC, USA, 2008. IEEE Computer Society. DOI: 10.1109/ICST.2008.45.

[9] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner (Eds.). *Model-Based Testing of Reactive Systems*. Springer, 2005. DOI: 10.1007/b137241.

[10] T. Chow. Testing software design modelled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978. DOI: 10.1109/TSE.1978.231496.

[11] Rita Dorofeeva, Khaled El-Fakih, and Nina Yevtushenko. An Improved Conformance Testing Method. In Farn Wang, editor, *Formal Techniques for Networked and Distributed Systems – FORTE 2005*, volume 3731 of Lecture Notes in Computer Science, pages 204–218. Springer, Berlin, Heidelberg, 2005. DOI: 10.1007/11562436_16.

[12] Jack Edmonds and Ellis L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming,* 5(1):88–124, 1973. DOI: 10.1007/BF01580113.

[13] István Forgács and Attila Kovács. *Practical Test Design*. BCS, The Chartered Institute for IT, 2019.

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

[15] Gregory Z. Gutin, Anders Yeo, and Alexey Zverovich. Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. *Discret. Appl. Math.*, 117(1-3):81–86, 2002. DOI: 10.1016/S0166-218X(01)00195-0.

[16] Drago Hercog. *Protocol Specification and Design. In Communication Protocols*. Springer, Cham, 2020. DOI: 10.1007/978-3-030-50405-2_2.

[17] Gerard J. Holzmann. *Design and Validation of Protocols*. Prentice-Hall, 1990.

[18] David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines – A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996. DOI: 10.1109/5.533956.

[19] Wenbin Li, Franck Le Gall, and Naum Spaseski. A survey on model-based testing tools for test case generation. In *Tools and Methods of Program Analysis*, pages 77–89. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-71734-0_7.

[20] Y. Lin and Y. C. Zhao. A new algorithm for the directed chinese postman problem. *Computers and Operations Research*, 15(6):577–584, 1988. DOI: 10.1016/0305-0548(88)90053-6.

[21] Gang Luo, Alexandre Petrenko, and Gregor V. Bochmann. Selecting Test Sequences for Partially-Specified Nondeterministic *Finite State Machines. In Proceedings of the IFIP WG6.1 7th International Workshop on Protocol Test systems VI*, pages 91–106. Springer, 1995. DOI: 10.1007/978-0-387-34883-4_6.

[22] Matheus Monteiro Mariano, Érica Ferreira de Souza, André Takeshi Endo, and Nandamudi Lankalapalli Vijaykumar. *Comparing graph-based algorithms to generate test cases from finite state machines. Journal of Electronic Testing*, 35(11–12):867–885, December 2019. DOI: 10.1007/s10836-019-05844-6.

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

[24] Gábor Árpád Németh and Péter Sótér. Teaching performance testing. *Teaching Mathematics and Computer Science*, 19(1):17–33, 2021. DOI: 10.5485/TMCS.2021.0518.

[25] S. C. Orloff. A Fundamental Problem in Vehicle Routing. *Networks*, 4:35–64, 1974. DOI: 10.1002/net.3230040105.

[26] Volnei A. Pedroni. *Finite State Machines in Hardware. Theory and Design (with VHDL and SystemVerilog)*. The MIT Press, London, England, 2013. DOI: 10.7551/mitpress/9657.001.0001.

[27] Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, and Anindya Das. Nondeterministic state machines in protocol conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI*, pages 363–378, NLD, 1993. North-Holland Publishing Co.

[28] M. Soucha and K. Bogdanov. SPYH-method: An improvement in testing of finite-state machines. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 194–203, July 2018. DOI: 10.1109/ICSTW.2018.00050.

[29] Mihalis Yannakakis and David Lee. Testing finite state machines: Fault detection. *Journal of Computer and System Sciences*, 50(2):209–227, 1995. DOI: 10.1006/jcss.1995.1019.

[30] Muhammad Nouman Zafar, Wasif Afzal, Eduard Enoiu, Athanasios Stratis, Aitor Arrieta, and Goiuria Sagardui. Model-based testing in practice: An industrial case study using GraphWalker. In *14th Innovations in Software Engineering Conference*, ISEC 2021, New York, NY, USA, 2021. Association for Computing Machinery. DOI: 10.1145/3452383.3452388.

[31] Gábor Árpád Németh and Máté István Lugosi. Test generation algorithm for the All-Transition-State criteria of Finite State Machines. *Infocommunications Journal*, 13(3):56–65, 2021. DOI: 10.36244/ICJ.2021.3.6.

Gábor Árpád Németh obtained his MSc in Electrical Engineering and his PhD in Computer Science at the Budapest University of Technology and Economics (BME), Department of Telecommunication and Media Informatics (TMIT) in 2007 and 2015, respectively. He worked at Ericsson between 2011 and 2018 on a performance testing tool used in the telecommunication industry. Currently, he works at the Eötvös Loránd University (ELTE) on topics related to software testing.

Máté István Lugosi obtained his BSc in Computer Science at Eötvös Loránd University (ELTE) in 2021. Currently, he works at Ericsson on embedded software of microwave network devices. He studies in the MSc program of Computer Science at ELTE in the Cryptography specialization.