Test generation algorithm for the All-Transition-State criteria of Finite State Machines

Gábor Árpád Németh and Máté István Lugosi

***Abstract*—In the current article a novel test generation algorithm is presented for deterministic finite state machine specifications based on the recently introduced All-Transition-State criteria. The size of the resulting test suite and the time required for test suite generation are investigated through analytical and practical analyses and are also compared to the Transition Tour, Harmonized State Identifiers and random walk test generation methods. The fault detection capabilities of the different approaches are also investigated with simulations applying randomly injected transfer faults.**

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

I. INTRODUCTION

Testing plays a vital role in the software development life cycle. The complexity of software is continuously increasing, whereas nowadays the time frame between two releases becomes shorter, raising the probability of faults. Compared to the complexity of the problem, only limited resources are allocated for testing to provide adequate quality for the end product. Although the execution of test cases are automated in most big software companies, test design is typically still done manually, which is a very time consuming process. To cope with this challenge, one can raise the level of automation for the design of test cases as well. 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, such as behaviour trees [8], Finite State Machines (FSMs) [6], [15], [18] and labelled transition systems [6]. This article focuses on FSM formal models, which have been extensively used in diverse areas such as telecommunication software and protocols [13], [14], software related to lexical analysis and pattern matching [3], hardware design [24] and embedded systems [5].

In this article we present a novel test generation algorithm for finite state machine specifications. Our approach is based on the All-Transition-State (ATS) criteria introduced in [10] and uses elements of the Chinese Postman Tour algorithms [9].

The body of the article is organized as follows. Section II discusses related terms regarding graphs, FSMs and conformance testing. The most relevant FSM-based test generation algorithms that are used as a reference point when evaluating our algorithm are also discussed here. Section III introduces our new test generation algorithm for the All-Transition-State criteria, demonstrates it through an example and provides an analysis of its complexity. Section IV presents simulations investigating the test generation time, the overall length of the test sequences and the fault coverage of our algorithm compared to existing methods. The main results of the paper are concluded in Section V with possible future directions.

Gábor Árpád Németh and Máté István Lugosi are with the Department of Computer Algebra, Faculty of Informatics, Eötvös Loránd University, H-1117 Budapest, Pázmány Péter sétány 1/C., Hungary, e-mail: nga@inf.elte.hu, mate.lugosi@gmail.com

DOI: 10.36244/ICJ.2021.3.6

II. PRELIMINARIES

A. Graphs

A *directed graph* is a $G = (V, E)$ (possibly with loop and parallel arcs), where $V = \{s_1, \ldots, s_n\}$ denotes the set of *nodes* and $E = \{e_1, \ldots, e_m\}$ denotes the set of *ordered* pairs of nodes (s_k, s_l) called *directed edges* or *arcs*. In a *weighted directed graph* a number – called *weight* – is assigned to each arc.

A *directed walk* is a finite and alternative sequence of nodes and arcs, $(s_1, e_1, s_2, ...e_{n-1}, s_n)$, where each s_k, s_{k+1} consecutive nodes are the end points of an intermediate edge e_k. A *directed trail* is a directed walk in which all arcs are distinct, a *directed path* is a directed trail in which all nodes are distinct.

A *directed cycle* is a directed trail where the first node is the same as the last node of the sequence. A directed graph is acyclic if it does not contain any directed cycles. A *spanning forest* of a G is an acyclic subgraph of G. A *spanning tree* ST of G is an acyclic subgraph of G which includes all of the nodes of G and exactly $|V| - 1$ arcs which are directed away from root node s_0, so that there is exactly one path from s_0 to any other node. An *inverse spanning tree* TS of G is an acyclic subgraph of G which includes all of the nodes of G and exactly $|V| - 1$ arcs which are directed toward a root node s_0, such a way that from every node $s_k \in V$ there exists exactly one directed path to s_0.

A directed graph is *strongly connected* if there exists a directed path between any two given nodes. The *strongly connected components* (SCCs) of graph G are the maximal strongly connected subgraphs of graph G.

Let the number of arcs originating from node s_j be denoted by $deg^+(s_j)$ (outdegree), and the number of arcs that lead to node s_j by $deg^-(s_j)$ (indegree). We say that node s_j is *balanced* iff $deg^-(s_j) = deg^+(s_j)$, otherwise unbalanced. We say that a directed graph is *Eulerian*, if it is strongly connected and balanced for every node.

A *bipartite graph* $G_B = (V^-, V^+, E)$ is a graph whose nodes can be divided into two disjoint and non-empty sets denoted with V^- and V^+ and every edge in E connects a node in V^- to one in V^+. A *matching* in G_B is a $E_m \subseteq E$ subset of its edges, where none of them share the same node. If sets V^- and V^+ cover the same number of nodes, then a *minimum weighted perfect matching* of G_B may exist, that covers all nodes of sets V^- and V^+ and the overall weights of its edges are minimal.

B. Finite State Machines

A Mealy Finite State Machine (abbreviated as 'FSM' in the rest of the article) M is a quadruple $M = (I, O, S, T)$ where I, O, and S are the finite and non-empty sets of *input symbols*, *output symbols* and *states*, respectively. T is the finite and non-empty 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. The number of states, inputs and transitions of an FSM are denoted by $n = |S|$, $p = |I|$ and $m = |T|$, respectively.

An FSM can be represented with a *state transition graph*, which is a directed labelled graph whose nodes and arcs correspond to the states and transitions, respectively. Each arc 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*.

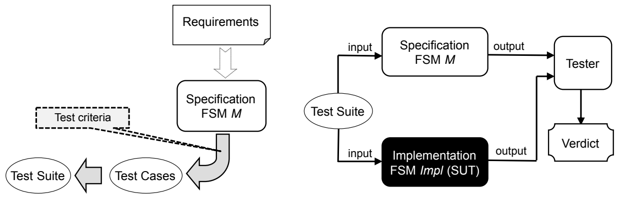
In case of deterministic FSMs the output and the next state of a transition can be given as a function of the start state and the input of a transition, where λ: $S \times I \to O$ denotes the *output function* and δ: $S \times I \to S$ denotes the *next state function*. Let us extend δ and λ from input symbols to finite *input sequences* I^* as follows: for a state s_1, an input sequence $x = i_1, \dots, i_k$ takes the machine successively to states $s_{j+1} = \delta(s_j, i_j)$, $j = 1, \dots, k$ with the final state $\delta(s_1, x) = s_{k+1}$, and produces an *output sequence* $\lambda(s_1, x) = o_1, \dots, o_k$, where $o_j = \lambda(s_j, i_j)$, $j = 1, \dots, k$. The string concatenation operator is denoted by ".".

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.

C. Conformance testing

The structure of FSM model-based test generation is shown in Figure 1(a): A formal specification model denoted by FSM M is derived from the requirements. From FSM M – according to some preset test criteria – *test cases* can be automatically generated; these are the pairs of input sequences



(a) Test generation (b) Conformance testing

Figure 1. Model-based testing

and expected output sequences of M. A set of test cases form a *test suite*. This test suite then can be applied to the System Under Test (SUT) that can be considered as an $Impl$ implementation machine of specification M – see Figure 1(b). Note that machine $Impl$ can be considered as a black box with unknown internal structure, one can only observe its output responses upon a given input sequence. The role of *conformance testing* is to check if the observed output sequences of $Impl$ are equivalent to the expected results derived from M – i.e. to check if $Impl$ *conforms* to M.

D. FSM Fault Models

FSM fault models describe the assumptions of the test engineer about implementation machine $Impl$ as SUT. A usual approach is that the faults do not increase the number of the states specified in FSM M [15], thus the fault model of [7] and [4] are typically restricted to the following two types of faults [15]:

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.

E. Test generation methods

In the following we discuss relevant FSM-based test generation methods that are used as reference points when comparing the performance of our new algorithm. Note that the Transition Tour is discussed in more detail because its elements are reused in our method.

1) Random walk: Starting from the initial state, in each step a transition leading from the current state is chosen randomly and traversed entering a new state until a given stop condition is fulfilled. Various stop conditions – such as a percentage of input/output symbols, visited states or transitions – can be selected based on testing goals.

Although this approach can be useful for exploratory testing, it is impractical for the functional testing of a large-scale software as the length of the test sequence can be much longer than the optimal solution.

2) Harmonized State Identifiers: The Harmonized State Identifiers (HSI) [17], [25] state verification method can be used to create a structured test suite for reduced, deterministic, strongly connected FSMs with reliable reset capability [28]. The resulting algorithm is the generalization of the W [7] and Wp [11] methods and it guarantees to discover all output and transfer faults of FSM $Impl$. According to simulations of [27] this is the most efficient of the W/Wp/HSI triple.

Each test case of HSI consists of the following parts:

- A *state cover set* $Q = \{q_1, \ldots, q_n\}$ responsible for reaching all states; the problem can be reduced to creating a spanning tree ST from initial state s_0.
- A *separating family of sequences* of Z responsible for verifying end states. The Z set is a collection of sets $Z_i, i = 1, \ldots, n$ of sequences (one set for each state) where for every non-identical pair of states s_i, s_j there exists a separating sequence. The Z set can be represented with a spanning forest over a state pair graph, the arcs of which are directed to state pairs that have a separating input [23].

Based on the parts discussed above, the algorithm consists of two stages, one responsible for identifying all states of the machine and the other for checking all remaining transitions.

The resulting test suite consists of no more than $p \cdot n^2$ test sequences, each one with a length less than $2 \cdot n$ interposed with the reset symbol [28]. Thus, the total length of the resulting test suite and the complexity of test generation is $O(p \cdot n^3)$.

3) Transition Tour: The Transition Tour (TT) [19] algorithm produces a test sequence that visits every transition of a reduced, deterministic, strongly connected specification FSM M at least once and returns to the initial state. This is the shortest tour that provides 100% state- and transition coverage of the specification. It guarantees to discover all output faults, but does not guarantee to find transfer faults.

The problem of generating the TT test sequence can be reduced to the Directed Chinese Postman Problem (DCPP) [9] with unit costs for the arcs of graph G (where G corresponds to FSM M). There are multiple algorithms [9], [16], [22] that solve this problem, typically consisting of two major parts:

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

I. Augmenting the original graph to make it Eulerian: Since the goal is to generate the shortest possible test sequence, minimal additional arcs should be added. This is achieved by finding a *minimum weighted perfect matching* on a *bipartite graph* G_B, with group V^- representing the $deg^+ < deg^-$ *negative balanced* nodes and group V^+ representing the $deg^+ > deg^-$ *positive balanced* ones. The weight on edges represents the shortest directed paths from V^- to V^+ nodes of the original graph G in the following way: As the arcs of G are considered to have unit costs, the lengths of the shortest paths are measured as the number of arcs they contain. Every *unbalanced* node s_k of graph G is represented by $|balance(s_k)| = |deg^+(s_k) - deg^-(s_k)|$ number of nodes in bipartite graph G_B ensuring that the matching will contain exactly $|balance(s_k)|$ number of arcs incident to s_k. Once this is done, the graph G can be augmented into an Eulerian graph G_E by multiplicating the arcs along the paths represented by the edges in G_B used in the matching.

II. Finding an Euler tour: After graph G has been augmented to an Eulerian graph G_E, an Euler tour in G_E can be found. There are two ways to represent Euler-tours: the edge-pairing representation and the next-node representation [9]. Here the latter one is considered, which defines an order of the outgoing arcs for each node and in the l_{th} visit of the node, the l_{th} arc is used in this ordering. Finding an Euler tour in this case can be reduced to the creation of an inverse spanning tree TS [9]. To find an Euler tour over G_E one can start at the initial node and specify an arbitrary order for outgoing arcs from each node with the restriction that the arc that is in TS will be the last in the ordering.

The presented method that produces a TT test sequence has $O(n^3 + m)$ time complexity. The lengths of the resulting test sequences is $O(m)$.

III. ALL-TRANSITION-STATE ALGORITHM

We have created a novel heuristic test generation algorithm for reduced, deterministic, strongly connected FSM models based on the All-Transition-State (ATS) criteria introduced in [10]. Note that the original criteria formulates three formal conditions that the test suite should satisfy, but since the third one is applicable for the guarding condition of the EFSM (Extended Finite State Machine) models, we focus on the following two:

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

The motivation behind applying these conditions are the following: (1) Condition I guarantees to find all output faults (as it covers all transitions of the FSM); (2) Condition II requires arc disjoint sequences for all transitions (if feasible) and both condition I and II require to visit all states after transition traversals; thus conditions I and II together are expected to discover most of the transfer faults (the actual fault coverage is investigated later in Section IV).

Building blocks of test sequences: In a nutshell, our algorithm uses a preamble part responsible for traversing all transitions of the FSM first, and then a postamble part responsible for traversing all states of the FSM to fulfill both conditions, but on different graphs. For condition I the original graph G (that corresponds to FSM M) will be used, for condition II different subgraphs of G can be selected when some t transitions are filtered out.

- *All Transition (AT):* This part specifies that all transitions of the given model should be covered at least once. This can be realized using the TT method without returning to the initial state at the end. Thus once all transitions are covered, the traversal of the resulting Euler tour stops.

- *All State (AS):* This part specifies that all states of a given model should be covered at least once. To find the shortest such sequence one can use a solution to the Traveling Salesperson problem [26], without the need to return to the initial state. Since the TSP problem is an NP hard problem, the Nearest Neighbour (NN) heuristic [12] is selected, which searches in each step for the closest unvisited state until such state exists.

ATS algorithm (high level view): To fulfill condition I, the AT and AS parts are generated in step 1 on graph G, respectively, then concatenated. The resulting sequence is called *main sequence* and it covers all transitions of the FSM and then visits all of its states.

For condition II, the AT and AS parts are created on different filtered subgraphs of G and then concatenated to generate appropriate *alternative sequences*. Then, these alternative sequences are applied one after the other. The standard version of our algorithm (denoted by ATS0) generates 2 alternative sequences in step 2 that are as arc disjoint as possible. Note that as these 2 alternative sequences do not necessarily meet condition II, an optional, iterative part is also presented in step 2.3 to provide additional alternative sequences. This iterative part terminates, if for all t transitions an arc disjoint sequence has been found (where it is feasible; ATSa version) or if a predefined iteration limit is reached (ATSx version). The output of the algorithm is a test suite that is the concatenation of the generated main sequence and the alternative sequences. The different versions of the ATS algorithm are summarized in Table I.

Table I
THE SUMMARY OF DIFFERENT ATS ALGORITHM VERSIONS

ID	notes	input	used graphs	output: test suite
ATS0	standard version	FSM M	original: G filtered: G^T, G^{ACT}	1 main sequence + 2 alternative seqs.
ATSa	version without iteration limit	FSM M	original: G filtered: G^T, G^{ACT}, G^{AC_k}	1 main sequence + max. $2n$ alternative seqs.
ATSx	version with iteration limit	FSM M, $depth$	original: G filtered: G^T, G^{ACT}, G^{AC_k}	1 main sequence + max. $depth + 2$ alternative seqs.

The 3 different versions (ATS0, ATSa, ATSx) of our algorithm allow the test engineer to find an appropriate trade-off between coverage and the length of the entire test suite. Note that after the detailed description a small scale example is presented to show step-by-step, how the algorithm works.

ATS algorithm (standard version, ATS0) :

STEP 1. Use AT and AS to create preamble and postamble subsequences, respectively on graph G. The concatenated preamble.postamble main sequence will guarantee that the test suite covers at least one walk from each transition to every state.

STEP 2. Create alternative sequences by concatenating the AT preamble and AS postamble subsequences generated on different subgraphs of G. To maintain the continuity of the entire sequence, each of the alternative sequences should start from the last state reached by the previous one.

STEP 2.1. For the first alternative sequence take the TS inverse spanning tree used in the Eulerian graph G_E during the execution of the AT part of step 1. Then, extend it with randomly selected shortest paths in G from the root node to each of the leaves of TS using breadth-first-search. This results in a strongly connected subgraph of G called G^T. The Eulerian augmentation of G^T is denoted with G_E^T used in AT preamble sequence generation. Then the postamble part is generated using AS on G^T.

STEP 2.2. For a second alternative sequence apply a filter on G that masks out the transitions belonging to G^T. This will be the complement graph of G^T, called G^{CT}. If G^{CT} is not strongly connected, then some transitions have to be reused from G^T, resulting in a graph G^{ACT}. The number of re-enabled transitions should be minimal in order to maintain the highest level of disjointedness using the following method:

STEP 2.2.1. $G^{ACT} := G^{CT}$. Let c denote the number of SCCs of G^{ACT}. Create a directed graph G_{SCC} with c number of nodes, each representing a distinct SCC of G^{CT}. Also create a $c \times c$ zero matrix A that denotes that each of the nodes of G_{SCC} are isolated at this stage.

STEP 2.2.2. For all i components of G_{SCC} check each outgoing arc of G from the nodes of component i and if it leads to component j (where $j \neq i$) and $A_{i,j} = 0$, then add an arc to G_{SCC} from the node representing component i to the node representing component j. Also set $A_{i,j} := 1$.

STEP 2.2.3. While $c > 1$:

STEP 2.2.4.1. Re-enable a random transition in the filter of G that connects two separate, previously unconnected components of G_{SCC} and add the corresponding arc to the filtered graph G^{ACT}.

STEP 2.2.4.2. Check for cycles in G_{SCC}, using depth-first search, if there is one, then merge the nodes that belong to the cycle into a single node representing a new larger SCC. Similarly, shrink the size of the corresponding A matrix. If h nodes were merged, then $c := c - (h - 1)$.

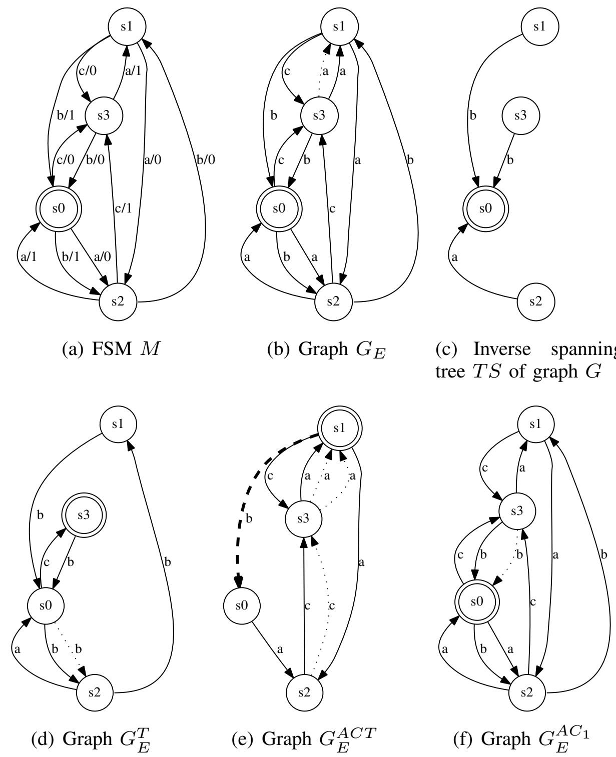
STEP 2.2.4. Once G^{ACT} is strongly connected again, generate preamble and postamble sequences using AT and AS, respectively.

Optional, iterative extensions for ATS (ATSa, ATSx): If transitions had to be re-enabled in step 2.2 to make G^{ACT} strongly connected, the alternative sequences generated for criterion II won't be entirely arc disjoint, i.e. criterion II is not met. In this case the following recursive part of graph filtering can be enabled:

STEP 2.3. The arcs that were both re-enabled in step 2.2

and in all previous iterations of step 2.3 (if there were previous iterations) are collected in the list arc_rem. Then, the arc_rem arcs are filtered out from G resulting in a graph G^{C_k} in the k^{th} iteration. Some of these arcs need to be re-enabled again to connect SCCs (similarly as in step 2.2) resulting in a subgraph G^{AC_k}. These re-enabled arcs remain in arc_rem list, the others are removed. Create an alternative sequence (by concatenating the appropriate AT preamble and AS postamble subsequences) on graph G^{C_k}. Run the function described above recursively until...

- no transitions remain in the list arc_rem or if the number of elements in arc_rem has not decreased since the previous step (ATSa).
- an iteration limit $depth$ is reached or the stop condition of ATSa is met (ATSx).



(a) FSM M (b) Graph G_E (c) Inverse spanning tree TS of graph G

(d) Graph G_E^T (e) Graph G_E^{ACT} (f) Graph $G_E^{AC_1}$

Figure 2. ATS example

ATS example: Here we demonstrate how our ATS algorithm works through a small scale example. We use the following notations in the figures: solid lines represent original arcs (i.e. the transitions of the FSM). Extra arcs, which make the graph balanced, are shown with dotted lines. The re-enabled transitions of filtered graphs that connect SCCs are shown with bold dashed lines. The initial state of each test sequence is denoted with a double circle. The input of each transition is also labeled on its corresponding arc in the graphs.

Consider FSM M in Figure 2(a). From this, an Eulerian graph G_E is created in step 1 – see Figure 2(b). The TS inverse spanning tree of G_E used by the AT part, when creating an Euler tour over G_E is shown in 2(c). The resulting AT input sequence of step 1 is *bbacacabaacb* starting at initial state s_0, followed by the AS input sequence *bbc* finishing at state s_3, forming a main sequence together. The first alternative sequence is created in step 2.1 using the G_E^T Eulerian graph of filtered graph G^T – see Figure 2(d). The resulting AT input sequence is *bbbbbac* starting at state s_3, followed by the AS input sequence *bbb* terminating at state s_1. The second alternative sequence is created in step 2.2 over G_E^{ACT} – see Figure 2(e). The resulting AT input sequence is $acabacac$[1], starting at state s_1, followed by the AS input sequence $aacab$ terminating at state s_0. Here the standard version of the ATS algorithm (ATS0) terminates. Note that arc $s_1 \to s_0$ is re-enabled in G_E^{ACT} to connect two SCCs, i.e. it is used both in the first and the second alternative sequences. Thus, the iterative ATSa extension of the algorithm (described in step 2.3) can be enabled to create an arc disjoint sequence for the $arc\_rem = \{s_1 \to s_0\}$ element. At the first iteration, graph $G_E^{AC_1}$ is created – see Figure 2(f), and $s_1 \to s_0$ is removed from arc_rem. The corresponding AT part *bbacbaacac* starts at state s_0 and is followed by the AS part *aaa*. As $arc\_rem = \{\}$ the algorithm terminates.

ATS complexity calculation:

Standard version (ATS0): The complexity of the AT and AS generation parts are $O(n^3 + m)$ and $O(n^2)$, respectively, due to the TT and the NN algorithms. Thus, step 1 and step 2 require $O(n^3 + m)$ elementary steps, resulting in a total complexity of $O(n^3 + m)$ and in an $O(m)$ overall length for the test suite (in case of deterministic and completely specified FSMs $m = p \cdot n$, resulting in a $O(n(n^2 + p))$ complexity and $O(p \cdot n)$ length of the test suite).

Iterative extensions (ATSx and ATSa): The iterative part requires $O(\eta(n^3 + m))$ additional complexity, where $\eta < 2 \cdot n$ in case of the ATSa and $\eta \leq min(depth, 2 \cdot n)$ in case of the ATSx version, because subgraph G^T of step 2.1 contains no more than $2 \cdot (n-1)$ arcs (the TS inverse spanning tree contains exactly $n-1$ arcs, and the tree that contains the shortest path from the root node to each of the leaves of TS contains no more than $n-1$ arcs) that at worst case need to be filtered out. The total length of the resulting test suite is $O(\eta \cdot m)$.

As our ATS algorithm traverses all transitions of the FSM (AT part of step 1) it guarantees to find all output faults. As the algorithm traverses all transitions, then visits all states (step 1) and also provides alternative sequences that try to be as arc-disjoint as possible, then visit all states (step 2) it is expected to find most of the transfer faults; the actual fault coverage of different ATS algorithm versions (ATS0, ATSa, ATSx) are investigated in the next section.

[1]Note that the second extra multiplication of the $s_3 \to s_1$ arc is not used as all transitions are covered at least once when the algorithm visits state s_3 for the third time, so there is no need to finish the Euler tour with returning to start state s_1.

IV. SIMULATION RESULTS

We implemented our novel ATS algorithm, the random walk with 100% transition coverage stop condition, the TT and the HSI-methods in C++ using the graph algorithms and data structures of the LEMON[2] library.

The simulations were executed on a server running an Ubuntu 18.04.5 LTS operating system with 1 GB memory and one core of a shared Intel Xeon Gold 6140 CPU with 2.30GHz clock frequency.

We generated strongly connected, reduced random FSMs to investigate the performance of the algorithms. The strongly connected property is ensured by first creating a random inverse spanning tree, the arcs of which are directed towards the root node. Then a directed path is built from the root node that visits each of the leaf nodes. Finally, arcs are added between random nodes to reach the desired average outdegree denoted by $\overline{deg^+}$.

Table II
INVESTIGATED SCENARIOS

ID	CS / PS	Number of states: min.	max.	size of step	$\overline{deg^+}$ / $\lvert I\rvert$	$\lvert O\rvert$	simulation goal
Scenario 1	PS	5	2000	5	5	5	complexity
Scenario 2	PS	5	800	5	25	5	complexity
Scenario 3	PS	5	100	5	5	2	fault cov.
Scenario 4	PS	5	100	5	5	5	fault cov.
Scenario 5	CS	5	2000	5	5	5	complexity
Scenario 6	CS	5	800	5	25	5	complexity
Scenario 7	CS	5	100	5	5	2	fault cov.
Scenario 8	CS	5	100	5	5	5	fault cov.

Different scenarios were created both for partially specified (PS) and completely specified (CS) FSMs[3] to investigate the complexity (time required for test generation and the size of the test suite) and the fault coverage of the algorithms – see Table II. In the last subsection the ATS algorithm is investigated on a small-scale telecommunication example.

A. *Partially specified machines*

1) Complexity investigations: Scenarios 1 and 2 examine how the time required for test generation and the overall length of the test sequences are affected by the number of states.

First, consider Scenario 1, where each state of the FSM has 5 transitions in average. Figure 3 shows the test generation time of the Random, TT and the ATS algorithms; the latter one with the standard version (ATS0), with the iterative versions with $depth$ parameters 1 (ATS1) and 2 (ATS2) and without a predefined $depth$ parameter (ATSa). The results indicate that the complexity of the TT and the ATS test generation is around the cubic function of the number of states. The test generation time of the Random algorithm is much less as it only selects a new transition randomly and checks if the stop condition is

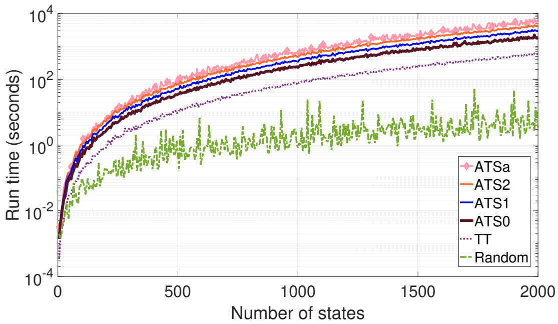


Figure 3. Scenario 1: Test generation time

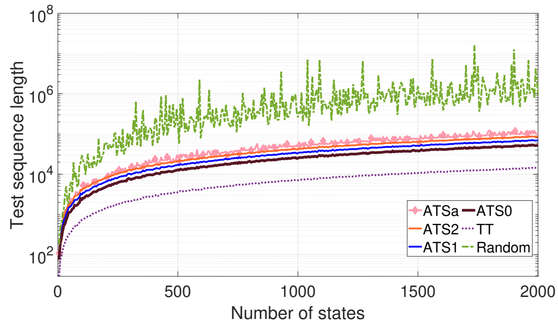


Figure 4. Scenario 1: Test sequence length

fulfilled at each step. Figure 4 shows the overall length of the resulting test sequences. As $\overline{deg^+}$ is fixed, the length of the test sequence is the linear function of the number of states. The length of the test sequence of ATS0, ATS1, ATS2 and ATSa is around 3.5, 4.7, 5.9 and 7 times longer on average as that of the one generated by the TT method, respectively.

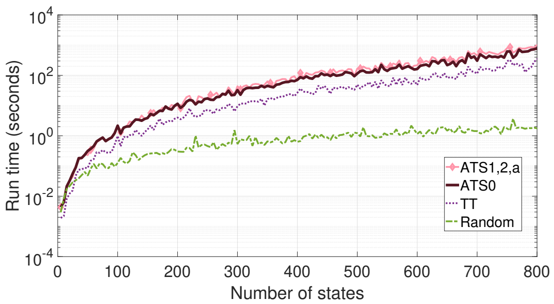


Figure 5. Scenario 2: Test generation time

We also investigate Scenario 2, when 25 transitions on average are set for each state of the FSM. Figures 5 and 6 show the test generation times and the overall lengths of the resulting test sequences, respectively. The trends are similar to the case of Scenario 1, but the complexities are higher due to denser FSMs. Also note that the ATS is able to create completely arc disjoint sequences in all cases even with 1 $depth$ parameter (ATS1) and if the number of states are relatively low, then even the standard version (ATS0) creates completely arc disjoint

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

[3]The motivation behind investigating both PS and CS machines with similar parameters is that the performance of the TT and ATS algorithms is expected to depend on how far each $s_j \in S$ state of the machine is from being balanced; in the latter case $deg^+(s_j) = \overline{deg^+} = |I|$ for all $s_j \in S$.

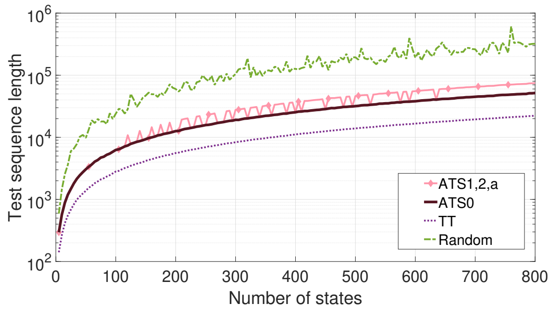


Figure 6. Scenario 2: Test sequence length

sequences. Due to this reason, the iterative extension of the ATS terminate earlier resulting in the same test generation times and lengths of test sequences for different ATS versions (ATS1, ATS2, ATSa). The length of the test sequence of ATS0 and ATS1/2/a is around 2.3 and 2.9 times longer on average as that of the one generated by the TT method, respectively.

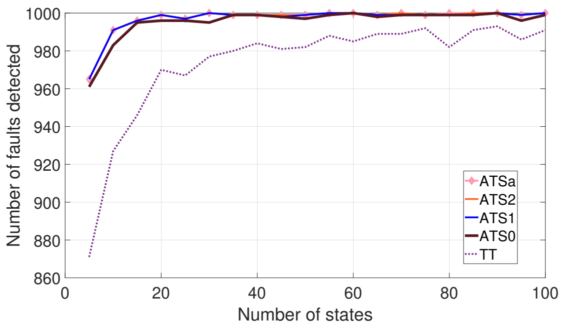


Figure 7. Scenario 3: Number of discovered faults

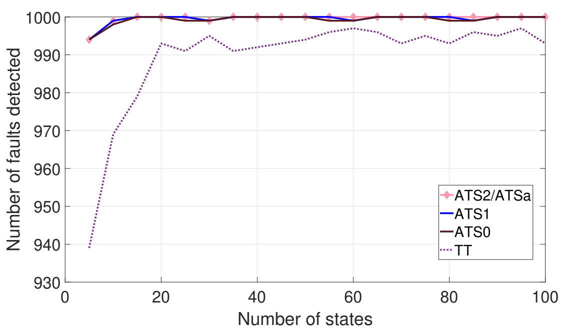


Figure 8. Scenario 4: Number of discovered faults

2) Fault coverage investigation: In Scenario 3 and 4 the fault coverage of different algorithms is investigated with randomly injected transfer faults[4] with 2 and 5 output symbols for the FSMs, respectively. Each data point in the figures had been obtained by 1000 simulation runs; in each simulation a single transition fault is injected to an FSM with given parameters and we observe how many times from these 1000 distinct cases do the algorithms discover the fault.

The results of the TT and the ATS method for Scenario 3 and 4 are presented in Figures 7 and 8, respectively. The results show that the ATS algorithm is much more effective in finding transfer faults than the TT, even with its standard version (ATS0). If the iterative part is switched on and the *depth* parameter increases or is switched off (ATS1 $\rightarrow$ ATS2 $\rightarrow$ ATSa), the fault coverage increases; for all but the smallest machines ATS1, ATS2 and ATSa is able to catch virtually all faults[5]. The relative number of discovered faults increases if the number of states increases both in case of the TT and of the ATS. The reason is that if the size of the test sequence increases, the probability that the desired output and the observed output of the test sequence differs increases. The difference between Scenario 3 and 4 simulations show that the probability of discovering faults increases as the number of output symbols is raised[6]. The reason is that different transitions with more possible output symbols to select from will more probably differ from each other.

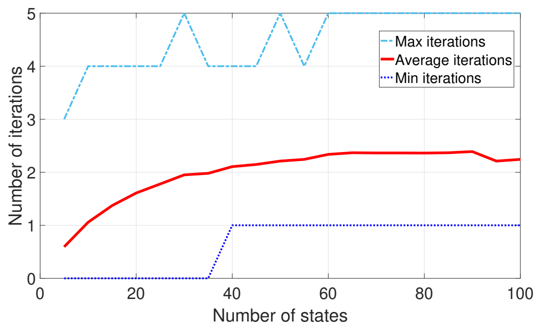


Figure 9. Scenario 3: Number of iterations of ATSa

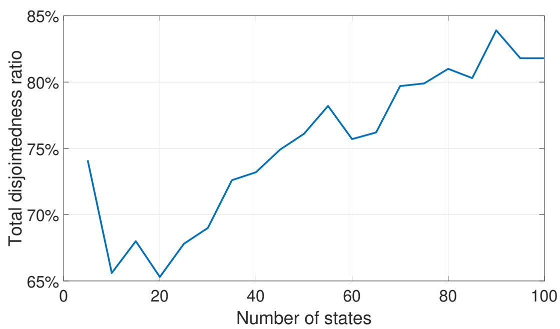


Figure 10. Scenario 3: Ratio of total arc disjoint test suites of ATSa

[4]Note that output faults are not investigated as the TT and the ATS algorithms traverse all transitions of the specification model, thus all of them are able to show both the absence or the presence of single output faults.

[5]Note that in Scenario 3 the fault coverage of ATS0 and ATS1 are almost identical in the performed simulations except at three points ($n = 45, 70, 85$) and that the fault coverage of ATS2 and ATSa only differs at one point ($n = 80$).

[6]Note that in Scenario 4 the fault coverage of ATS1 and ATS2 are almost identical in the performed simulations except at two points ($n = 60, 85$), while the fault coverage of ATS2 and ATSa is identical.

The minimum, the maximum and the average number of iterations for the ATSa algorithm version is also investigated – the results for Scenario 3 are presented in Figure 9[7]. For Scenario 3 Figure 10 presents the ratio when ATSa terminates because for all t transition an arc disjoint sequence has been found (in other cases for some transitions no arc disjoint sequence can be found due to the structure of the FSM)[7].

B. Completely specified machines

Similar scenarios were created for completely specified machines as in case of partially specified ones, but instead of average outdegree, we used the term number of input symbols, as for all states the number of outgoing transitions will be equal with this parameter.

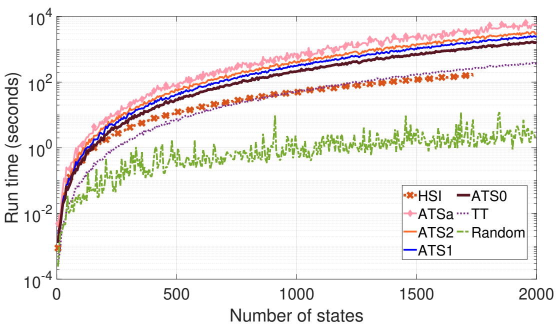


Figure 11. Scenario 5: Test generation time

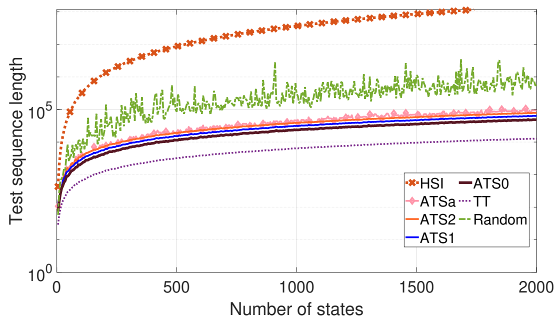


Figure 12. Scenario 5: Test sequence length

1) Complexity investigations: First consider Scenario 5, where the FSMs have 5 input symbols. Figure 11 and 12 show the test generation time and the entire length of the resulting test suite, respectively for the Random, HSI, TT algorithms and for the standard (ATS0) and iterative versions (ATS1, ATS2) of the ATS algorithm.

As in case of partially specified machines, the complexity of the TT and the ATS test generation is the cubic function of the number of states and the length of the TT and the ATS test sequences is the linear function of the number of states. Note that the test generation time of the TT and the different versions of ATS are about 35% and $15-22\%$ less than in case of their partially specified counterpart (Scenario 1), respectively. The reason is that in case of completely specified FSMs, every state has the same number of outgoing transitions, thus less extra arc multiplication is required in the Eulerian graph G_E of FSM M compared to the partially specified FSMs. For the same reason the overall length of the TT and the ATS test sequences are around 11% and 8-9% less in Scenario 5 compared to Scenario 1.

The test generation complexity is less than the theoretic cubic upper limit in case of the HSI method. The reason is that each member of the separating family of sequences typically consists of a test sequence with 1 or 2 length instead of the theoretical worst case $n-1$ length. However, the size of the test suite generated by the HSI is significantly bigger than the ones generated by the TT and the ATS, as this test suite systematically checks all n states and $n \cdot (p-1)$ remaining transitions of the FSM and the verification of a state or the end state of a transition requires $n-1$ distinct sequences.

The test generation time and the entire length of the resulting test suite for FSMs with 25 input symbols are presented in Figure 13 and 14, respectively.

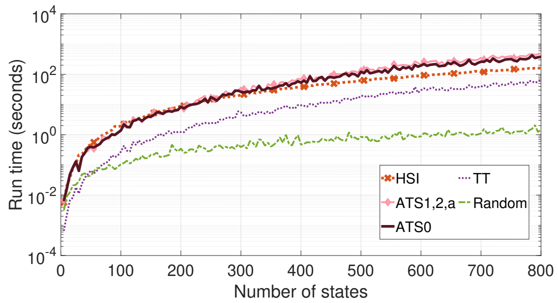


Figure 13. Scenario 6: Test generation time

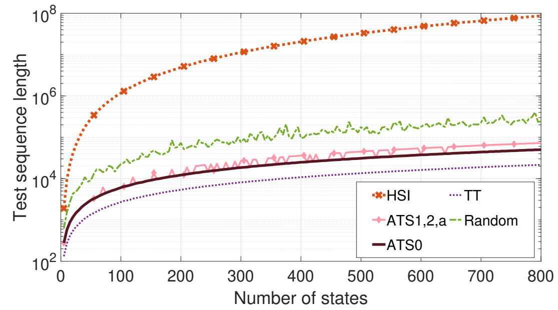


Figure 14. Scenario 6: Test sequence length

2) Fault coverage investigation: The results of the TT, the ATS and the HSI methods for Scenario 7 and 8 are presented in Figures 15 and 16, respectively. As expected, the structured HSI finds all next state faults and the TT-method discovers the least number of faults of the triple. The ATS algorithm is very efficient in discovering faults even with the standard version (ATS0) and it can be further enhanced if the iterative part is switched on (ATS1, ATS2 and ATSa). Note that in

[7]Note that the results are very similar for Scenario 4 as the output symbols of the transitions do not affect the test generation of ATS.

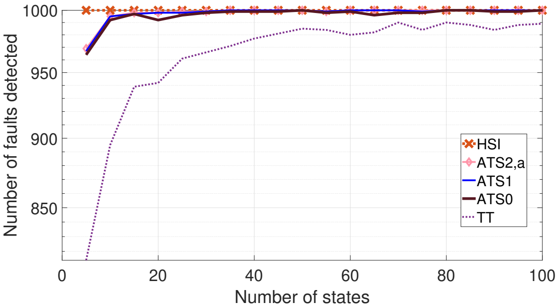


Figure 15. Scenario 7: Number of discovered faults

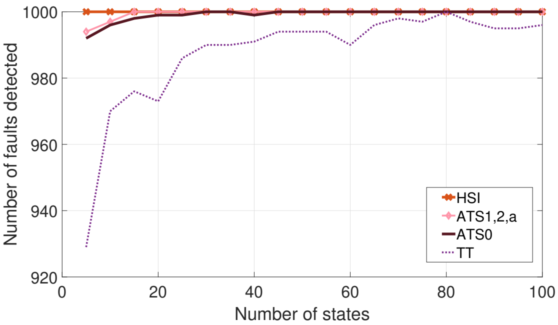


Figure 16. Scenario 8: Number of discovered faults

Scenario 8 ATS1, ATS2 and ATSa provide exactly the same fault coverage in the observed simulations and at and above 15 states they are able to discover all transfer faults as the HSI but with the fraction of its test suite size.

C. SIP UAC registration example

Simulations were also performed to investigate the ATS algorithm with an example from the telecommunication domain. For this, the following functionalities of the User Agent Client (UAC) during the registration process of the SIP (Session Initiation Protocol) [1] over the TCP (Transmission Control Protocol) transport layer were considered:

- Successful new registration (see Section 2.1 of [2])
- Cancellation of registration (see Section 10.2.2 of [1] and Section 2.4 of [2])
- Handle negative responses for registration requests (see Section 10.3 / $4^{th} - 6^{th}$ points of [1]).
- Interval too brief (see Section 10.3 / 7^{th} point of [1])
- Silent discard (see Section 10.2.7 / 7^{th} point of [1])
- Re-registration (see Sections 10.2.1.1 and 10.2.4 of [1])

The resulting FSM is presented in Figure 17. Note that only the signaling level was considered; a detailed description about how this FSM can be constructed from the related call-flows is presented in [21].

The length of the TT test sequence is 19 transitions, the overall length of the sequences generated by ATS0 is 47. Note that ATS0 algorithm can not find arc disjoint alternative sequences for three transitions and due to the structure of the FSM, the iterative version (ATSa) can not find arc disjoint sequences for these transitions, either.

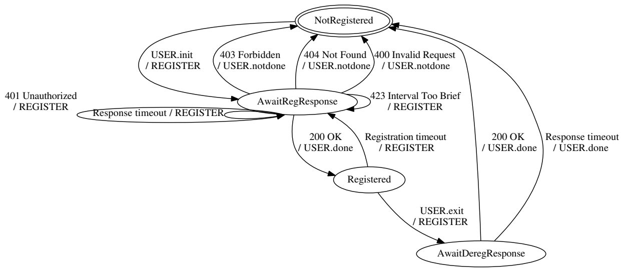


Figure 17. FSMs for the registration process of SIP UAC

As the FSM has 12 transitions and 4 states, $12 \cdot (4-1) = 36$ different atomic transition faults are possible; the corresponding 36 faulty FSMs were created and the fault coverage of TT and ATS was investigated. The TT was able to discover 32 and the ATS0 was able to find 35 faults.

V. CONCLUSION AND FUTURE WORKS

In the current article we proposed a new heuristic algorithm for the All-Transition-State criteria of deterministic finite state machine specifications. The length of the resulting test suite and its fault coverage can be fine-tuned with the three different versions of our algorithm (standard, iterative with and without an iteration limit) allowing the test engineer to find a suitable trade-off between the overall length of the test suite and fault coverage. The simulations show that the size of the resulting test suite has the same order of magnitude as the one produced by the TT-method, while its fault detection capability is near as effective as the one generated by the HSI-method, but with the fraction of its test suite size.

In the future we would like to extend the ATS algorithm to handle changing specifications, i.e. to identify the effects of changes in the test suite derived for a previous system version and to only update those parts that are necessary. As our algorithm reused some fundamental parts of the TT-method, many parts of the incremental TT [20] method can be utilized to fulfill this purpose. We also plan to extend our method for Extended Finite State Machine models, where the guarding conditions over variable values can also be considered when generating the test suite. We also would like to perform an extensive analysis with specification machines of different problem domains.

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).

The authors would like to thank László Dervalics for the implementation of the HSI-method.

REFERENCES

[1] RFC 3261: SIP: Session Initiation Protocol, 2002. https://tools.ietf.org/html/rfc3261 Accessed: 2021-09-02.

[2] RFC 3665: Session Initiation Protocol (SIP) Basic Call Flow Examples, 2003. https://tools.ietf.org/html/rfc3665 Accessed: 2021-09-02.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1st edition, 2008. DOI: 10.1017/CBO9780511809163.

[4] 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.

[5] 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.

[6] 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.

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

[8] R. Geoff Dromey. Formalizing the Transition from Requirements to Design. In Zhiming Liu and Jifeng He, editors, *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, pages 173– 205. World Scientific Series on Component-Based Development, 2006. DOI: 10.1142/9789812772831_0006.

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] Y. Linand 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.

[17] 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.

[18] 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.

[19] 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.

[20] Gábor Árpád Németh and Zoltán Pap. The incremental maintenance of transition tour. *Fundam. Inf.*, 129(3):279–300, July 2014. DOI: 10.3233/FI-2014-972.

[21] 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.

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

[23] Zoltán Pap, Mahadevan Subramaniam, Gábor Kovács, and Gábor Árpád Németh. 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*, TestCom'07/FATES'07, pages 244–259, Berlin, Heidelberg, 2007. Springer-Verlag. DOI: 10.1007/978-3-540-73066-8_17.

[24] 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.

[25] 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.

[26] J. B. Robinson. *On the Hamiltonian game (a traveling-salesman problem)*. RAND Corporation, Santa Monica, CA, 1949.

[27] 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.

[28] 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.

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.