

---

# NOVEL STRATEGY GENERATING VARIABLE-LENGTH STATE MACHINE TEST PATHS \*

---

**Vaclav Rechberger**

Dept. of Computer Science, Faculty of Electrical Engineering,  
Czech Technical University in Prague, Karlovo namesti 13, 121 35, Prague, Czechia  
rechtva1@fel.cvut.cz

**Miroslav Bures**

Dept. of Computer Science, Faculty of Electrical Engineering,  
Czech Technical University in Prague, Karlovo namesti 13, 121 35, Prague, Czechia  
miroslav.bures@fel.cvut.cz

**Bestoun S. Ahmed**

Dept of Mathematics and Computer Science, Karlstad University, 651 88 Karlstad, Sweden  
Dept of Computer Science, FEE, Czech Technical University in Prague, Czechia  
bestoun@kau.se

**Hynek Schvach**

Department of Military Medical Service Organisation and Management  
University of Defence, Trebesska 1575, 500 01, Hradec Kralove, Czechia  
hynek.schvach@unob.cz

## ABSTRACT

Finite State Machine is a popular modeling notation for various systems, especially software and electronic. Test paths can be automatically generated from the system model to test such systems using a suitable algorithm. This paper presents a strategy that generates test paths and allows to start and end test paths only in defined states of the finite state machine. The strategy also simultaneously supports generating test paths only of length in a given range. For this purpose, alternative system models, test coverage criteria, and a set of algorithms are developed. The strategy is compared with the best alternative based on the reduction of the test set generated by the established N-switch coverage approach on a mix of 171 industrial and artificially generated problem instances. The proposed strategy outperforms the compared variant in a smaller number of test path steps. The extent varies with the used test coverage criterion and preferred test path length range from none to two and half fold difference. Moreover, the proposed technique detected up to 30% more simple artificial defects inserted into experimental SUT models per one test step than the compared alternative technique. The proposed strategy is well applicable in situations where a possible test path starts and ends in a state machine needs to be reflected and, concurrently, the length of the test paths has to be in a defined range.

**Keywords** System testing · software testing · internet of things · model-based testing · path-based testing · finite state machines.

---

\* Paper accepted for publication in *International Journal of Software Engineering and Knowledge Engineering*

## 1 Introduction

In system testing, test scenarios are usually defined to describe tests performed on a System Under Test (SUT). The core part of a test scenario is a sequence of actions that must be performed during the test [1]. Such sequences can be created manually by a test analyst based on design documentation of the SUT. They can also be created based on the knowledge of the system's functionality. Such an approach might be ineffective for complex systems and is prone to design defects in test scenarios. The approach may also lead to a set of test scenarios, for which it is not clear the tests cover which parts of the SUT. The test design process can be automated to minimize these drawbacks, subject to the established Model-Based Testing (MBT) discipline [2, 3].

In MBT, part or viewpoint on the SUT is modeled by a suitable notation. Then, given the required test coverage criteria, test paths are generated from the model using dedicated algorithms [4]. A variety of SUT modeling notations can be used within MBT, e.g., [3, 4]. This study focuses on Finite State Machine (FSM) to model parts of the SUT. FSM and its variants are widely used in industry as a modeling notation for system modeling and testing. FSM describes a natural aspect of a wide variety of systems, modules, or data objects processed by a system that is switching from state to state. Therefore, FSMs are one of the fundamental modeling options. Here, FSM-based testing is one of the key testing methods [5].

In real-world projects, budgets and testing time are typically limited [6]. Certain pragmatism and prioritization in covering essential parts of the SUT by more thorough tests and less critical ones by more lightweight tests are highly desirable here. From the test practitioner's viewpoint, it is the foremost approach to achieve the best quality of a SUT given these budget and time constraints. Test coverage criteria are used to determine the thoroughness of the paths [1]. However, determining the suitable coverage criteria is tricky here. In addition, a careful algorithmic design and implementation are needed to generate test paths that meet the designed coverage criteria. To this end, several test coverage criteria and algorithms have been defined and examined for FSMs in the literature. A careful analysis of the literature is given in Section 2.

With the availability of several strategies and studies, the current techniques do not sufficiently reflect the fact that the test paths can effectively start and end only in certain states of the FSM. Attempting to start or end a test in certain states might be highly ineffective or even futile [7, 8]. In addition, the test path length is less explored in the literature, which is more critical from the test practitioner's perspective. Too long test paths are hard to maintain, and if interrupted by a defect in an SUT, it is tricky to test the rest of the flow<sup>2</sup>. Also, too short test paths might be ineffective because of related overhead implied by the execution of the test paths. For example, putting the SUT into an initial state, cleaning procedures after tests, or test reporting effort [9].

Considering the aforementioned gaps in the literature a new strategy will lead to an effort-effective method of FSM testing. To this end, this paper presents a strategy that supports both requirements and compares it with an ad-hoc approach based on an established N-switch test path concept. The contributions of the paper are as follows: (1) Novel strategy that generates FSM test paths concurrently, allowing one to limit their length and express in which states of FSM a test path can start and end is presented, (2) a comparable strategy based on the established N-switch concept is presented, and (3) both strategies are compared using several criteria, including their effectiveness in detecting artificial defects in SUT models, and the results are discussed.

This paper is organized as follows. Section 2 introduces the formal preliminaries used in this work and summarizes related works. Section 3 presents the proposed FSM test path generation strategy, starting with an SUT model and the definition of the test coverage criteria. The algorithms that generate the test paths are presented here, and our proposal can be compared with an alternative test path generation strategy. Section 6 presents the method used in the experiments and their results. Section 7 discussed possible threats to validity and the last section concludes the paper.

## 2 Background and Related Work

One of the common notations of FSM used in MBT is based on a directed graph. The typical model is defined as a directed graph  $\mathcal{G} = (V, E, v_s, V_e)$  such that  $V \neq \emptyset$  is a finite set of vertices representing FSM states and  $E \subseteq N \times N$  is a nonempty set of edges  $e \in E$  representing FSM transitions. Furthermore,  $v_s \in V$  is the start state of the state machine,  $V_e \subset V$  is a set of end states of the state machine [10, 11]. Within this graph, a test path  $t$  is a path in  $\mathcal{G}$ .

Alternatively, part or aspect of the SUT expressed by FSM can be modeled by a Regular Expression (RE) [12]. RE describes FSM so that every possible word (sequence of transitions in the FSM) that fits a pattern defined by RE

<sup>2</sup><https://dzone.com/articles/17-best-tips-to-write-effective-test-cases> or <https://reqtest.com/testing-blog/learn-how-to-write-effective-test-cases/> to give few examples

corresponds to a path in this FSM. We denote RE, defining the model as  $\phi$ . A test path  $t$  is a word allowed by  $\phi$ . In both models,  $T$  denotes a set of test paths.

$T$  satisfies *Node Coverage*, when each  $v \in V \in \mathcal{G}$  is presented in at least one  $t \in T$ . In the literature, the Node Coverage is also alternatively denoted as *All States Coverage*. [1, 13].  $T$  satisfies *Edge Coverage*, when all edges (transitions)  $e \in E \in \mathcal{G}$  are present in at least one  $t \in T$ . This criterion is commonly called *0-Switch coverage* or *All Transitions Coverage* [14, 15].  $T$  also satisfies *Edge-Pair Coverage* when each path consisting of two adjacent edges  $e \in E$  must occur at least once in at least one  $t \in T$  [1, 16]. Edge-Pair Coverage is also mentioned in the literature as *All Transition Pairs Coverage* and *1-Switch Coverage* [14]. Generalized *N-Switch Coverage* is satisfied, when every combinations  $N + 1$  adjacent transitions (edges of  $\mathcal{G}$ ) must occur at least once in a  $t \in T$  [17].

To generate  $T$  from  $\mathcal{G}$  or  $\phi$ , a number of algorithms can be found in the literature. These algorithms differ by the test coverage criteria that are satisfied by the generated  $T$ . They also differ in the effectiveness of the generated  $T$ . Generally, two test sets that satisfy the same test coverage criteria may still differ in a number of test steps or test paths, which affects the overall effectiveness of the testing process. The algorithms were mostly implemented to generate test cases for specific classes of applications.

Devroey *et al.* proposed an algorithm to generate test suites for software product lines using Feature Diagram and Feature Transition System (FTS) as a SUT model [18]. Since FTS is a directed graph, this algorithm can also be applied to solve the problem discussed in this study. The algorithm uses a branch-and-bound approach and uses heuristics for efficient test path search. Instead of a breadth or depth-first search, the algorithm explores the graph using priorities, where the branch is prioritized when it has a higher score. In the algorithm, a score is used describing how this branch will visit many unvisited states, and how test sets generated by this branch will cover many new states. The score is evaluated using an accessibility matrix computed using a modified Warshall algorithm. Instead of distances, matrix cells contain feature expressions used to evaluate products capable of executing a transition, respectively, a set of transitions changing state of a system from one to another.

Another comparable algorithm was implemented by Alava *et. al.* [19]. In their work, an approach is proposed to generate automated tests for *Java Page Flow* web applications. The main input for this process is a directed graph called *Design View (DView)*. DView is a directed graph, where nodes are pages or actions, and edges are links or forwards. The FSM of the page flows and the coverage criteria are obtained from DView. Comparable coverage criteria used in this approach are *All pages* (equivalent to *All Node Coverage*) and *All actions* (equivalent to *All Edge Coverage*). Test cases are generated to ensure the execution of these test paths.

Carvalho and Tsuchiya exploit model checking to generate test paths for SUT parts described as FSMs, as model checkers can generate counterexamples as proof when a model does not satisfy the specification [14]. The tool uses the NuSMV modeling language to define an FSM. The method aims to support *Node Coverage*, *Edge Coverage*, and *Edge-pair Coverage*. The coverage criteria were defined using the NuSMV language. When the SUT model and the test coverage criteria are prepared in NuSMV notation, the test paths are generated using a heuristic algorithm. Using their model, the authors identified this test set generation problem as an NP-hard covering problem.

Another comparable approach was proposed by Liu and Xu [20] to generate a test set for FSM [20]. RFSM is an extended FSM with a special label notation that gives the RFSM the ability to model more details, for example, a number of transitions or different types of node repetitions [21]. This approach employs a Regular Finite State Machine (RFSM) to model the SUT using an MTTool graphical interface. An algorithm is used to transform the RFSM into ERE and then to generate the test paths. The used ERE is a classical regular expression for designing SUT behavior with extended grammar, giving the ability to model the nature of synchronous and concurrent task execution (transitions or sequences of transitions). ERE can be generated from RFSM. The SUT model in RFSM can be created in two ways: using an MTTool graphical editor or text input, using the commands of the author's proprietary R language. The ERE model is parsed into a set of submodels to avoid state space explosion problems during the generation of the test paths. The algorithm generating the test paths accepts this set of submodels and a set of test requirements, where these test requirements are parts of the SUT model or their combinations that the generated test paths must cover.

From the approaches using RE to model SUT behavior, Kilincceker *et al.* proposed a method that consists of the toolchain for the generation of test paths from SUT parts modeled as a regular expression or from an FSM, which is further converted to RE [22]. In this approach, the context table is used during the generation of the test paths. Details can be derived from the toolchain source code available in a GitHub repository [23]. The tool is available freely for further analysis and comparison with newly developed alternatives.

Kilincceker *et al.* also presented an approach for generating test paths using a SUT specified in Hardware Description Language (HDL) language [24]. The work transfers HDL code into the FSM model that gave this approach a more expansive application field. During the generation of the test path, the FSM is further transformed to RE, in particular

the extended RE model proposed by Liu et al. mentioned in [25]. The approach also includes a model minimization process to speed up the generation of test paths. To obtain test paths, RE is parsed into a Syntax tree from which the test paths are finally generated using an algorithm specified in the study.

Fazli and Mohsen proposed the Strongly Connected Component (SCC) for the generation of prime paths and test sets based on them [26]. This method divides a problem of prime paths generation into smaller sub-problems that lead to better time and space efficiency. The input of this method is a Component Flow Graph, and its output is a set of test paths and a set of prime paths that have been covered. In their study, Fazli and Mohsen experimentally compare three approaches for the generation of prime paths for FSM. The results of the comparison showed that SCC performed well in terms of memory consumption and processing time.

Jia *et al.* proposed a method for the generation of whole program paths to satisfy branch coverage [27]. They employ a divide-and-conquer approach to achieve this goal, which makes this method similar to the SCC-based method discussed above. First, the authors generate a base path set (BPS) for each partial function of a Control Flow Graph, which serves as the first part of the SUT model. Then, in this graph, the algorithm identifies function call nodes using the second part of the SUT model, the Function Call Graph. These function call nodes are then joined with the test paths generated for a particular function call. This process is top-down to gradually join all function call nodes with function paths they call. Here, flags are used to mark functions that have actually been traversed. In this approach, recursive function calls are not supported, adding limitations to the method, since the recursive call is a common construct used routinely in programming.

From other alternatives, Klalil and Labiche presented an approach to generate FSM test paths, supported by a tool called STAGE-1 [28]. Their test path generation method supports *Round Trip Coverage*. Besides that, Random, Depth Traversal and Breadth Traversal criteria are discussed as alternatives. More test sets are produced for each given test coverage criterion (worst, best, and average cases) for further analysis.

Despite the fact that FSM testing is the well-established subarea of the system testing discipline, no work we have found so far is directly addressing: (1) the possibility to explicitly set a start and end of a test path in a SUT model and (2) to determine expected length range of the test paths. Regarding the modeling notation, no major rework or model redefinition is needed, and we easily build a SUT model for the proposed strategy by extension of  $\mathcal{G}$  (see Section 3.1 later). Regarding the test coverage criteria to address the goals of this paper, we need to define alternative criteria. The reason is to satisfy the first goal in which we need to neglect irrelevant or infeasible test paths (e.g., paths that are not starting and ending in explicitly given FSM states) to produce an effective set of test paths.

Considering the available algorithms, the majority of the approaches discussed in this Section, unfortunately, assume that a test path can start and end in any state of an FSM and does not provide a sufficient mechanism for expressing the required test path length. The available methods primarily focus on optimizing the test set. The goal is to minimize the number of test paths and steps while still satisfying the given test coverage criteria.

### 3 The Proposed Approach

The **Flexible State Machine Test (FSMT)** strategy is an alternative approach to generate a more effective set of test paths to satisfy alternative test coverage criteria.

FSMT is based on the following adjustments to the traditional *N-switch Coverage* approach: (1) In addition to the start and end of the tested state machine, we also introduced the possible start and end of the test path, and (2) instead of sequences of uniform length implied by the N-switch Coverage criterion, we defined the length range of generated test paths. A consequence of these adjustments is that we need to define alternative test coverage criteria that must be satisfied by a set of test paths. We propose such criteria later in Section 3.2.

#### 3.1 SUT model

We model the SUT as a directed multigraph  $G = (V, E, L, \varepsilon, v_s, V_e, V_{ts}, V_{te})$ , where  $V$  is a set of vertices representing FSM states,  $E$  is a set of edges representing FSM transitions, and  $L$  is a set of edge labels. Edge  $e \in E$  defined by  $\varepsilon : E \rightarrow \{(s, f, l) \mid s, f \in V \wedge l \in L\}$ , where  $s$  is the start vertex of edge  $e$ ,  $f$  is the end vertex of edge  $e$ , and  $l$  is the label of edge  $e$ . Furthermore,  $v_s \in V$  is the start vertex of the state machine,  $V_e \subset V$  is a set of end vertices of the state machine,  $V_{ts} \subset V$  is a set of possible start of test paths,  $V_{te} \subset V$  is a set of possible end of test paths,  $v_s \in V_{ts}$  and  $V_e \subset V_{te}$ . Moreover,  $V_{ts}$  and  $V_{te}$  can have nonempty intersect. During the creation of the SUT model,  $V_{ts}$  and  $V_{te}$  are defined by the test engineer using the design documentation of the SUT or his knowledge and experience with SUT testability.

The test path  $p$  is a path in  $G$  that starts at  $v_{ts} \in V_{ts}$  and ends at  $v_{te} \in V_{te}$ . A test path is a sequence of edges and  $P$  is a set of all test paths. The SUT model  $G$  is an input to the test path generation strategy defined later in Section 3.3, together with the test coverage criteria defined in Section 3.2.

### 3.2 Test coverage criteria

Test coverage criteria serve to determine a level of guarantee, how many possible path combinations would be exercised by the paths present in a  $P$ . For this reason, the test coverage criterion is accepted as an input to an algorithm that generates  $P$ .

We use two test coverage criteria, *FSMT-level-1 Coverage* and *FSMT-level-2 Coverage*, which differ by the number of test path transitions. *FSMT-level-1 Coverage* is designed for lower intensity tests and *FSMT-level-2 Coverage* for higher intensity tests. This, in turn, added more flexibility to select what fits the testing goal in practice.

A set of all test paths  $P$  satisfies *FSMT-level-1 Coverage*, when all the following conditions are satisfied:

1. Each of the test paths  $p \in P$  must start in a vertex from  $V_{ts}$  and end in a vertex from  $V_{te}$ ,
2. each vertex from  $V_{ts}$  must be presented as the start vertex of a  $p \in P$ , and,
3. for each  $p \in P$ ,  $minLength \leq length(p) \leq maxLength$ , where  $length(p)$  is the length of a test path  $p$  in the number of its edges.

In addition to the rules given above, *FSMT-level-1 Coverage* does not provide any additional requirement on how  $V_{ts}$  and  $V_{te}$  must be chained or combined in the test paths. Furthermore, it is not required that all vertices of  $V_e \cup V_{te}$  be present as an end vertex of a  $p \in P$ . Also, *FSMT-level-1 Coverage* in general does not require to visit either the entire  $E \in G$  or even  $V \in G$ . The *FSMT-level-1 Coverage* criterion is designed for lower intensity FSM tests when prioritization is needed for any reason, such as not having enough resources.

In the same way, a set of all test paths  $P$  satisfies *FSMT-level-2 Coverage*, when all the following conditions are satisfied:

1.  $P$  satisfies *FSMT-level-1 Coverage*, and,
2. each  $e \in E \in G$  that can be part of a  $p \in P$  that starts at a vertex from  $\{v_s\} \cup V_{ts}$ , ends in a vertex from  $V_e \cup V_{te}$  and  $minLength \leq length(p) \leq maxLength$ , where  $length(p)$  is the length of  $p$  in the number of its edges, must be present in  $p$ .

In this paper, we use a term *subsume* to indicate that the meeting of a test coverage criterion  $C_1$  subsumes  $C_2$  if each test set that satisfies  $C_1$  will also satisfy  $C_2$ . To this end, *FSMT-level-2 Coverage* subsumes the *FSMT-level-1 Coverage* criterion.

In contrast to *FSMT-level-1 Coverage*, the *FSMT-level-2 Coverage* is designed for more intensive tests when all FSM transitions must be executed during the tests. Still, *FSMT-level-2 Coverage*, in general, does not require visiting neither  $E \in G$  nor  $V \in G$ . Furthermore, a consequence of *FSMT-level-1 Coverage* and *FSMT-level-2 Coverage* that is defined in this way is that for certain  $G$  combined with certain ranges of  $minLength$  and  $maxLength$ ,  $P$  it could not exist. In such a case, the problem can be solved by changing  $minLength$  and  $maxLength$ , or adding more possible  $V_{ts}$  and  $V_{te}$  to  $G$ .

### 3.3 FSMT strategy

The FSMT strategy comprises a few algorithms that aim to generate effective test sets. Here, the strategy is to generate a set of test paths for the SUT model  $G$ , the expected length range of the test path, and the coverage criterion from the options given in 3.2. To determine this test coverage criterion, we use a switch *testCoverage* where its value 1 means *FSMT-level-1 Coverage* and 2 means *FSMT-level-2 Coverage*. For a certain test path length range  $minLength - maxLength$ , it is possible that  $P$  would not meet the given test coverage criteria. In such a case, the test path length range  $minLength - maxLength$  must be adjusted.

The main Algorithm 1 (*GenerateTestPathsFSMT*) accepts the SUT model  $G$  (defined in Section 3.1), minimal length of test paths (denoted as  $minLength$ ), maximum length of test paths (denoted as  $maxLength$ ), and a switch for the test coverage criterion (*testCoverage*, defined in Section 3.2). The algorithm returns a set of test paths  $P$  and a set of uncovered edges  $E_{uncovered}$ . First, the algorithm iterates at all vertices of  $G$  in which a test path can start (denoted as  $V_{ts}$ ) and tries to find the shortest path in the range to a vertex in which a test path can end (denoted as  $V_{te}$ ).  $V_{ts}$  and  $V_{te}$  are part of the SUT model  $G$ , which is given to the Algorithm 1 as input. In this iterating, the

Algorithm 2 (*FindShortestPathInRange*) is used. Edges used in the paths found in this process are considered covered. The uncovered edges, denoted as  $E_{uncovered}$ , are those that are not part of any of these identified paths. After that, if  $testCoverage = 2$ , the algorithm tries to satisfy the *FSMT-level-2* criterion. In this case, it is taking random edges from the set of uncovered edges and tries to find the shortest path that is (1) longer than  $minLength$  (inclusive), and (2) shorter than  $maxLength$  (inclusive), and (3) composing of the maximum number of uncovered edges, using Algorithm 4 (*FindShortestPathInRangeForEdge*).

The Algorithm 2 (*FindShortestPathInRange*) accepts the SUT model  $G$ , minimal length of test paths ( $minLength$ ), maximal length of test paths ( $maxLength$ ),  $testCoverage$  switch, set of uncovered edges ( $E_{uncovered}$ ) and a vertex in which the algorithm starts construction of a test path (denoted as  $v_{ts}$ ). The Output of the Algorithm 2 is a set of test paths  $P$ . At the beginning, the next path to proceed (denoted  $p_{next}$ ) is set to an empty path. The start vertex of the constructed path (denoted as  $v_{last}$ ) is set to  $v_{ts}$  and the queue of paths to process  $Q$  is initiated empty. After that, there is a cycle repeated while  $p_{next}$  is not empty. At the beginning of this cycle, the algorithm tests if the size of  $p_{next}$  is the same as or lower than the maximal length of the test path ( $maxLen$ ). If so, the algorithm checks if the constructed path ends at one of the nodes of  $V_{ts}$ . If this condition is met, the algorithm considers this path a test path to be returned. Otherwise, it will check if the length of the constructed path is less than  $maxLen$ . If this is true, the algorithm creates a new path  $p_{new}$  for each outgoing edge from  $v_{next}$  by concatenating this edge with  $p_{next}$ . This part is done using the Algorithm 3 (*RemoveParallelEdges*). Then  $p_{new}$  is pushed to the queue  $Q$  which contains paths to process. At the end of the cycle, if  $Q$  is not empty, the algorithm pulls another path to be processed from  $Q$ , assigns it to  $p_{next}$  and sets  $v_{next}$  to the last vertex of  $p_{next}$ , otherwise  $p_{next}$  is set to be an empty path. At the end of the cycle, there is no test path composed, so an empty path is returned.

The algorithm 3 (*RemoveParallelEdges*) accepts a set of edges from which parallel edges must be removed (denoted as  $E_{toFilter}$ ), a set of uncovered edges (denoted as  $E_{uncovered}$ ) and  $testCoverage$  switch. The output of this algorithm is a set of edges in which no parallel edges occur, denoted as  $E_{filtered}$ . The algorithm iterates over  $E_{toFilter}$ , and the actual iterated edge is denoted  $e_{unfiltered}$ . The algorithm tries to find a parallel edge  $e_{parallel}$  to the actual edge  $e_{unfiltered}$  in the set  $E_{filtered}$ . If  $e_{parallel}$  does not exist, it will add  $e_{unfiltered}$  to the set  $E_{filtered}$ . Otherwise, if all edges have to be covered (as indicated by  $testCoverage$  switch) and if  $e_{parallel} \notin E_{uncovered}$ , the algorithm swaps  $e_{parallel} \in E_{filtered}$  with the actual edge  $e_{unfiltered}$ . Finally, the algorithm returns set  $E_{filtered}$ .

The algorithm 4 (*FindShortestPathInRangeForEdge*) accepts an edge  $e_{uncovered}$  that must be present on a built test path, the SUT model  $G$ ,  $minLen$ ,  $maxLen$ ,  $testCoverage$  and a set of uncovered edges  $E_{uncovered}$ . The algorithm starts its exploration in  $e_{uncovered}$  and from this edge, it traverses the graph  $G$  forwards (following the edges directions) to an end vertex from  $V_{te}$  of a possible test path. The algorithm then traverses the  $G$  backwards (in reverse direction than the directions of the edges) to a start vertex from  $V_{ts}$  of a possible test path. The goal is to find a test path that starts at a vertex of  $V_{te}$ , ends at a vertex of  $V_{ts}$ , contains  $e_{uncovered}$ , is longer than  $minLen$  inclusive and is shorter than  $maxLen$  inclusive. If no such path exists, an empty path is returned. This exploration is done by the Breadth First Search (BFS) principle simultaneously in both discussed directions and is described in a technical subroutine specified in Algorithm 5 (*FindPathInRangeForEdgeDirected*).

Algorithm 5 takes the next path to be processed, checks whether it is possible to use this path to create the full test path, (in the current or later iteration) and initiates preparation of the next moves that will be processed by this algorithm in the next iterations. The algorithm 5 uses two sub-routines, *EvaluateCandidate*, specified in Algorithm 6 and *PrepareNextMoves*, specified in Algorithm 7.

The Sub-routine *EvaluateCandidate* (Algorithm 6) accepts a semi-test path and evaluates whether this path can be used altogether with an actual found semi-path for construction of the full test path. If it does so, it returns this full test path. Otherwise, it stores the evaluated semi-test path for possible later usage. The Sub-routine *PrepareNextMoves* (Algorithm 7) accepts a path, extends this path with the next step in an appropriate direction, and puts this path in a queue of paths that are stored for further processing.

## 4 N-switch set reduction strategy

To have a comparable alternative to the proposed FSMT, in the initial experiments, we use **N-switch Set Reduction (NSR) strategy**. It is based on the generation of all *N-Switch Coverage* test paths and subsequent filtering of these paths. There are two levels of filtering done:

1. Remove the paths that do not start at a vertex from  $\{v_s\} \cup V_{ts}$  and end at a vertex from  $V_e \cup V_{te}$ , and,
2. remove further duplication in the test paths that can be removed from  $P$ , so that  $P$  still satisfies the test coverage criteria defined in Section 3.2.

---

**Algorithm 1** Generate test paths for SUT model by FSMT strategy

---

**Function:** **GenerateTestPathsFSMT**

**Input:**  $G, minLength, maxLength, testCoverage$

**Output:** Set of test paths  $P$  and set of uncovered edges  $E_{uncovered}$

```

1:  $P \leftarrow \emptyset$ 
2:  $E_{uncovered} \leftarrow E$ 
3: for each  $v_{ts} \in V_{ts}$  do
4:    $p_{new} \leftarrow \mathbf{FindShortestPathInRange}(G, minLength, maxLength, testCoverage, E_{uncovered}, v_{ts})$ 
5:   if  $p_{new}$  is not empty then
6:      $E_{uncovered} \leftarrow E_{uncovered} \setminus \{e \mid e \text{ is present in } p_{new}\}$ 
7:      $P \leftarrow P \cup \{p_{new}\}$ 
8:   if  $testCoverage = 2$  then
9:     while  $E_{uncovered}$  is not empty do
10:       $e_{uncovered} \leftarrow \mathbf{any } e \in E_{uncovered}$ 
11:       $E_{uncovered} \leftarrow E_{uncovered} \setminus \{e_{uncovered}\}$ 
12:       $p_{new} \leftarrow \mathbf{FindShortestPathInRangeForEdge}(e_{uncovered}, G, minLength, maxLength,$ 
13:         $testCoverage, E_{uncovered})$ 
14:      if  $p_{new}$  is not empty then
15:         $E_{uncovered} \leftarrow E_{uncovered} \setminus \{e \mid e \text{ is present in } p_{new}\}$ 
16:         $P \leftarrow P \cup \{p_{new}\}$ 
16: return  $(P, E_{uncovered})$ 

```

▷ an empty set of test paths  
▷ a set of edges uncovered by test paths  
▷ return all found paths and a set of uncovered edges

---



---

**Algorithm 2** Find the shortest path in range

---

**Function:** **FindShortestPathInRange**

**Input:** SUT model  $G, minLength, maxLength, testCoverage, E_{uncovered}, v_{ts}$

**Output:** Path  $p$  ▷ if no path is found then an empty path is returned

```

1:  $p_{next} \leftarrow \text{empty path}$ 
2:  $v_{last} \leftarrow v_{ts}$ 
3:  $Q$  is an empty queue of paths
4: do
5:   if  $|p_{next}| \leq maxLength$  then
6:     if  $(|p_{next}| \geq minLength) \wedge (v_{last} \in V_{te})$  then
7:       return  $p_{next}$ 
8:     if  $p_{next} < maxLength$  then
9:        $E_{outgoing} \leftarrow \text{edges outgoing from } v_{last}$ 
10:       $E_{outgoing} \leftarrow \mathbf{RemoveParallelEdges}(E_{outgoing}, E_{uncovered}, testCoverage)$ 
11:      for each  $e_{outgoing} \in E_{outgoing}$  do
12:         $p_{new} \leftarrow p_{next}$  appended with  $e_{incoming}$  at its end
13:        push  $p_{new}$  to  $Q$ 
14:      if  $Q$  is not empty then
15:         $p_{next} \leftarrow \mathbf{pull}$  from  $Q$ 
16:         $v_{next} \leftarrow \text{the last vertex of } p_{next}$ 
17:      else
18:         $p_{next} \leftarrow \text{empty path}$ 
19:      while  $p_{next}$  is not empty
20: return empty path

```

▷ no path found

---

---

**Algorithm 3** Remove parallel edges
 

---

**Function:** RemoveParallelEdges

**Input:**  $E_{toFilter}$ ,  $E_{uncovered}$ ,  $testCoverage$ 
**Output:** Set of edges  $E_{filtered}$ 

```

1:  $E_{filtered} \leftarrow \emptyset$  ▷ an empty set of edges
2: for each  $e_{unfiltered} \in E_{toFilter}$  do
3:    $e_{parallel} \leftarrow$  an edge from  $E_{filtered}$  parallel to  $e_{unfiltered}$ , if such an edge does not exist,  $e_{parallel} \leftarrow nil$ 
4:   if  $e_{parallel}$  is nil then
5:      $E_{filtered} \leftarrow E_{filtered} \cup \{e_{unfiltered}\}$ 
6:   else if  $testCoverage = 2$  then
7:     if  $e_{parallel} \notin E_{uncovered}$  then
8:        $E_{filtered} \leftarrow (E_{filtered} \setminus \{e_{parallel}\}) \cup \{e_{unfiltered}\}$ 
    
```

---



---

**Algorithm 4** Find the shortest path in the range of the edge
 

---

**Function:** FindShortestPathInRangeForEdge

**Input:**  $e_{uncovered}$ ,  $G$ ,  $minLength$ ,  $maxLength$ ,  $testCoverage$ ,  $E_{uncovered}$ 
**Output:** Path  $p$  ▷ if no path is found then an empty path is returned

```

1:  $E_{map}$  and  $S_{map}$  are empty maps of paths. The key in the map is the length of the path.
2:  $E_{queue}$  and  $S_{queue}$  are empty queues of paths
3:  $p_{end}$  and  $p_{start}$  are paths one edge long created from  $e_{uncovered}$ 
4: push  $p_{end}$  to  $E_{queue}$ 
5: push  $p_{start}$  to  $S_{queue}$ 
6:  $startMin \leftarrow 1$ ,  $endMin \leftarrow 1$ ,  $startMinCount \leftarrow 1$ ,  $endMinCount \leftarrow 1$ 
7:  $startMaxCount \leftarrow 0$ ,  $endMaxCount \leftarrow 0$ 
8: while ( $E_{queue}$  is not empty)  $\vee$  ( $S_{queue}$  is not empty) do
9:   if  $S_{queue}$  is not empty then
10:     $(p, startMin, startMinCount, startMaxCount, S_{queue}, S_{map}) \leftarrow$ 
    FindPathInRangeForEdgeDirected(
     $minLength, maxLength, testCoverage, startMin, endMin, startMinCount, startMaxCount, S_{map}, E_{map}, S_{queue}, V_{ts}, TRUE$ 
    )
11:    if  $p$  is not empty then return  $p$ 
12:   if  $E_{queue}$  is not empty then
13:     $(p, endMin, endMinCount, endMaxCount, E_{queue}, E_{map}) \leftarrow$  FindPathInRangeForEdgeDirected(
     $minLength, maxLength, testCoverage, endMin, startMin, endMinCount, endMaxCount, E_{map}, S_{map}, E_{queue}, V_{te}, FALSE$ 
    )
14:   if  $p$  is not empty then return  $p$ 
15: return empty path
    
```

---

The strategy for the second filtering level differs depending on the test coverage criteria.

The main algorithm 8 (**GenerateTestPathsNSR**) accepts the SUT model  $G$ , minimal length of test paths ( $minLength$ ), maximal length of the test paths ( $maxLength$ ) and a switch for the test coverage criterion ( $testCoverage$ ). The algorithm first generates all paths in  $G$  of length  $N$ , where  $minLength \leq N \leq maxLength$ . This is done by a subroutine described in algorithm 9 (**FindPathsInRange-ForEdgeRecursive**) that iterates over all  $G$  edges. In each iteration, the algorithm generates all possible paths of the required length beginning at the start vertex of the iterated edge. This job is done recursively. In one iteration, Algorithm 9 checks if the path has the required length. If so, it puts it in the set of results  $P$ . After that, the algorithm checks if this path can be extended. If this condition is met, the algorithm calls itself recursively for each outgoing edge of the path the last vertex until this exploration is within the given test path limit ( $minLength$  to  $maxLength$ ).

After all possible paths of length  $N$  are generated, algorithm 10 (**FilterTestPaths**) reduces  $P$  to keep only paths that are valid test paths from the *FSMT-level-1* and *FSMT-level-2* viewpoint - the test path starts in  $\{v_s\} \cup V_{ts}$  and ends at a vertex from  $V_e \cup V_{te}$ . In this phase,  $P$  still contains a lot of duplication in the test path. Therefore, another reduction  $P$  is performed using Algorithm 11 (**ReduceTestPathsSet**). Here, the paths of  $p$  are analyzed if more paths start in a particular vertex from  $\{v_s\} \cup V_{ts}$  and if so, only one of these paths is kept in  $P$ .



---

**Algorithm 5** Find the path in the range for the edge

---

**Function: FindPathInRangeForEdgeDirected**

**Input:**  $minLength, maxLength, testCoverage, processedMin, otherMin,$

$processedMinCount, processedMaxCount, M_{processed}, M_{other}, Q_{processed}, V_{destination}, backward$

**Output:** path  $p$ ,  $processedMin$ ,  $processedMinCount$ ,  $processedMaxCount$ ,  $Q_{processed}$ ,  $M_{processed}$   $\triangleright$  if no path is found then an empty path is returned

```

1:  $p_{processed} \leftarrow$  pull from  $Q_{processed}$ , decrease  $processedMinCount$  by 1
2: if  $|p_{processed}| \leq maxLength$  then
3:   if  $backward$  then
4:      $v_{processed} \leftarrow$  start vertex of path  $p_{processed}$ 
5:   else
6:      $v_{processed} \leftarrow$  end vertex of path  $p_{processed}$ 
7:   if  $v_{processed} \in V_{destination}$  then  $\triangleright$  Destination vertex reached, actual path will be evaluated as a candidate to
   create full test path
8:      $(p_{full}, M_{processed}) \leftarrow$  EvaluateCandidate( $p_{processed}, M_{processed}, M_{other}, backward, otherMin,$ 
 $maxLength$ )
9:     if  $p_{full}$  is not an empty path then
10:      return ( $p_{full}, processedMin, processedMinCount, processedMaxCount, Q_{processed},$ 
 $M_{processed}$ )
11:   if  $(|p_{processed}| + 1 + otherMin) < maxLength$  then
12:      $(Q_{processed}, processedMaxCount) \leftarrow$  PrepareNextMoves( $Q_{processed}, p_{processed}, v_{processed},$ 
 $processedMaxCount$ )
13: if  $processedMinCount = 0$  then
14:    $startMinCount \leftarrow startMaxCount, startMaxCount \leftarrow 0$ 
15:    $processedMin \leftarrow processedMin + 1$ 
16: return (empty path,  $processedMin, processedMinCount, processedMaxCount, Q_{processed}, M_{processed}$ )

```

---



---

**Algorithm 6** Evaluate candidate

---

**Function: EvaluateCandidate**

**Input:**  $p_{processed}, M_{processed}, M_{other}, backward, otherMin, maxLength$

**Output:** path  $p$ ,  $M_{processed}$   $\triangleright$  if no path is found then an empty path is returned

```

1:  $lowerBound \leftarrow \max(0, minLength - |p_{processed}|) + 1$ 
2:  $upperBound \leftarrow maxLength - |p_{processed}| + 1$ 
3: for each  $i \in \{lowerBound, \dots, upperBound\}$  do
4:   if  $M_{other}$  contains key  $i$  then
5:      $p_{other} \leftarrow M_{other}[i]$   $\triangleright$  value for key  $i$ 
6:     if  $backward$  then
7:        $p \leftarrow$  ( $p_{processed}$  without its last edge) appended with  $p_{other}$ 
8:     else
9:        $p \leftarrow$  ( $p_{other}$  without its last edge) appended with  $p_{processed}$ 
10:    return ( $p, M_{processed}$ )
11: if  $(|p_{processed}| + otherMin) \leq maxLength$  then
12:   if  $M_{processed}$  does not contain key  $|p_{processed}|$  then
13:      $M_{processed}[|p_{processed}|] \leftarrow p_{processed}$ 
14:   else if  $(testCoverage = 2) \wedge (M_{processed}[|p_{processed}|]$  contains less edges from  $E_{uncovered}$  than
 $p_{processed})$  then
15:      $M_{processed}[|p_{processed}|] \leftarrow p_{processed}$ 
16: return (empty path,  $M_{processed}$ )

```

---

---

**Algorithm 7** Prepare next moves

---

**Function:** PrepareNextMoves

**Input:**  $Q_{processed}, p_{processed}, v_{processed}, processedMaxCount$

**Output:**  $Q_{processed}, processedMaxCount$  ▷ if no path is found then an empty path is returned

```

1: if backward then
2:    $E_{next} \leftarrow$  all edges incoming to  $v_{processed}$ 
3: else
4:    $E_{next} \leftarrow$  all edges outgoing from  $v_{processed}$ 
5:  $E_{next} \leftarrow$  RemoveParallelEdges(  $E_{next}, E_{uncovered}, testCoverage$  )
6: for each  $e_{next} \in E_{next}$  do
7:   if backward then
8:      $p_{new} \leftarrow$   $e_{next}$  added at the start of  $p_{processed}$ 
9:   else
10:     $p_{new} \leftarrow$   $p_{processed}$  with  $e_{next}$  added at its end
11:   push  $p_{new}$  to  $Q_{processed}$ 
12:    $processedMaxCount \leftarrow processedMaxCount + 1$ 
13: return ( $Q_{processed}, processedMaxCount$ )

```

---



---

**Algorithm 8** Generate test paths for the SUT model by NSR strategy

---

**Function:** GenerateTestPathsNSR

**Input:** SUT model  $G, minLen, maxLen, testCoverage$

**Output:** Set of test paths  $P$

```

1:  $P \leftarrow \emptyset$  ▷ empty set of paths
2: for each  $e \in E$  do
3:    $p \leftarrow$  empty path
4:    $P_{new} \leftarrow$  FindPathsInRangeForEdgeRecursive(  $p, P, e, G, minLen, maxLen$  )
5:    $P \leftarrow P \cup P_{new}$ 
6:  $P \leftarrow$  FilterTestPaths( $P, G$  )
7:  $P \leftarrow$  ReduceTestPaths( $P, G, testCoverage$  )
8: return  $P$ 

```

---



---

**Algorithm 9** Find paths in range for edge

---

**Function:** FindPathsInRangeForEdgeRecursive

**Input:**  $p, P, e, G, minLen, maxLen$

**Output:** Set of test paths  $P$

```

1: if ( $|p| \geq minLen \wedge |p| \leq maxLen$ ) then
2:    $P \leftarrow P \cup \{p\}$ 
3: if  $|p| < maxLen$  then
4:    $p \leftarrow$   $p$  with  $e$  added at its end
5:    $E_{outgoing} \leftarrow$  edges outgoing of  $v$ ,  $v$  is a vertex to which  $e$  is incoming
6:   for each  $e_{outgoing} \in E_{outgoing}$  do
7:      $P_{new} \leftarrow$  FindPathsInRangeForEdgeRecursive( $p, P, e, G, minLen, maxLen$  )
8:      $P \leftarrow P \cup P_{new}$ 
9: return  $P$ 

```

---

---

**Algorithm 10** Filter test paths

---

**Function:** FilterTestPaths

**Input:**  $P, G$

**Output:** Set of test paths  $P_{filtered}$

```

1:  $P_{filtered} \leftarrow \emptyset$  ▷ empty set of paths
2: for each  $p \in P$  do
3:    $v_s \leftarrow$  the first vertex of  $p$ ,  $v_e \leftarrow$  the last vertex of  $p$ 
4:   if ( $v_s \in V_{ts} \wedge v_e \in V_{te}$ ) then
5:      $P_{filtered} \leftarrow P_{filtered} \cup \{p\}$ 
6: return  $P_{filtered}$ 

```

---



---

**Algorithm 11** Reduce test paths set

---

**Function:** ReduceTestPathsSet **Input:**  $P, G, testCoverage$

**Output:** Set of test paths  $P_{filtered}$

```

1:  $P_{reduced} \leftarrow \emptyset$  ▷ empty set of paths
2:  $S_{covered} \leftarrow \emptyset$ , ▷ empty set of vertices
3:  $E_{coveredEdges} \leftarrow \emptyset$  ▷ empty set of edges
4: for each  $p \in P$  do
5:    $v_s \leftarrow$  the first vertex of  $p$ 
6:   if ( $v_s \notin V_{ts} \vee ((testCoverage = 2) \wedge$  an edge of  $p$  is not in  $E_{coveredEdges}$ )) then
7:      $P_{reduced} \leftarrow P_{reduced} \cup \{p\}$ 
8:      $S_{covered} \leftarrow S_{covered} \cup \{v_s\}$ 
9:      $E_{coveredEdges} \leftarrow E_{coveredEdges} \cup \{\text{edges of } p\}$ 
10: return  $P_{reduced}$ 

```

---

## 5 Initial implementation of the proposed strategy

We have implemented the FSMT strategy on the Oxygen experimental MBT platform, developed by our research group [29, 30]. The Oxygen platform is implemented in Java and can be downloaded and run as an executable JAR file. Java 1.8 Standard Development Kit or Java Runtime 1.8 environment is required to be installed on a local machine. The Oxygen platform with FSTM<sup>3</sup> has been released for free public use. Oxygen provides a visual editor to create an SUT model  $G$ . The schema is based on a simplified UML notation for state machines. Since the possible test paths start and end in UML are not available, they are marked by the color filling of a particular state symbol. The FSM states that are in  $V_{ts}$  are marked by a green background, the states in  $V_{te}$  by a red background, and if a state belongs to both  $V_{ts}$  and  $V_{te}$ , yellow coloring is used.

The start and end states of the FSM and its states are dragged to a canvas from the upper panel. If a state belongs to  $V_{ts}$  or  $V_{te}$ , it can be selected by a checkbox on the right panel when a particular state is selected in a diagram. FSM transitions are created by dragging a mouse from one state to another. By default, nodes are marked by letters, and transitions are marked by numbers when first placed on a canvas. These names can be changed in the right panel when a particular object is selected to edit. Other metadata such as state or transition description or test step expected result could be added there as well.

The created FSM can be validated for basic modeling errors such as inaccessible states, missing start, and others. When a schema is valid for FSM, the implemented FSMT strategy can generate the test paths. At this stage, the parameters  $minLength$ ,  $maxLength$ , and  $testCoverage$  are entered into a dialog box. More test sets with different parameters can be generated and are stored in the project tree in the left application panel. From this project tree, the test set can be opened in a separate window and selected test paths can be visualized in the SUT model by a bold line (see Figure 1).

The generated test paths can then be exported in open formats based on XML, CSV, and JSON. The exported files can be easily used by a test management tool that supports a manual testing process or a test automation tool. In the same way, we implemented the NSR strategy, which works as a baseline to compare FSMT within the initial experiments.

---

<sup>3</sup><http://still.felk.cvut.cz/download/oxygen3.zip>

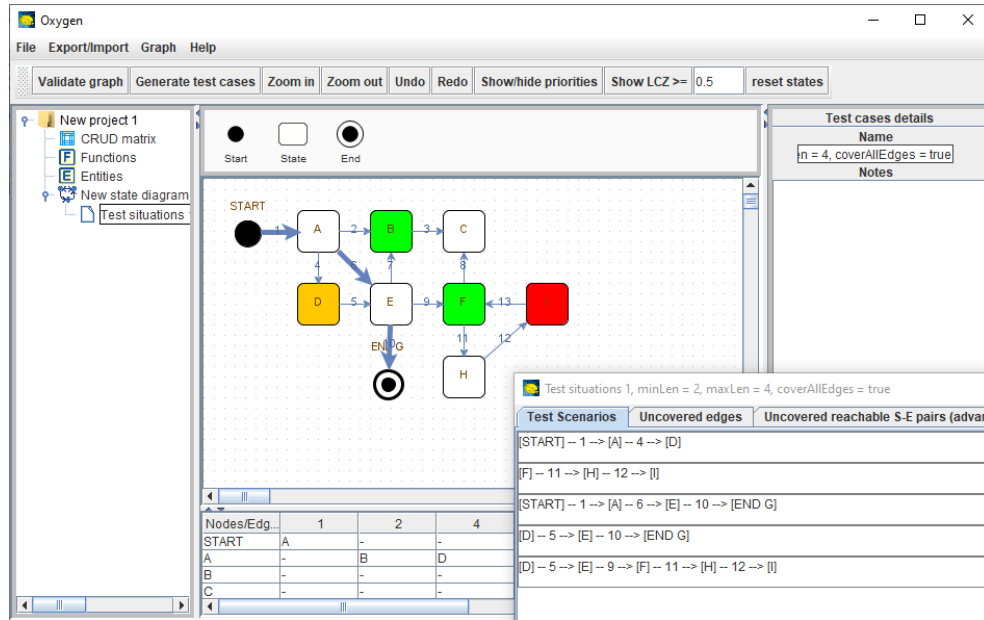


Figure 1: Visualization of generated test paths in the Oxygen platform.

Table 1: Allowed test path length ranges for the experiments. LR stands for Length Range.

LR set ID	<i>minLength</i>	<i>maxLength</i>	variability in test path length
1	2	4	2
2	2	6	4
3	2	8	6
4	4	8	4

## 6 Evaluation Experiments

Our FSMT strategy is unique in concurrently addressing the need for explicitly defined test paths that start and end and address the possibility of specifying the expected length of the test paths. To this end, it is challenging to identify a state-of-the-art strategy that would be completely comparable. So far, the NSR approach presented in Section 4 is the best comparable option for the proposed FSMT strategy. The FSMT has been successfully applied in Skoda Auto car manufacturer to integration and acceptance tests of the produced automobiles. Due to the non-disclosure agreement, we are not allowed to give extensive details; however, industrial FSMs from this project were used as problem instances in the following experiments.

### 6.1 Experiment method and set up

In the following experiments, we used the FSMT and NSR implemented on the Oxygen platform to generate  $P$  for the  $G$  problem instances described in Section 6.2. We run the FSMT and NSR for four sets of length ranges (determined by the intervals *minLength* to *maxLength*), as specified in Table 1.

We analyze the following properties of the generated test paths:

- $|P|$
- $len$  = total length of all  $p \in P$ , measured in the number of edges
- $avlen$  = average length of all  $p \in P$ , measured in number of edges
- $unique$  = number of unique edges in all  $p \in P$
- $ut = \frac{len}{unique}$

The  $ut$  defined above expresses how many non-unique FSM transitions ( $G$  edges) need to repeat in a test path to test all unique transitions. The higher  $ut$  is, the higher this "edge duplication" is in a test set  $P$ .

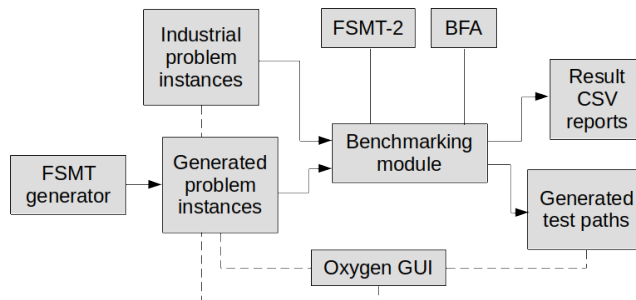


Figure 2: The infrastructure used for the experiments.

We used a benchmarking module that is part of the Oxygen platform. This module allows comparing individual algorithms that compute the test paths for a set of SUT models. The selected set of algorithms run for individual SUT models. The generated test paths are recorded in an Oxygen project. At the same time, the benchmarking module determines the defined properties of the generated test paths and can export them in a special CSV format to allow further analysis and processing of the experimental results. The whole experimental set-up is depicted in Figure 2.

As explained before, in this experiment, we used two types of **problem instances**, anonymized and modified industrial FSMs from Skoda Auto and also artificially generated problem instances, generated by a special **FSMT generator** (details follow in Section 6.2). These problem instances, together with the input parameters *minLength*, *maxLength*, and *testCoverage*, are an input to **benchmarking module**. Two algorithms, FSMT and NSR, are connected to this module. Then, all  $P$  is generated for all problem instances in the input. The benchmarking module produces CSV reports with the properties of  $P$  described in this section and enriches the Oxygen project files with the problem instances by these generated test paths.

## 6.2 The used problem instances

In the experiments, we used a mix of two types of problem instances: modifications of real industrial project state machine models and artificial SUT models generated by a special tool. In the presented results, we used six initial FSM-based models created by Skoda Auto test engineers. These models describe various parts of tested cars<sup>4</sup>. We further modified these models by removing the names of the states and transitions and slight modifications of each FSM to create four different problem instances. These modifications included adding cycles to an FSM, adding possible test starts and test ends, adding or removing a state, and adding and removing a transition. The result was 24 problem instances ( $G$ ) for a set of initial experiments. The selected properties of these instances are presented in Table 2. For individual properties, minimal, maximal, average, and median values are given.

In Table 2, *cycles* denotes the number of  $G$  cycles, *avg cycle length* denotes the average length of these cycles. The *parallel edge groups* denotes the number of groups of parallel edges present in  $G$ , *parallel edges* denotes the total number of parallel edges in  $G$ , and *avg D+* denotes the average node incoming degree. The *avg D-* denotes the average node outgoing degree, and *avg D* denotes the average node degree. By  $|V_{ts} \cap V_{te}|$  we denote states in which a test path can both start and end.

We generated an additional set of problem instances by ModelGen, a specialized module of the Oxygen platform. One of the functionality of this module is the generation of  $G$  problem instances by expected properties of the graph entered as input. These properties include:  $|V|$ ,  $|E|$ , number of  $G$  cycles,  $|V_{ts}|$ ,  $|V_{te}|$ ,  $|V_{ts} \cap V_{te}|$ , and  $|V_e|$ . For the experiments, we generated another 147 problem instances, varying by their properties as given in Table 3. The meaning of the metrics is the same as in Table 2.

In total, we created 171 problem instances as presented in Table 4. We considered this sample extensive enough to carry out the first experiments with the proposed strategy.

## 6.3 Measurement of defect detection potential of test path sets

An important question regarding the effectiveness of the generated  $P$  is its potential to detect possible defects present in a SUT. This potential typically grows with the number of path combinations present in  $P$ , but the exact relation

<sup>4</sup>Due to the confidentiality and Non-disclosure agreement, the types and brands of the tested cars were not mentioned in this paper.

Table 2: Properties of problem instances created from industry project FSMs.

<b>metric</b>	<b>min</b>	<b>max</b>	<b>average</b>	<b>median</b>
$ V $	31	57	40.7	38
$ E $	41	95	64.8	66.5
<i>cycles</i>	0	18	6.5	5.5
<i>avg cycle length</i>	0	22	4.1	5.2
$ V_e $	1	21	8.1	4.5
<i>parallel edges</i>	0	18	4	1
<i>parallel edge groups</i>	0	9	2	0.5
<i>avg D+</i>	1	2	1.6	1.7
<i>avg D-</i>	1	2	1.6	1.7
<i>avg D</i>	2.1	4.1	3.2	3.4
$ V_{ts} $	1	17	7.5	6.5
$ V_{te} $	1	25	8.4	5
$ V_{ts} \cap V_{te} $	0	6	1.5	1.5
<i>SINGLE type defects</i>	6	49	19.6	18
<i>PAIR type defects</i>	5	57	16.7	13
<i><math>e_i</math> to <math>e_a</math> distance</i>	0.6	2.8	1.6	1.6

Table 3: Properties of artificially generated problem instances.

<b>metric</b>	<b>min</b>	<b>max</b>	<b>average</b>	<b>median</b>
$ V $	15	23	17.7	15
$ E $	23	35	31	35
<i>cycles</i>	2	3	2.5	2.5
<i>avg cycle length</i>	4	30.7	10.3	9
$ V_e $	1	1	1	1
<i>parallel edges</i>	0	0	0	0
<i>parallel edge groups</i>	0	0	0	0
<i>avg D+</i>	1.5	2.3	1.8	1.5
<i>avg D-</i>	1.5	2.3	1.8	1.5
<i>avg D</i>	3	4.7	3.6	3.1
$ V_{ts} $	1	2	1.5	1.5
$ V_{te} $	1	2	1.5	1.5
$ V_{ts} \cap V_{te} $	0	2	1	1
<i>SINGLE type defects</i>	2	11	5.8	5
<i>PAIR type defects</i>	1	10	5.3	5
<i><math>e_i</math> to <math>e_a</math> distance</i>	1.2	4	2.3	2.3

is difficult to identify. We added two types of fictional defects into experimental problem instances to evaluate  $P$  produced by the presented FSMT strategy and baseline NSR.

A defect present in an SUT must be activated by a  $p \in P$  to allow its detection by a tester or an automated test. Defect of a SINGLE type is defined at an  $e \in E \in \mathbf{G}$ , and we consider it to be activated when a  $p \in P$  visits  $e$ .

PAIR type defect simulates data consistency defects in an SUT. It is defined as  $(e_i, e_a)$ ,  $e_i, e_a \in E \in \mathbf{G}$ , where there exists a path from  $e_i$  to  $e_a$ . Transition  $e_i$  causes simulated inconsistency of data stored in the SUT and transition  $e_a$  causes its defective behavior. To activate the defect,  $p \in P$  must visit  $e_i$  and then visit  $e_a$ .

The numbers of SINGLE and PAIR type of artificial defects in experimental problem instances are given in Table 4. for all problem instances, in Table 2 for problem instances generated from industry project FSMs and in Table 3 for artificially generated problem instances. In Table 2, 3 and 4,  *$e_i$  to  $e_a$  distance* denotes the number of edges between  $e_i$  and  $e_a$ , averaged for all problem instances.

In the evaluation of  $P$  properties, we further analyze the numbers of activated simulated defects, denoted as  $\mathcal{A}_S$  for the SINGLE type and  $\mathcal{A}_P$  for the PAIR type. Then we measure the average number of simulated defects activated by one test path step, denoted as  $\mathcal{E}_S = \frac{\text{SINGLE activated}}{\text{steps}}$  for the SINGLE type and  $\mathcal{E}_P = \frac{\text{PAIR activated}}{\text{steps}}$  for the PAIR type.

Table 4: Properties of all problem instances used in the experiments.

metric	min	max	average	median
$ V $	15	57	21	15
$ E $	23	95	35.8	35
<i>cycles</i>	0	18	3.1	3
<i>avg cycle length</i>	0	30.7	9.5	8.7
$ V_e $	1	21	2	1
<i>parallel edges</i>	0	18	0.6	0
<i>parallel edge groups</i>	0	9	0.3	0
<i>avg D+</i>	1	2.3	1.8	1.5
<i>avg D-</i>	1	2.3	1.8	1.5
<i>avg D</i>	2.1	4.7	3.5	3.1
$ V_{ts} $	1	17	2.4	2
$ V_{te} $	1	25	2.5	2
$ V_{ts} \cap V_{te} $	0	6	1.1	1
<i>SINGLE type defects</i>	2	49	8	6
<i>PAIR type defects</i>	1	57	7.1	6
<i><math>e_i</math> to <math>e_a</math> distance</i>	0.6	4	2.2	2.3

Table 5: Number of test path sets found for individual length ranges, strategies and test coverage criteria. LR stands for Length Range.

LR set ID	Strategy	Test Coverage	$N_{all}$	$N_{industry}$	$N_{artificial}$
1	FSMT	FSMT-level-1	143	119	24
1	FSMT	FSMT-level-2	143	119	24
1	NSR	FSMT-level-1	138	119	19
1	NSR	FSMT-level-2	138	119	19
2	FSMT	FSMT-level-1	148	124	24
2	FSMT	FSMT-level-2	148	124	24
2	NSR	FSMT-level-1	143	124	19
2	NSR	FSMT-level-2	143	124	19
3	FSMT	FSMT-level-1	152	128	24
3	FSMT	FSMT-level-2	152	128	24
3	NSR	FSMT-level-1	147	128	19
3	NSR	FSMT-level-2	147	128	19
4	FSMT	FSMT-level-1	147	123	24
4	FSMT	FSMT-level-2	147	123	24
4	NSR	FSMT-level-1	142	123	19
4	NSR	FSMT-level-2	142	123	19

## 6.4 Experiment results and discussion

As explained in Section 3.2, for certain  $G$  in combination with certain test set length ranges  $minLength$  to  $maxLength$ ,  $P$  might not exist. This situation can be solved by changing  $minLength$  and  $maxLength$ , or adding more possible  $V_{ts}$  and  $V_{te}$  to  $G$ . However, this effect was present in the experiments and detail of its extent is given in Table 5. For defined test set length ranges (see Table 1), out of total 171 problem instances,  $P$  was returned for 138 up to 152 instances, depending on the strategy and test coverage criterion, denoted as  $N_{all}$  in Table 5. More detail is given separately for industrial ( $N_{industry}$ ) and artificial ( $N_{artificial}$ ) problem instances.

Table 6 shows the experimental results for FSMT and NSR the problem instances summarized in Table 4 and expected test path length ranges as specified in Table 1. In Table 6, the average values for all results are given and  $diff$  is a value for NSR divided by a value for FSMT.

Starting with **FSMT-level-1 Coverage**, FSMT outperformed NSR in parameters  $len$ ,  $|P|$  and  $avlen$  for the four test path length ranges examined. Taking into account  $len$ , the total number of FSM transitions (test steps) in a test set, which is the parameter that gives the closest idea of the effort needed to execute the test paths, the difference between the strategies changed with the test path length range interval. For the test path length range set ID 1 (see Table 1) where  $maxLength - minLength = 2$ , the  $diff$  was 1.6. For length range sets ID 2 and 4, where  $maxLength -$

Table 6: Overall experimental results for FSMT and NSR (averages for all problem instances).

Strategy	$len$	$ P $	$avlen$	$unique$	$ut$	$\mathcal{A}_S$	$\mathcal{A}_P$	$\mathcal{E}_S$	$\mathcal{E}_P$
<b>Length range set 1: <math>minLength = 2, maxLength = 4</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	9.5	3.0	3.2	6.8	1.4	3.55	0.25	0.29	0.017
FSMT	6.0	2.5	2.6	5.1	1.1	2.28	0.04	0.35	0.009
<i>diff</i>	1.6	1.2	1.3	1.3	1.2	1.6	6.0	0.8	2.0
<i>FSMT-level-2 Coverage</i>									
NSR	19.4	5.8	3.3	10.9	1.6	6.15	0.58	0.25	0.023
FSMT	21.3	7.6	2.9	13.6	1.5	7.08	0.56	0.27	0.018
<i>diff</i>	0.9	0.8	1.1	0.8	1.1	0.9	1.0	0.9	1.3
<b>Length range set 2: <math>minLength = 2, maxLength = 6</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	15.1	3.4	4.4	9.2	1.6	4.48	0.65	0.25	0.036
FSMT	7.2	2.7	2.8	6.0	1.2	2.65	0.11	0.35	0.013
<i>diff</i>	2.1	1.3	1.6	1.5	1.4	1.7	5.9	0.7	2.7
<i>FSMT-level-2 Coverage</i>									
NSR	33.4	7.0	4.5	14.2	2.1	7.63	1.40	0.20	0.035
FSMT	32.0	9.7	3.6	17.3	1.8	8.67	0.87	0.23	0.024
<i>diff</i>	1.04	0.7	1.3	0.8	1.2	0.9	1.6	0.9	1.4
<b>Length range set 3: <math>minLength = 2, maxLength = 8</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	20.6	3.6	5.5	10.7	1.9	5.15	1.19	0.23	0.048
FSMT	7.8	2.8	3.0	6.4	1.2	2.71	0.11	0.34	0.014
<i>diff</i>	2.6	1.3	1.8	1.7	1.6	1.9	11.3	0.7	3.5
<i>FSMT-level-2 Coverage</i>									
NSR	45.8	7.5	5.8	16.1	2.6	8.15	1.97	0.17	0.040
FSMT	37.8	10.4	4.0	19.3	1.9	9.21	1.11	0.21	0.029
<i>diff</i>	1.2	0.7	1.4	0.8	1.3	0.9	1.8	0.8	1.4
<b>Length range set 4: <math>minLength = 4, maxLength = 8</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	21.7	3.4	6.2	10.7	2.0	5.37	1.31	0.22	0.052
FSMT	10.9	2.5	4.5	8.0	1.3	3.53	0.36	0.30	0.034
<i>diff</i>	2.0	1.4	1.4	1.3	1.5	1.5	3.6	0.7	1.5
<i>FSMT-level-2 Coverage</i>									
NSR	46.6	7.0	6.4	15.7	2.7	8.30	2.10	0.17	0.042
FSMT	39.2	8.2	5.1	18.2	2.0	9.12	1.53	0.20	0.039
<i>diff</i>	1.2	0.9	1.3	0.9	1.3	0.9	1.4	0.8	1.1

$minLength = 4$ , differences were 2.1 and 2.0. For length range sets ID 3 where  $maxLength - minLength = 6$ , the difference was the largest, 2.6. No such trend is obvious for  $|P|$  in relation to the expected test path length difference. However, as expected, this trend is present for  $avlen$  in the same way for  $len$  (difference increasing from 1.3 to 1.8 with growing  $maxLength - minLength$ ).

The test sets produced by NSR contain more unique FSM transitions, which is a consequence of the fact that these sets contain more transitions in general. Regarding  $ut$ , which measures the extent to how many FSM transitions have to be repeated to test one unique FSM transition, the results for FSMT are better than for NSR. The difference in  $ut$  also increases with  $maxLength - minLength$ . However, not as obviously as in the case of  $len$ .

To give an overall summary, averaged by all test path length ranges, for *FSMT-level-1 Coverage*, FSMT produced test paths with approximately one-half of the total steps than NSR and approximately by 25% lower number of test paths in  $P$ .

Regarding the potential of test path sets to detect artificial defects inserted into SUT models, overall, longer test paths generated by NSR detected more SINGLE and PAIR type defects ( $\mathcal{A}_S$  and  $\mathcal{A}_P$  in Table 6). This is a natural effect, and to evaluate the effectiveness of the set of test paths, the indicators  $\mathcal{E}_S$  and  $\mathcal{E}_P$  must be analyzed. Here, FSMT constantly outperformed NSR in the detection of SINGLE type defects ( $\mathcal{E}_S$ ) for all length range sets, *diff* ranging



Table 7: Overall experimental results for FSMT and NSR for industrial problem instances separately (averages for all problem instances).

Strategy	$len$	$ P $	$avlen$	$unique$	$ut$	$\mathcal{A}_S$	$\mathcal{A}_P$	$\mathcal{E}_S$	$\mathcal{E}_P$
<b>Length range set 1: <math>minLength = 2, maxLength = 4</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	25.6	7.8	3.4	16.9	1.4	10.57	0.91	0.30	0.024
FSMT	18.0	7.7	2.3	14.0	1.2	5.83	0.09	0.36	0.007
<i>diff</i>	1.4	1.0	1.5	1.2	1.2	1.8	10.5	0.8	3.5
<i>FSMT-level-2 Coverage</i>									
NSR	58.6	17.0	3.5	31.0	1.8	20.78	2.09	0.23	0.036
FSMT	79.1	29.8	2.6	50.9	1.5	26.91	2.52	0.27	0.028
<i>diff</i>	0.7	0.6	1.3	0.6	1.1	0.8	0.8	0.8	1.3
<b>Length range set 2: <math>minLength = 2, maxLength = 6</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	40.9	8.5	4.8	22.6	1.7	12.35	1.78	0.25	0.041
FSMT	21.2	8.3	2.4	16.5	1.2	6.96	0.30	0.36	0.015
<i>diff</i>	1.9	1.0	2.0	1.4	1.4	1.8	5.9	0.7	2.8
<i>FSMT-level-2 Coverage</i>									
NSR	92.4	18.2	4.9	36.1	2.4	24.48	4.61	0.17	0.042
FSMT	100.5	34.0	2.9	57.7	1.7	31.26	3.13	0.25	0.029
<i>diff</i>	0.9	0.5	1.7	0.6	1.4	0.8	1.5	0.7	1.5
<b>Length range set 3: <math>minLength = 2, maxLength = 8</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	54.8	8.9	6.2	25.5	2.0	15.48	3.61	0.23	0.055
FSMT	21.8	8.4	2.5	16.6	1.2	7.00	0.17	0.36	0.009
<i>diff</i>	2.5	1.1	2.5	1.5	1.6	2.2	20.8	0.6	5.9
<i>FSMT-level-2 Coverage</i>									
NSR	117.3	17.7	6.4	36.9	3.0	25.09	5.78	0.14	0.041
FSMT	106.7	34.8	3.0	59.2	1.8	32.12	3.35	0.24	0.029
<i>diff</i>	1.1	0.5	2.1	0.6	1.7	0.8	1.7	0.6	1.4
<b>Length range set 4: <math>minLength = 4, maxLength = 8</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	54.6	8.0	6.6	24.3	2.1	15.09	3.61	0.22	0.055
FSMT	29.8	7.1	4.1	20.6	1.3	9.04	0.87	0.30	0.037
<i>diff</i>	1.8	1.1	1.6	1.2	1.6	1.7	4.2	0.7	1.5
<i>FSMT-level-2 Coverage</i>									
NSR	116.5	16.5	6.8	35.5	3.2	24.57	5.78	0.14	0.041
FSMT	109.8	25.3	4.3	53.2	1.9	29.91	4.26	0.21	0.042
<i>diff</i>	1.1	0.6	1.6	0.7	1.6	0.8	1.4	0.6	1.0

from 0.7 to 0.8 (for evaluation of artificial defects, smaller *diff* means better result). This is a significant result - test path sets generated by FSMT detect approximately 20-30% more defects per one test path step than NSR.

On the contrary, NSR outperforms FSMT in effectiveness in detecting PAIR type defects ( $\mathcal{E}_P$ ) and *diff* ranges from 1.5 to 3.5. However, this result has to be interpreted in the context of the number of detected PAIR type defects, which is very low compared to the SINGLE type. The results suggest that the state-machine-based testing technique with test coverage criteria as defined in this study is potentially ineffective for such a type of defect. Considering the results for baseline NSR in this aspect, the question is if a state-machine-based testing technique, in general, is effective in detecting PAIR type defects. However, the answer is beyond the scope of this study. For PAIR type defects, alternative techniques based on the life-cycle of data objects, e.g. the Data Cycle Test (DCyT) [9], are available.

For **FSMT-level-2 Coverage** (which subsumes *FSMT-level-1 Coverage* criterion), the results of FSMT are better than those of NSR. However, the difference is not so significant as in the case of *FSMT-level-1 Coverage*. Regarding *len*, no significant differences are present for test path length range sets ID 1 and 2 having  $maxLength \leq 6$ . But the difference is 1.2 for sets ID 3 and 4 of the length range having  $maxLength = 8$ . Here, for more complex test path generation problems, FSMT starts outperforming NSR.

Table 8: Overall experimental results for FSMT and NSR for artificial problem instances separately (averages for all problem instances).

Strategy	$len$	$ P $	$avlen$	$unique$	$ut$	$\mathcal{A}_S$	$\mathcal{A}_P$	$\mathcal{E}_S$	$\mathcal{E}_P$
<b>Length range set 1: <math>minLength = 2, maxLength = 4</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	6.8	2.2	3.2	5.1	1.3	2.19	0.13	0.28	0.016
FSMT	4.0	1.6	2.6	3.6	1.1	1.60	0.03	0.34	0.009
<i>diff</i>	1.7	1.4	1.2	1.4	1.2	1.4	3.8	0.8	1.7
<i>FSMT-level-2 Coverage</i>									
NSR	12.8	3.9	3.3	7.5	1.6	3.32	0.29	0.25	0.021
FSMT	11.7	3.9	3.0	7.3	1.4	3.25	0.18	0.27	0.016
<i>diff</i>	1.1	1.0	1.1	1.0	1.1	1.0	1.6	0.9	1.3
<b>Length range set 2: <math>minLength = 2, maxLength = 6</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	10.8	2.5	4.4	6.9	1.6	3.02	0.44	0.25	0.035
FSMT	4.9	1.7	2.9	4.3	1.1	1.85	0.07	0.35	0.013
<i>diff</i>	2.2	1.4	1.5	1.6	1.4	1.6	6.0	0.7	2.6
<i>FSMT-level-2 Coverage</i>									
NSR	23.5	5.1	4.5	10.6	2.0	4.50	0.81	0.20	0.033
FSMT	20.6	5.6	3.7	10.6	1.8	4.48	0.45	0.22	0.023
<i>diff</i>	1.1	0.9	1.2	1.0	1.1	1.0	1.8	0.9	1.4
<b>Length range set 3: <math>minLength = 2, maxLength = 8</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	14.9	2.7	5.4	8.2	1.8	3.30	0.76	0.22	0.047
FSMT	5.5	1.8	3.1	4.7	1.1	1.94	0.09	0.33	0.015
<i>diff</i>	2.7	1.5	1.7	1.7	1.6	1.7	8.1	0.7	3.2
<i>FSMT-level-2 Coverage</i>									
NSR	33.9	5.8	5.7	12.6	2.5	5.10	1.28	0.17	0.040
FSMT	26.3	6.3	4.2	12.6	1.9	5.09	0.71	0.20	0.029
<i>diff</i>	1.3	0.9	1.3	1.0	1.3	1.0	1.8	0.8	1.4
<b>Length range set 4: <math>minLength = 4, maxLength = 8</math></b>									
<i>FSMT-level-1 Coverage</i>									
NSR	16.2	2.6	6.2	8.4	2.0	3.55	0.88	0.22	0.052
FSMT	7.7	1.7	4.6	5.9	1.3	2.50	0.27	0.30	0.034
<i>diff</i>	2.1	1.5	1.4	1.4	1.5	1.4	3.3	0.7	1.5
<i>FSMT-level-2 Coverage</i>									
NSR	35.0	5.4	6.3	12.5	2.6	5.26	1.41	0.17	0.043
FSMT	27.4	5.3	5.2	12.4	2.1	5.24	1.02	0.20	0.039
<i>diff</i>	1.3	1.0	1.2	1.0	1.3	1.0	1.4	0.8	1.1

Regarding the number of test paths  $|P|$ , FSMT produces a slightly higher number of test paths, on average 25%. No clear trend is observed in relation to the testing path length range. Consequently, the average length of test paths ( $avlen$ ) is, on average, 30% lower for FSMT.

Taking into account the presence of unique FSM transitions in the test paths measured by  $ut$ , FSMT gives a slightly better result than NSR for  $maxLength = 6$ , where the difference is 1.2, which further increases to 1.3 for  $maxLength = 8$ .

To summarize, for *FSMT-level-2 Coverage*, FSMT produced test paths having approximately 20% fewer steps than the test paths produced by NSR for test path length ranges that have  $maxLength = 8$ . On the contrary, no significant differences are observed for  $maxLength \leq 6$ . Regarding the overall number of these test paths, FSMT produced test sets with approximately 25% more test paths than NSR. Lower  $len$  practically implies lower testing costs, and at this point, this metric is much more significant than  $|P|$ . Hence, for one-half of examined cases ( $maxLength = 8$ ). We can conclude that FSMT outperformed NSR, and for the second half, there is no significant difference between the results of the algorithms. To this end, it is worth noticing that *FSMT-level-2 Coverage* subsumes *FSMT-level-1 Coverage* and is designed for more intense tests.

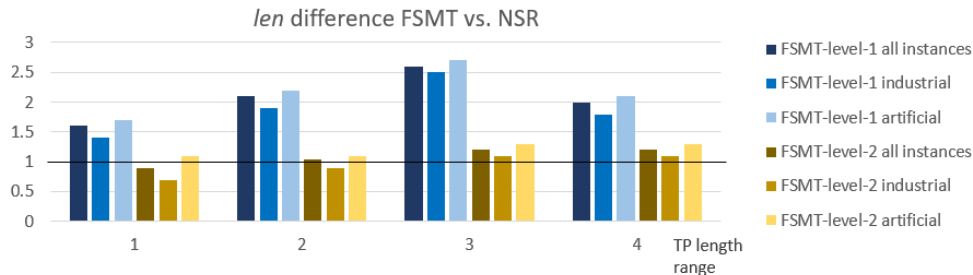


Figure 3: Difference in  $len$  between FSMT and NSR for all problem instances together, then separately for industrial and artificial problem instances.  $FSMT\text{-level-1}$  and  $FSMT\text{-level-2}$  coverage criteria apply to both FSMT and NSR strategies.

For  $FSMT\text{-level-2}$  Coverage, FSMT outperformed NSR in the detection of SINGLE type defects ( $\mathcal{E}_S$ ) for all length range sets,  $diff$  ranging from 0.8 to 0.9, practically meaning that  $P$  generated by FSMT detect approximately 10-20% more defects per one test path step than NSR. NSR outperformed FSMT in effectiveness to detect PAIR type defects ( $\mathcal{E}_P$ ),  $diff$  ranging from 1.1 to 1.4. It is noticeable that this difference is much smaller than in the case of  $FSMT\text{-level-1}$  Coverage criteria, but regarding the low number of PAIR type defects, the result for SINGLE type defect is much more significant.

The presented results showed good performance of the proposed FSMT strategy compared to the NSR strategy. The results show that a strategy such as NSR, based on the generation of all possible  $N\text{-switch}$  Coverage test paths and their subsequent filtering, is not optimal to generate a test set satisfying  $FSMT\text{-level-1}$  and  $FSMT\text{-level-2}$  criteria. Our FSMT is needed to construct the test paths.

Particular data and differences for industrial and artificial problem instances separately are given in Tables 7 and 6. However, the trends in the data are very similar to the overall results discussed in this section for the properties of the set of test paths, as well as their potential to detect artificial defects for both SINGLE and PAIR types.

Overall summary of  $len$ , the main proxy for the testing costs is given in Figure 3. The difference in  $len$  for FSMT and NSR is shown separately for all the four expected test path (TP) length ranges (specified in Table 1) and for all instances of problems together, followed by instances of industrial problems and instances generated artificially.

As we consider the  $len$  as the main indicator used in the experiments, its relation to the length of general test cases and the input parameter  $minLength$  shall be mentioned. As we explained in Section 1, too short test cases are considered suboptimal by test engineers. However, what is "too short" might differ from project to project; hence, we give the engineer the liberty to determine the minimal length of the test paths by the  $minLength$  parameter. This minimal length is part of the test coverage criteria that the generated set of test path  $P$  must satisfy. During the process of  $P$  generation, the proposed FSMT strategy tries to minimize the total length of these test paths ( $len$ ). However,  $P$  must satisfy defined test coverage criteria, so its test paths cannot be shorter than  $minLength$  specified by the test engineer.

Table 9 compares averaged properties of  $P$  generated by FSMT and NSR for the  $FSMT\text{-level-1}$  and  $FSMT\text{-level-2}$  coverage criteria for all problem instances. In Table 9, ratio of averaged value of  $P$  properties for all problem instances for  $FSMT\text{-level-2}$  to this averaged value for  $FSMT\text{-level-1}$  is presented. The last two lines of Table 9 present the average of these differences for all ranges of test path lengths.

For NSR, the difference between  $FSMT\text{-level-2}$  and  $FSMT\text{-level-1}$  is 2.2 on average in  $len$  averaged for all problem instances and 2 in  $|P|$ . For FSMT, this difference is 4.1 for  $len$  and 3.4 for  $|P|$ . These differences have to be interpreted in the context of the average values  $len$  and  $|P|$  for the coverage criteria  $FSMT\text{-level-2}$  and  $FSMT\text{-level-1}$  separately (see Table 6). As NSR produces  $P$  with more test path steps and more test paths in general, and this difference is more significant for  $FSMT\text{-level-1}$  test coverage, this effect is also reflected in the differences presented in Table 9.

There is no significant difference in  $avlen$  for NSR and a slight difference of 1.2 for FSMT. The difference between unique edges on the test paths ( $unique$ ) is 1.5 for NSR and 2.7. for FSMT, which corresponds to the difference for  $len$ . Figure 4 presents the data analyzed for FSMT.

To summarize, test path sets that satisfy  $FSMT\text{-level-2}$  Coverage criterion that subsumes  $FSMT\text{-level-1}$  Coverage criterion generally consist of the approximately two times higher total number of steps in test paths for NSR and approximately four times for FSMT (although, the total number of steps in test path sets generated by FSMT does not exceed this number for NSR; see Table 6).

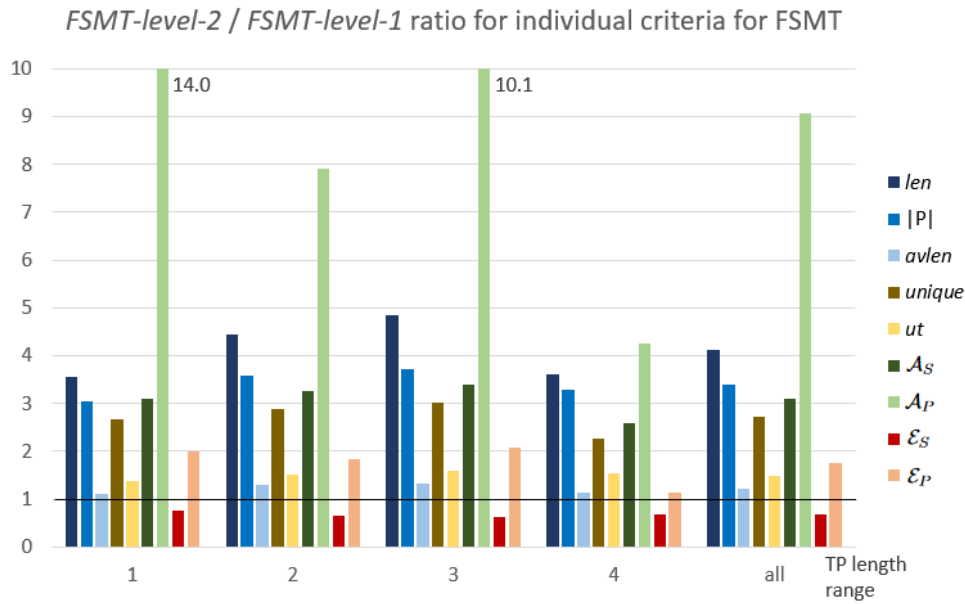


Figure 4: Comparison of averaged properties of all test path sets for *FSMT-level-1* and *FSMT-level-2* coverage criteria for the FSMT strategy.

Table 9: Comparison of test path set properties for *FSMT-level-1* and *FSMT-level-2* Coverage levels for all problem instances

Strategy	<i>len</i>	$ P $	<i>avlen</i>	<i>unique</i>	<i>ut</i>	$\mathcal{A}_S$	$\mathcal{A}_P$	$\mathcal{E}_S$	$\mathcal{E}_P$
<b>Length range set 1: <math>minLength = 2, maxLength = 4</math></b>									
NSR	2.0	1.9	1.0	1.6	1.1	1.7	2.3	0.9	1.4
FSMT	3.6	3.0	1.1	2.7	1.4	3.1	14.0	0.8	2.0
<b>Length range set 2: <math>minLength = 2, maxLength = 6</math></b>									
NSR	2.2	2.1	1.0	1.5	1.3	1.7	2.2	0.8	1.0
FSMT	4.4	3.6	1.3	2.9	1.5	3.3	7.9	0.7	1.8
<b>Length range set 3: <math>minLength = 2, maxLength = 8</math></b>									
NSR	2.2	2.1	1.1	1.5	1.4	1.6	1.7	0.7	0.8
FSMT	4.8	3.7	1.3	3.0	1.6	3.4	10.1	0.6	2.1
<b>Length range set 4: <math>minLength = 4, maxLength = 8</math></b>									
NSR	2.1	2.1	1.0	1.5	1.4	1.5	1.6	0.8	0.8
FSMT	3.6	3.3	1.1	2.3	1.5	2.6	4.3	0.7	1.1
<b>Average for length range set 1-4</b>									
NSR	2.2	2.0	1.0	1.5	1.3	1.6	1.9	0.8	1.0
FSMT	4.1	3.4	1.2	2.7	1.5	3.1	9.1	0.7	1.8

Regarding the question, how much *FSMT-level-1 Coverage* and *FSMT-level-2 Coverage* criteria differ in the potential of test paths to detect defects, the differences in  $\mathcal{E}_S$  and  $\mathcal{E}_P$  in Table 9 must be analyzed. The results suggest that, on average, test path sets that satisfy *FSMT-level-1 Coverage* detect approximately 30% more SINGLE type defects per one test path step for FSMT and 20% more for NSR. However, this fact has to be interpreted in proper context; despite this result, test path sets satisfying *FSMT-level-2 Coverage* detect more defects in total (see  $\mathcal{A}_S$  in Table 9). Regarding the low potential of both FSMT and NSR to detect PAIR type defects, we consider the difference for  $\mathcal{E}_P$  to be insignificant.

## 7 Threats to validity

In this set of experiments, a few threats may cause bias in the results. The first threat is whether the NSR used in the experiments is the best to compare with FSMT objectively. The second issue is whether a set of 171 problem instances used in the experiments is extensive enough. In the experiments, we used a combination of industrial and artificially generated FSMs with a wide variety of sizes and other properties as well as four expected test path length ranges (see Table 1). The related question is whether the examined problem instances are close enough to real-world examples. In the experiments, we used 24 problem instances created by an independent industrial team from FSMs models for various parts of Skoda Auto cars. Taking into account the trends observed for all problem instances (see Table 6) and comparing them with the trends observed for these industrial problem instances (see Table 7) and the generated problem instances (see Table 8) separately, the results and trends are very similar. Therefore, no significant bias shall be caused by the choice of the SUT models used in the experiments.

The last threat is related to the selection of the appropriate criteria for the comparison. In this study, we presented properties of test sets based on their size that are good proxies for estimating the required test effort, which is one of the key aspects in the real industrial testing process. In the comparison, we also use the number of two types of defects detected by a test path step. However, defects used in the experimental evaluation are artificial and randomly distributed in SUT models; this fact has to be taken into account when drawing conclusions from the results.

## 8 Conclusion

In this study, we proposed an MBT technique to generate test paths for FSM in an implemented strategy. The new strategy allows us to concurrently express the possible start and end of test paths in an FSM and generate those that have a length in the given interval. The already published literature may address these requirements separately but not concurrently. The practical applicability of the proposed approach has already been verified through several real models from the car industry. We have compared the proposed FSMT with the best comparable alternative, NSR. We evaluated data from 1368 runs in total. We used a combination of 171 problem instances, two coverage criteria, and four test path length ranges. For all problem instances and all test path length ranges, FSMT clearly outperformed NSR for *FSMT-level-1 Coverage* where it produced test paths with approximately only 50% of total steps compared to the test paths produced by NSR. Furthermore, the number of total test paths in  $P$  produced by FSMT was approximately 25% lower than for NSR.

For *FSMT-level-2 Coverage* the difference in the total number of steps in test paths was not significant for test length ranges that have  $maxLength \leq 6$ , but relevant for "longer" test path length ranges having  $maxLength = 8$ , where FSMT produced test paths with approximately less 20% total steps than the test paths produced by NSR. As a trade-off, FSMT produced test sets with approximately 25% more test paths than NSR.

As the total number of steps in test paths is the most important indicator that has a direct impact on testing costs, we can consider that FSMT outperforms NSR in all situations examined for *FSMT-level-1 Coverage* and one-half of examined situations for *FSMT-level-2 Coverage*, were in the second half, there was no significant difference.

Regarding the potential of test path sets to detect artificial defects inserted in a SUT model, FSMT generated test path sets detected approximately 20-30% more SINGLE type defects per test path step than NSR for *FSMT-level-1 Coverage* criterion and 10-20% for *FSMT-level-2 Coverage* criterion. The number of detected PAIR type defects was generally very low and suggested potential inefficiency of this version of the state-machine-based testing technique to detect them.

The results show good applicability of the proposed FSMT in situations when possible test path starts and ends in FSM needs to be reflected and, concurrently, the length of the test paths have to be in a defined range.

## Acknowledgments

The project is supported by CTU in Prague internal grant SGS20/177/OHK3/3T/13 “Algorithms and solutions for automated generation of test scenarios for software and IoT systems.” The authors acknowledge the support of the OP VVV funded project CZ.02.1.01/0.0/0.0/16.019 /0000765 “Research Center for Informatics.” Bestoun S. Ahmed has been supported by the Knowledge Foundation of Sweden (KKS) through the Synergi Project AIDA - A Holistic AI-driven Networking and Processing Framework for Industrial IoT (Rek:20200067).

## References

- [1] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.
- [2] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.
- [3] Tanwir Ahmad, Junaid Iqbal, Adnan Ashraf, Dragos Truscan, and Ivan Porres. Model-based testing using uml activity diagrams: A systematic mapping study. *Computer Science Review*, 33:98–112, 2019.
- [4] Wenbin Li, Franck Le Gall, and Naum Spaseski. A survey on model-based testing tools for test case generation. In *International Conference on Tools and Methods for Program Analysis*, pages 77–89. Springer, 2017.
- [5] A Simao, A Petrenko, and JC Maldonado. Comparing finite state machine test. *IET software*, 3(2):91–105, 2009.
- [6] Capers Jones and Olivier Bonsignour. *The economics of software quality*. Addison-Wesley Professional, 2011.
- [7] Karnig Derderian, Robert M Hierons, Mark Harman, and Qiang Guo. Estimating the feasibility of transition paths in extended finite state machines. *Automated Software Engineering*, 17(1):33–56, 2010.
- [8] Abdul Salam Kalaji, Robert Mark Hierons, and Stephen Swift. Generating feasible transition paths for testing from an extended finite state machine (efsm). In *2009 international conference on software testing verification and validation*, pages 230–239. IEEE, 2009.
- [9] Michiel Vroon, Bart Broekman, Tim Koomen, and Leo van der Aalst. *TMap next: for result-driven testing*. Uitgeverij kleine Uil, 2013.
- [10] Jiacun Wang. *Formal Methods in Computer Science*. CRC Press, 2019.
- [11] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.
- [12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.
- [13] Jidong Lv, Kaicheng Li, Guodong Wei, Tao Tang, Chenling Li, and Weihui Zhao. Model-based test cases generation for onboard system. In *2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS)*, pages 1–6, 2013.
- [14] C. de Souza Carvalho and T. Tsuchiya. Coverage criteria for state transition testing and model checker-based test case generation. In *2014 Second International Symposium on Computing and Networking*, pages 596–598, 2014.
- [15] M.P.E. Heimdahl and D. George. Test-suite reduction for model based tests: effects on test quality and implications for testing. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 176–185, 2004.
- [16] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 220–229, 2009.
- [17] T. Takagi, N. Oyaizu, and Z. Furukawa. Concurrent n-switch coverage criterion for generating test cases from place/transition nets. In *2010 IEEE/ACIS 9th International Conference on Computer and Information Science*, pages 782–787, 2010.
- [18] Xavier Devroey, Gilles Perrouin, and Pierre-Yves Schobbens. Abstract test case generation for behavioural testing of software product lines. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2, SPLC '14*, page 86–93, New York, NY, USA, 2014. Association for Computing Machinery.
- [19] J. Alava, T. M. King, and P. J. Clarke. Automatic validation of java page flows using model-based coverage criteria. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 439–446, 2006.

- [20] P. Liu and Z. Xu. Mtttool: A tool for software modeling and test generation. *IEEE Access*, 6:56222–56237, 2018.
- [21] P. Liu and H. Miao. A new approach to generating high quality test cases. In *2010 19th IEEE Asian Test Symposium*, pages 71–76, 2010.
- [22] Onur Kilincceker, Alper Silistre, Moharram Challenger, and Fevzi Belli. Random test generation from regular expressions for graphical user interface (gui) testing. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 170–176, 2019.
- [23] Kilincceker. Model-based ideal testing of gui programs. <https://github.com/kilincceker/MBIT4SW>, 2020.
- [24] Onur Kilincceker, Ercument Turk, Moharram Challenger, and Fevzi Belli. Regular expression based test sequence generation for hdl program validation. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 585–592, 2018.
- [25] Pan Liu, Jun Ai, and Zhenning Jimmy Xu. A study for extended regular expression-based testing. In *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, pages 821–826, 2017.
- [26] Ebrahim Fazli and Mohsen Afsharchi. A time and space-efficient compositional method for prime and test paths generation. *IEEE Access*, 7:134399–134410, 2019.
- [27] Wei Jia, Yawen Wang, Yuwei Zhang, and Yunzhan Gong. Whole program paths generation method. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pages 1–4, 2018.
- [28] Hoda Khalil and Yvan Labiche. State-based tests suites automatic generation tool (stage-1). In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 357–362, 2017.
- [29] Miroslav Bures. Pctgen: Automated generation of test cases for application workflows. In *New Contributions in Information Systems and Technologies*, pages 789–794. Springer, 2015.
- [30] Miroslav Bures, Tomas Cerny, and Matej Klima. Prioritized process test: More efficiency in testing of business processes and workflows. In *International Conference on Information Science and Applications*, pages 585–593. Springer, 2017.