# SPYH-method: an Improvement in Testing of Finite-State Machines

Michal Soucha
*Department of Computer Science*
*The University of Sheffield, UK*
msoucha1@sheffield.ac.uk

Kirill Bogdanov
*Department of Computer Science*
*The University of Sheffield, UK*
k.bogdanov@sheffield.ac.uk

***Abstract*—Test generation based on finite-state machines can improve the quality of software that can be modelled by a finite-state machine. The cost of testing depends on the size of the constructed test suite, that is, the number of tests and their combined length. There are two advanced testing methods, the H-method and the SPY-method, that generate small test suites but focus on different aspects during the construction of a test suite. This paper proposes a new testing method called the SPYH-method that combines the advantages of the two methods in order to reduce the cost of testing by constructing smaller test suites.**

***Index Terms*—model-based testing, finite-state machines, software testing**

## I. Introduction

Finite-state machines (FSM) can be used to model a wide variety of systems, from hardware components to software applications. Based on the FSM specification of a system, one can test if the system is implemented correctly. If it is not, a test should reveal the difference between the specification and the implementation. However, there can always be an error that is not revealed by any test of the given test suite. For this reason, one would like a test method that is precise in what it assumes about both the system under test and its model. This paper deals with testing methods that construct so-called *$m$-complete* test suites providing a guarantee that if the implementation and the specification respond equally to all tests of the test suite, then either the implementation is correct with respect to the specification or it has more than $m$ states. There are two state-of-the-art testing methods that construct $m$-complete test suites. They are called the H-method [1] and the SPY-method [2]. This paper proposes the SPYH-method that combines the new ideas introduced by the H- and SPY-methods.

The specification is expected to be modelled by a deterministic finite-state machine (DFSM) that is minimal and completely specified. Informally, a minimal machine has all states uniquely identifiable and a completely-specified machine has transitions for each input from all states. The new testing method will be explained on Mealy machines that is a usual representative of DFSM, but the method can be adapted easily to work with other types of DFSM, such as Moore machines and deterministic finite automata. A Mealy machine is defined as a septuple $(S, X, Y, s_0, D, \delta, \lambda)$ where $S, X$ and $Y$ are finite sets of states, inputs and outputs, respectively, $s_0 \in S$ is the initial state, $D$ is a domain of defined transitions, $\delta$ is a transition function and $\lambda$ is an output function such that $D \subseteq S \times X$, $\delta : D \to S$ and $\lambda : D \to Y$. Both the transition and output functions are lifted to work with sequences of inputs, or input sequences, in a usual way; $\delta^*(s, \varepsilon) = s$ and $\lambda^*(s, \varepsilon) = \varepsilon$ for each state $s \in S$ where $\varepsilon$ is the empty sequence, and $\delta^*(s, x \cdot v) = \delta^*(\delta(s, x), v)$ and $\lambda^*(s, x \cdot v) = \lambda(s, x) \cdot \lambda^*(\delta(s, x), v)$ for all $(s, x) \in D$ and $v$ is an input sequence consisting of defined transitions. The function $\delta^*$ thus provides the state reached by the given input sequence and $\lambda^*$ returns the sequence of output symbols observed along the way. A DFSM is *completely specified* if all transitions are defined, that is, $D = S \times X$. A DFSM is *minimal* if all states are reachable from the initial state and every pair of states is distinguishable. A state $s_i$ is reachable from a state $s_j$ if there is an input sequence $u$ such that $\delta^*(s_i, u) = s_j$. States $s_i, s_j$ are distinguishable if there is an input sequence $w$ such that $\lambda^*(s_i, w) \neq \lambda^*(s_j, w)$. Such a sequence $w$ is called a *separating sequence* of $s_i$ and $s_j$. Note that a test, or a test sequence, is an input sequence that results in the corresponding sequence of outputs when it is applied to the implementation. If a set $T$ contains sequences $uw, vw$, then $w$ is a *common extension* of $u$ and $v$ in $T$. Finally, $\text{pref}(u)$ denotes a set of all prefixes of $u$.

The paper is structured as follows. The following section sketches the H- and SPY- methods and defines a sufficient condition for a test suite to be $m$-complete. Section III describes the new testing method, the SPYH-method, and a comparison of the testing methods is demonstrated on the results of experiments in Section IV. Section V concludes this paper.

## II. Test Suite Completeness

A testing method constructing $m$-complete test suites needs to verify that each transition is defined correctly in the implementation in order to guarantee the correspondence with the specification. The approach, how a transition is verified, depends on the number $m$, on the state identification sequences of the specification and on the availability of reset that transfers the implementation to its initial state reliably. Note that state identification sequences are like separating sequences that are common for several pairs of states and so their use can reduce the size of the test suite. This paper considers that (i) $m$ is greater than or equal to the number $n$ of states of the minimal specification; (ii) the specification does not have to possess either an adaptive distinguishing sequence [3] or a unique input output sequence of any state [3]; (iii) the implementation can be reset so that the test suite can contain several sequences. If $m$ is greater than $n$, then the implementation can have $l$ *extra states* where $l = m - n$. These correspond to 'inefficiency' in the implementation and testing has to ensure that they behave as the corresponding states of the specification they were derived from. Therefore, this paper uses the term 'extra state' to refer to incorrectly implemented additional states, that is, those behaving differently to corresponding specification states.

All testing methods for DFSMs with reset follow the same approach to construct an $m$-complete test suite. Each state $s \in S$ is represented by its access sequence $\bar{s}$ that is usually the shortest path from the initial state $s_0$ to $s$. The access sequences $\bar{s}$ of all states form a state cover $\bar{S}$ that is usually initialized (contains the empty sequence $\varepsilon$) and prefix-closed. A transition $(s_i, x)$ is verified if its target state $s_j$ corresponds to $\bar{s_j}$. If $\bar{s_j} = \bar{s_i}x$, then the transition is verified trivially. Otherwise, it needs to be shown that the state reached by $\bar{s_i}x$ cannot be any state $s_k \in S$ other than $s_j$ and also that it cannot be one of $l$ possible extra states. Appending appropriate separating sequences to $\bar{s_i}x$ ensures that the target state $\delta(s_i, x)$ cannot be any state other than $s_j$. If all states reached by a sequence $u$ of length up to $l$ from $\bar{s_i}x$ are shown to correspond to the access sequence $\bar{s_k}$ where $s_k = \delta^*(s_i, xu)$, then it proves that $\bar{s_i}x$ is not an extra state. The conditions on the choice of separating sequences are captured formally in the following description of the H-method.

The H-method proposed in [1] constructs an $m$-complete test suite $T$ in the following four steps:

1) $T = P$ where $P = \bar{S} \cdot X^{\leq m-n+1}$.
2) For each $\bar{s_i}, \bar{s_j} \in \bar{S}$ such that $s_i \neq s_j$, if $T$ does not contain a common separating extension $w$ of $\bar{s_i}$ and $\bar{s_j}$ (that is, $\neg\exists \bar{s_i} \cdot w, \bar{s_j} \cdot w \in T : \lambda^*(s_i, w) \neq \lambda^*(s_j, w)$), then add such $\bar{s_i} \cdot w$ and $\bar{s_j} \cdot w$ to $T$.
3) For each $\bar{s_i} \in \bar{S}, v \in P \setminus \bar{S}$ such that $s_i \neq s_v = \delta^*(s_0, v)$, if $T$ does not contain a common separating extension $w$ of $\bar{s_i}$ and $v$ (that is, $\neg\exists \bar{s_i} \cdot w, v \cdot w \in T : \lambda^*(s_i, w) \neq \lambda^*(s_v, w)$), then add such $\bar{s_i} \cdot w$ and $v \cdot w$ to $T$.
4) If $m > n$, then for each $u, v \in P \setminus \bar{S}$ such that $u \in \text{pref}(v)$ and $s_u = \delta^*(s_0, u) \neq s_v = \delta^*(s_0, v)$, if $T$ does not contain a common separating extension $w$ of $u$ and $v$ (that is, $\neg\exists u \cdot w, v \cdot w \in T : \lambda^*(s_u, w) \neq \lambda^*(s_v, w)$), then add such $u \cdot w$ and $v \cdot w$ to $T$.

These steps capture conditions on tests included in an $m$-complete test suite but they do no specify how to obtain the common separating extensions $w$. A detailed implementation of the H-method was proposed in [4]. It specifies that the common separating extensions are chosen to enlarge the test suite with the least number of symbols that extend some test sequences. The H-method thus constructs an $m$-complete test suite based on **fixed state cover** extended with sequences of length $m - n + 1$, and it chooses **separating sequence on the fly** in order to distinguish required pairs of sequences.

The SPY-method proposed in [2] optimizes the size of resulting test suites in a very different way to the H-method. It uses **fixed separating sequences** but the **state cover can vary**. Separating sequences can be appended to any sequence that is proven to represent the corresponding state. The following theory allows such a use of different access sequences instead of a fixed one.

Convergence and divergence of test sequences with respect to a set of machines are important notions that enable one to use properties of regular languages in testing. Regular languages are equivalent to finite-state machines, however, this fact was not directly utilized for testing before the notions of convergence and divergence were proposed in [2], [5]. The structure of a DFSM was described just by the sets of access and separating sequences but the relations between sequences were missing.

*Definition 1:* Given a set of FSMs $F$, two tests are *$F$-convergent*, if both test sequences lead from the initial state to the same state in each FSM of $F$ (they converge in each FSM of $F$). Two tests are *$F$-divergent*, if both test sequences lead from the initial state to different states (they diverge in each FSM of $F$).

Definition 1 can be applied to different sets of machines but the two most important are $F$ representing the fault domain $F_T$ of DFSMs that pass the test suite $T$, that is, all implementations that respond to all tests as the specification, and set $F$ containing just the specification $M$. Two sequences $u$ and $v$ are thus $F_T$-convergent if for each $N \in F_T$, it holds that $\delta_N^*(q_0, u) = \delta_N^*(q_0, v)$ where $\delta_N$ is the transition function of a DFSM $N$; $u$ and $v$ *$F_T$-converges* [5]. Similarly, $u$ and $v$ are $M$-convergent in state $s$ if $\delta_M^*(s_0, u) = \delta_M^*(s_0, v) = s$. The curly brackets representing a set are omitted in the case of a single machine for simplicity, that is, $M$-convergent or $M$-divergent is used instead of $\{M\}$-convergent or $\{M\}$-divergent. Moreover, the set is omitted when it is clear from the context. A necessary condition for two test sequences to be $F_T$-convergent ($F_T$-divergent) is that they need to be $M$-convergent ($M$-divergent). This follows from the fact that the specification $M$ always passes its test suite $T$ and so $M$ is always in $F_T$. There is also a sufficient condition for $F_T$-divergence. Two test sequences $u$ and $v$ are $F_T$-divergent if they are $T$-separable, that is, there is a common separating extension of $u$ and $v$ in $T$.

The convergence relation is reflexive, symmetric and transitive, which means that it is an equivalence relation over the set of tests [2]. Tests in $T$ can be thus partitioned into corresponding equivalence classes

$$[u_i] = \{u_j \in T \mid u_i \text{ and } u_j \text{ are } F_T\text{-convergent}\}.$$

The following properties hold for any $u, v \in T$ such that $[u] = [v]$ [2, Lemma 1]:

1) for any sequence $w$: $[uw] = [vw]$,
2) for any sequence $t \in T : [u] \neq [t] \implies [v] \neq [t]$.

As the equivalence classes groups sequences that lead to the same state, the behaviour on all extensions of these sequences is the same as well and so $T$ can be extended to work with the equivalence classes as follows. If there is $u' \in [u]$ that is in $T$, then $[u] \in T$ and any extension of $u'$ in $T$ is an extension of $[u]$ in $T$. The classes will be called *convergent*.

Convergent classes are derivable from a pairwise comparison of tests based on their convergence and divergence, however, other notions are needed to describe relations between those classes.

*Definition 2:* Given a test suite $T$ for the specification $M$, a set of tests in $T$ is *$F_T$-convergence-preserving* if all its $M$-convergent tests are $F_T$-convergent; a set of tests in $T$ is *$F_T$-divergence-preserving* if all its $M$-divergent tests are $F_T$-divergent.

For example, a set of two $T$-separable tests is $F_T$-divergence-preserving. All machines that pass $T$ respond differently to each of the two $T$-separable tests, hence, the two test sequences are $F_T$-divergent and the set of these two sequences is $F_T$-divergence-preserving. With the convergence, it is a little harder as there is no simple sufficient condition for two or more tests to be $F_T$-convergent. The following theorem that is the same as [2, Theorem 2] except the denotation states when one can declare two tests $F_T$-convergent.

*Theorem 3:* Given a test suite $T$ for a FSM $M$ and $l = m - n \geq 0$, let $u$ and $v$ be $M$-convergent tests in $T$, such that, for any sequence $w$ of length $l$, there exist tests $u' \in [u], v' \in [v]$ and an $F_T$-divergence-preserving state cover for $M$ in $T$ containing $\{u', v'\} \cdot \text{pref}(w)$. Then, $u$ and $v$ are $F_T$-convergent.

A sufficient condition for an $m$-complete test suite based on the convergence of test sequences was proposed in [2] and its revised version is captured in the following theorem.

*Theorem 4: (SPY-condition)* Given a test suite $T$ for a DFSM $M$ and the corresponding fault domain $F_T$ of machines with up to $m$ states. If $T$ contains a $F_T$-convergence-preserving initialized transition cover for $M$, then $T$ is an $m$-complete test suite for $M$.
The proof of Theorem 4 in [2] shows that each machine $N$ in $F_T$ needs to be equivalent to $M$ due to the isomorphism between states of $M$ and $N$. The initial states correspond to each other as $T$ contains their access sequence, that is, $\varepsilon \in T$ as $T$ is initialized. All $M$-convergent sequences are $F_T$-convergent and so they lead to the same state in every $N \in F_T$, that is, there is one to one correspondence of states of $M$ and states of $N$. All transitions of $M$ are tested as $T$ contains transition cover. Therefore, Theorem 4 holds.

The SPY-method proposed in [2] employs the convergence relation (Definition 1) to reduce test branching and thus the number of sequences in the resulting test suite $T$. Test branching is the branching of the *testing tree*. A testing tree groups test sequences with common prefixes and is thus more compact and space-efficient than the test suite represented by a set of test sequences. The method aims to meet the SPY-condition (Theorem 4) that requires $F_T$-convergence-preserving initialized transition cover included in $T$. It is accomplished by verification of all transitions. A transition from state $s_i$ to state $s_j$ on input $x$ is *verified* if $\bar{s}_i \cdot x$ and $\bar{s_j}$ are proven to be $F_T$-convergent. The SPY-method first designs an $F_T$-divergence-preserving, initialized, prefix-closed and minimal state cover $\bar{S}$ and then employs Theorem 3 to verify all transitions. Note that transitions included in the state cover are verified as $\bar{S}$ is prefix-closed so that $\bar{s}_i \cdot x = \bar{s_j}$. The method uses fixed separating sequences called *harmonized state identifiers* (HSI) for distinguishing the reached states. The harmonized state identifier $H_i$ of state $s_i$ uniquely identifies $s_i$ amongst all states because for each pair of different states $(s_j, s_k)$ the related HSIs $H_j$ and $H_k$ contain a separating sequence $w_{jk}$ of $s_j$ and $s_k$ either as-is $w_{jk} \in H_j \cap H_k$ or as a prefix of longer sequences in $H_j$ and $H_k$. Traditionally, verification of a state $s_j$ requires state identification sequences to be applied from that very state. In the presence of multiple elements in $[\bar{s_j}]$, any of them can be extended by such a state identification sequence. This is referred to as 'distributing' sequences of $H_j$ over access sequences of $s_j$. Where such a sequence has a prefix of some other sequence from $s_k$ or an existing sequence from $s_k$ can be extended to the said verification sequence, only a few test input symbols are (as opposed to the whole verification sequence) needed to be added. In other words, the aim of distributing sequences is to extend existing sequences, thereby avoiding branching in the testing tree (every branch means a separate test sequence).

Algorithm 1 is an adapted version of the SPY-method proposed in the original paper [2] with a small space optimization. The original version stores all classes of convergent sequences which is not needed. Only convergent classes of the fixed access sequences of $\bar{S}$ are needed in the algorithm and so only those are handled as proposed in [4].

The SPY-method starts with the construction of state cover $\bar{S}$ and harmonized state identifiers $H_i$. Those are then concatenated accordingly to form an $F_T$-divergence-preserving state cover which is a precondition in Theorem 3 (proving convergence of two test sequences). Each transition $(s, x)$ such that $\bar{s} \cdot x$ is not proven to be convergent with $\bar{s_x}$, where $s_x = \delta(s, x)$, is verified by appending HSIs to the extensions of $\bar{s} \cdot x$ and $\bar{s_x}$. A queue $U$ helps to traverse the extensions from the shortest ones to the ones of the given maximal length $l$ that represents the number of extra states. For each extension $u$ and each separating sequence $w$ from the corresponding HSI, a suitable sequence to extend is chosen using the function APPENDSEPARATINGSEQUENCE called on lines 8 and 9 of Algorithm 1. When all extensions of length up to $l$ are appended to both convergent classes and all reached states are verified by appending the related HSIs, sequences $\bar{s} \cdot x$ and $\bar{s_x}$ are deemed to be proven $F_T$-convergent and their convergent classes are merged. Note that the classes of their successors are merged as well; for instance, sequences of $[\bar{s} \cdot x \cdot x']$ enlarge the convergent class $[\bar{s_i}]$ if $\delta(s_x, x') = s_i$ and transition $(s_x, x')$ is already verified.

Algorithm 2 describes APPENDSEPARATINGSEQUENCE$([u], w)$ that chooses a sequence $u_{best}$ of the given convergent class $[u]$ such that it is the most suitable for the extension by the given sequence $w$. The objective

**Algorithm 1:** SPY-method

```
input  : A minimal DFSM $M$ with $n$ states
input  : A number of extra states $l$; $m = n + l$
output: An $m$-complete test suite $T$ for $M$
1  $H_i \leftarrow$ harmonized state identifier of $s_i$, for all $s_i \in S$
2  $T \leftarrow \{\bar{s_i} \cdot H_i \mid \bar{s_i} \in \bar{S}\}$
3  foreach unverified transition $(s, x)$ s.t. $s_x = \delta(s, x)$ do
4      $U \leftarrow \{\varepsilon\}$    // a queue of sequences $X^{\leq l}$
5      while $U$ is not empty do
6          pop $u$ from $U$
7          foreach $w \in H_i$ for $s_i = \delta^*(s_x, u)$ do
8              APPENDSEPARATINGSEQUENCE($[\bar{s}]$, $xuw$)
9              APPENDSEPARATINGSEQUENCE($[\bar{s_x}]$, $uw$)
10         if $|e| < l$ then $U \leftarrow U \cup u \cdot X$
11     merge $[\bar{s} \cdot x \cdot u]$ and $[\bar{s_x} \cdot u]$ for all sequences $u$ where
        the two classes are not empty
12 return $T$
```

function is to minimize branching of the testing tree and total number of input symbols in $T$. Therefore, the default value of $u_{best}$ is the shortest sequence of the given class $[u]$. If there is $u'$ in $[u]$ with a maximal extension $w'$ that is a prefix of $w$, then it is chosen such $u'$ that has the longest $w'$. A sequence is maximal in a set if it is not a proper prefix of another sequence in the set. Note that the function checks whether $w$ is already in $T$ (line 4 of Algorithm 2). This choice minimizes the number of added symbols to $T$ even if there is no other option than add a new test sequence to $T$. Finally, the function APPENDSEPARATINGSEQUENCE either enlarges $T$ with a new test sequence $u_{best} \cdot w$ or appends the suffix of $w$ to the test sequence $u_{best} \cdot w'$ in $T$ that is maximal in $T$ and a $w'$ is the corresponding prefix of $w$.

**Algorithm 2:** APPENDSEPARATINGSEQUENCE($[u]$, $w$)

```
1  $u_{best} \leftarrow$ the shortest $u' \in [u]$, $maxLength \leftarrow -1$
2  foreach $u' \in [u]$ do
3      $w' \leftarrow$ the longest prefix of $w$ such that $u'w' \in T$
4      if $w' = w$ then return
5      if there is no $v$ such that $u'w'v \in T$ and $|w'| >$
        $maxLength$ then
6          $u_{best} \leftarrow u'$, $maxLength \leftarrow |w'|$
7  add $u_{best} \cdot w$ to $T$
```

## III. SPYH-METHOD

The idea of a new testing method emerged from the analysis of the two most advanced testing methods described in the previous section, the SPY-method and the H-method. The H-method uses fixed state cover that is extended with separating sequences chosen on the fly. The SPY-method uses separating sequences of fixed harmonized state identifiers but appends them to different access sequences that were proven to be convergent with the one in the fixed state cover. This section proposes a novel testing method called the SPYH-method that is a combination of the SPY- and H- methods. After the idea of the method is sketched, its implementation is described in the following subsection, the complexity is then discussed and the section is concluded with a running example of how the SPYH-method generates a test suite.

The SPYH-method is similar to the SPY-method as it also aims to satisfy the SPY-condition (Theorem 4). It gradually verifies transitions by proving the convergence but for the verification of the reached state it uses the approach of the H-method, that is, the separating sequences are chosen on the fly. The approach of choosing a separating sequence was also adapted to work with classes of convergent sequences.

The order in which unverified transitions are processed influences the size of resulting test suite. Therefore, a small optimization is proposed. The optimization sorts unverified transitions according to the sum of the lengths of the related access sequences, in particular, the value $|\bar{s}| + |\bar{s_x}|$ is used for transition $(s, x)$ leading to state $s_x$ where $\bar{s}, \bar{s_x}$ are fixed access sequences in $\bar{S}$. The convergent classes related to states that are closer to the initial state increase in their sizes sooner than the others and so the choice of convergent sequences to extend is higher when transitions from these states are to be verified. Hence, when a separating sequence is to extend an access sequence, there is a higher probability that just a few symbols are appended to a current test sequence than that the entire new test sequence is added to the test suite.

### A. Implementation

The SPYH-method is implemented as a combination of approaches and functions from the authors' implementation of the SPY- and H- methods. The main difference is handling convergent sequences as the SPYH-method works with all convergent classes, not only those related to access sequences of a fixed state cover like in the SPY-method. Classes are stored and represented as *convergent nodes* (CN) of a *convergent graph*. A convergent graph is a transition diagram of a DFSM. Initially, it corresponds to the testing tree that captures traces of test sequences. Then, two subtrees are merged when the sequences leading to the roots of the subtrees are proven to be convergent. Such a merge can create a cycle in the convergent graph, therefore, it has no longer a tree structure. The corresponding merge is done every time two sequences become $F_T$-convergent. Finally, the convergent graph represents the specification $M$ on which the design of the test suite $T$ is based. Convergent nodes thus group prefixes of test sequences such that these prefixes are convergent if they reach the same node. Whenever any following algorithm uses a convergent class (denoted by [.]), the implementation works with the corresponding convergent node. Incremental state merging is also at the heart of passive learning methods such as RPNI [6] and Blue Fringe [7]. It corresponds to a known dichotomy between learning and testing [8]: a DFSM learnt from an $m$-complete test suite using state merging should be equivalent to the specification.

Algorithm 3 captures the main flow of the SPYH-method. It first designs an initialized prefix-closed state cover $\bar{S}$ that is then made $F_T$-divergence-preserving using the function DISTINGUISH. Every sequence of $\bar{S}$ needs to be distinguished from each other to create an $F_T$-divergence-preserving $\bar{S}$. The proposed optimization sorts the unverified transitions that are then processed. An unverified transition $(s, x)$ first needs to be covered in $T$ (line 6) and then both $[\bar{s} \cdot x], [\bar{s_x}]$, where $s_x = \delta(s, x)$, need to be extended to satisfy Theorem 3. DISTINGUISHFROMSET takes care of suitable extensions satisfying all requirements so that both convergent classes (and CNs as well) can be then merged as they become convergent. Finally, the $m$-complete test suite is returned.

```
Algorithm 3: SPYH-method
input : A minimal DFSM $M$ with $n$ states
input : A number of extra states $l$; $m = n + l$
output: An $m$-complete test suite $T$ for $M$
1 $T \leftarrow \bar{S}$
2 foreach $\bar{s} \in \bar{S}$ do
3     DISTINGUISH($[\bar{s}], \bar{S}$)
4 sort unverified transitions according to $|\bar{s}| + |\bar{s_x}|$
    calculated for each transition $(s, x)$ and $s_x = \delta(s, x)$
    in the increasing order
5 foreach unverified transition $(s, x)$ with $s_x = \delta(s, x)$ do
6     add $\bar{s} \cdot x$ to $T$ if not there
7     DISTINGUISHFROMSET($[\bar{s} \cdot x]$, $[\bar{s_x}]$, copy of $\bar{S}$, $l$)
8     merge $[\bar{s} \cdot x \cdot u]$ and $[\bar{s_x} \cdot u]$ for all sequences $u$ where
        the two classes are not empty
9 return $T$
```

DISTINGUISH described in Algorithm 4 separates the given convergent class $[u]$ from the classes $[v]$ of all $v$ in the given $V$ that correspond to different states than $\delta^*(s_0, u)$. The choice of separating sequences is the same as in the case of the H-method but it works with convergent classes. First, a pair of convergent classes is compared and the function GETPREFIXOFSEPSEQ determines the prefix $w'$ of their separating sequence such that (i) the prefix $w'$ is already in $T$ as a common extension of both classes and (ii) the corresponding separating sequence should enlarge $T$ the least. If the classes are not distinguished in the existing testing tree, then the shortest separating sequence $w$ of states reached by $w'$ from $\delta^*(s_0, u)$ and $\delta^*(s_0, v)$ is appended to the related classes using APPENDSEPARATINGSEQUENCE defined in Algorithm 2.

Algorithm 5 specifies the function DISTINGUISHFROMSET that controls which pairs of convergent classes are to be distinguished in order to satisfy Theorem 3 and so verify a transition. DISTINGUISHFROMSET extends two sequences simultaneously as both $\bar{s} \cdot x$ and $\bar{s_x}$ (and their extensions) are to be in an $F_T$-divergence-preserving state cover according to Theorem 3. The given set $V$ of sequences always contains the fixed state cover $\bar{S}$ and any sequence that is added to $V$ on lines 5 and 6 (and then removed on lines 11 and 12). Using this adding and removing of sequences, the classes of proper prefixes of $[u]$ and $[v]$ are distinguished from $[u]$ and $[v]$ which is needed for a divergence-preserving set. This corresponds to the step 4) of the H-method. The required extensions of length up to $l$ do not have to be already in $T$, therefore, lines 8 and 9 of Algorithm 5 add them gradually by one symbol. DISTINGUISH is called (lines 1 and 3) before the recursive call of DISTINGUISHFROMSET (line 10). It means that the separating sequences appended to $u$ can cover some of the required extensions and sequences appended to these extensions can have these separating sequences as prefixes. Hence, the total number of test sequences could be reduced compared to the case when the construction of all extensions of length $l$ is followed by appending separating sequences. The boolean variable *notReferenced* controls that a fixed access sequence $\bar{s}$ is not added to $V$ if it is already there and that DISTINGUISH is not called with a class $[v]$ representing a state. As $[u]$ represents $[\bar{s} \cdot x]$ when the function is first called from Algorithm 3, this class and its successor do not correspond to the classes of states and so the condition is not checked on lines 1, 5 and 12.

```
Algorithm 4: DISTINGUISH($[u], V$)
1 foreach $v \in V$ such that $\delta^*(s_0, u) \neq \delta^*(s_0, v)$ do
2     $(e, w') \leftarrow$ GETPREFIXOFSEPSEQ($[u], [v]$)
3     if $e > 0$ then
4         $w \leftarrow$ the shortest separating sequence of
            $\delta^*(s_0, uw')$, $\delta^*(s_0, vw')$
5         APPENDSEPARATINGSEQUENCE($[u]$, $w'w$)
6         APPENDSEPARATINGSEQUENCE($[v]$, $w'w$)
```

```
Algorithm 5: DISTINGUISHFROMSET($[u]$, $[v]$, $V$, depth)
1 DISTINGUISH($[u], V$)
2 notReferenced $\leftarrow \forall \bar{s} \in \bar{S} : \bar{s} \notin [v]$
3 if notReferenced then DISTINGUISH($[v], V$)
4 if depth $> 0$ then
5     add $u$ to $V$
6     if notReferenced then add $v$ to $V$
7     foreach $x \in X$ do
8         APPENDSEPARATINGSEQUENCE($[u]$, $x$)
9         APPENDSEPARATINGSEQUENCE($[v]$, $x$)
10        DISTINGUISHFROMSET($[ux]$, $[vx]$, $V$, depth $- 1$)
11    if notReferenced then pop $v$ from $V$
12    pop $u$ from $V$
```

The last part of the SPYH-method is the function that checks all common extensions of the given classes and chooses the best prefix of a separating sequence that should extend $T$. GETPREFIXOFSEPSEQ in Algorithm 6 uses the variables *minEst* and *bestPrefix* to store information about the best way so far to distinguish the given sequences. An estimate of symbols that would extend $T$ is calculated for each maximal common extension. The estimate includes the length of access sequences as well if a new test sequence is to enlarge $T$, that is, if there is no maximal sequence in the corresponding convergent class. The function HASLEAF defined on lines 5–6 checks whether the given class contains a sequence that is maximal in $T$ and so it provides information if the chosen separating sequence can easily extend a current test sequence. Note that HASLEAF is implemented as a property of convergent nodes that keep track which sequences reach the nodes but do not have successors in the testing tree. The initial estimation of the number of symbols that extend $T$ assumes the worst case where the shortest separating sequence $w$ will extend both classes and no prefix of $w$ extends any sequences of the convergent classes. Therefore, *minEst* gets twice the length of $w$ plus the lengths of the shortest sequences $u, v$ of the given classes if they do not contain a maximal sequence. All inputs are then compared if they begin a common extension that could lead to a better estimate. If both classes have an extension starting with the input $x$, then a recursive call is made to check the corresponding classes $[ux], [vx]$ unless $x$ separates states reached by $u, v$ or $x$ transfers these states to the same state. In the former case (line 12 of Algorithm 6), the function returns that the given classes are already distinguished in $T$. In the latter case (line 13), another input $x$ is considered as no separating sequence can start with $x$. The recursive call of the function returns two values, an estimate $e$ of symbols that enlarge $T$ and a sequence $w'$ that is an extension of the given $[ux]$ and $[vx]$ in $T$ and is also a prefix of the best identified separating sequence of states reached by $ux, vx$. If $e$ is 0, then $[ux]$ and $[vx]$ are distinguished in $T$ and so $[u], [v]$ are distinguished as well. Otherwise, $e$ is compared with *minEst* that stores the minimal estimate amongst inputs $x$ considered so far. If $e$ is lower or equal to *minEst* (line 16), then *minEst* is updated with $e$ and *bestPrefix* with $xw'$. Note that the equality in the condition on line 16 means that longer extensions are favoured in the selection. This aims to select extensions that are not proper prefixes of other extensions and so the number of sequences in $T$ does not increase when the chosen separating sequence is appended to $[u]$ and $[v]$. If one of $[u], [v]$ has no extension starting with $x$ (lines 19–26 and 28–34), then it depends on which of the classes has such an extension. The function ESTIMATEGROWTHOFT defined in Algorithm 7 initializes $e$ with an estimate of the number of symbols added $T$ by appending a separating sequence. If just $x$ separates the reached states ($e = 1$), then only the class that is not extended with $x$ is checked whether it contains a maximal sequence and if not, the length of the shortest access sequence increases $e$ (lines 24 and 33). Otherwise, the other class needs to be checked as well (lines 21–23 and 30–32). If $e$ is lower than *minEst*, both *minEst* and *bestPrefix* are updated accordingly (lines 25–26 and 34). Finally, both variables are returned as the values capturing the best way to distinguish the given $[u]$ and $[v]$.

If an input $x$ does not start extensions of both $u$ and $v$ in $T$, then the function ESTIMATEGROWTHOFT estimates the number of symbols that would be added to $T$ to distinguish the given states $s_u$ and $s_v$ if the separating sequence began with $x$. Algorithm 7 defining ESTIMATEGROWTHOFT assumes that one of classes $[u], [v]$ is extended with $x$ in $T$. Therefore, it returns 1 if $x$ separates the given states. If $x$ cannot begin the shortest separating sequence because the states go on $x$ to themselves or to a single state, then $2n$ is returned. Note that twice the number of states $n$ is always greater than twice the length of the shortest separating sequence of a state pair. Otherwise, ESTIMATEGROWTHOFT estimates that $T$ would

**Algorithm 6:** GETPREFIXOFSEPSEQ($[u]$, $[v]$)

```
1  Let $u, v$ be the shortest sequences of the given classes
2  $s_u \leftarrow \delta^*(s_0, u)$, $s_v \leftarrow \delta^*(s_0, v)$
3  minEst $\leftarrow 2|w|$ where $w$ is the shortest separating
     sequence of $s_u, s_v$
4  bestPrefix $\leftarrow \varepsilon$
5  HASLEAF($[u]$):
6      returns true if there is $u' \in [u]$ that is maximal in $T$
7  if not HASLEAF($[u]$) then minEst $\leftarrow$ minEst $+ |u|$
8  if not HASLEAF($[v]$) then minEst $\leftarrow$ minEst $+ |v|$
9  foreach $x \in X$ do
10     if there are $u' \in [u]$ and $w_u \in X^*$ such that
         $u'xw_u \in T$ then
11         if there are $v' \in [v]$ and $w_v \in X^*$ such that
             $v'xw_v \in T$ then
12             if $\lambda^*(s_u, x) \neq \lambda^*(s_v, x)$ then return $(0, \varepsilon)$
13             if $\delta(s_u, x) = \delta(s_v, x)$ then continue
14             $(e, w') \leftarrow$ GETPREFIXOFSEPSEQ($[ux]$, $[vx]$)
15             if $e = 0$ then return $(0, \varepsilon)$
16             if $e \leq$ minEst then
17                 minEst $\leftarrow e$, bestPrefix $\leftarrow xw'$
18         else
19             $e \leftarrow$ ESTIMATEGROWTHOFT($s_u$, $s_v$, $x$)
20             if $e \neq 1$ then
21                 if HASLEAF($[u]$) then $e \leftarrow e + 1$
22                 else if not HASLEAF($[ux]$) then
23                     $e \leftarrow e + |u| + 1$
24             if not HASLEAF($[v]$) then $e \leftarrow e + |v|$
25             if $e <$ minEst then
26                 minEst $\leftarrow e$, bestPrefix $\leftarrow x$
27     else if there are $v' \in [v]$ and $w_v \in X^*_{\uparrow}$ such that
         $v'xw_v \in T$ then
28         $e \leftarrow$ ESTIMATEGROWTHOFT($s_u$, $s_v$, $x$)
29         if $e \neq 1$ then
30             if HASLEAF($[v]$) then $e \leftarrow e + 1$
31             else if not HASLEAF($[vx]$) then
32                 $e \leftarrow e + |v| + 1$
33         if not HASLEAF($[u]$) then $e \leftarrow e + |u|$
34         if $e <$ minEst then minEst $\leftarrow e$, bestPrefix $\leftarrow x$
35 return (minEst, bestPrefix)
```

be enlarged by $2 \cdot |w| + 1$ symbols where $w$ is the shortest separating sequence of next states of $s_u, s_v$ on $x$ such that $w$ would be appended to both $[ux], [vx]$ and $+1$ stands for one $x$ appended to either $[u]$ or $[v]$. The function assumes that no prefix of $w$ is an extension of $[ux]$ or $[vx]$ in $T$, therefore, the estimate is always higher than or equal to the actual number of symbols that enlarge $T$ by appending $w$ to $[ux], [vx]$. Note that instead of constructing $w$ for every call of ESTIMATEGROWTHOFT, the SPYH-method stores lengths of the shortest separating sequences for each pair of states. Besides the lengths, the same array stores information to which pair of states the pair of states transfers on particular input and if an input can begin a separating sequence. This structure is called a state pair array of all separating sequences and was introduced in [4]. The separating sequence $w$ is then obtained based on the connections between cells of the array only once for each call of DISTINGUISH.

**Algorithm 7:** ESTIMATEGROWTHOFT($s_u$, $s_v$, $x$)

```
1 if $\lambda(s_u, x) \neq \lambda(s_v, x)$ then return 1
2 if $\delta(\{s_u, s_v\}, x) = \{s_u, s_v\}$ or $|\delta(\{s_u, s_v\}, x)| = 1$ then
3     return $2n$
4 return $2 \cdot |w| + 1$ where $w$ is the shortest separating
    sequence of $\delta(s_u, x)$ and $\delta(s_v, x)$
```

## B. Time and Space Complexity

Meaningful time and space complexities are not easy to derive as they are really dependent on the structure of machine under test, that is, access sequences of states and their separating sequences, the number of states $n$, the number of inputs $p$ and others. The space complexity for the SPYH-method includes the resulting testing tree, the convergent graph and a state pair array of all separating sequences. The convergent graph represents the machine under test in the end so that it takes $O(n)$ space. The state pair array has space of $O(n^2)$. However, the testing tree depends on test sequences. Its size is bounded by the total length of test sequences, that is, the size of test suite, but it is usually much smaller because each common prefix of several test sequences is stored in the testing tree just once. The upper bound of the size of test suite is possible to derive by considering the W-method that is the oldest testing method [9], [10]. Each test sequence has three parts: the access sequence of a state, the input of the tested transition, the extension of length up to the given $l$ and a separating sequence; the access and separating sequences are at most $n-1$ long. The length of a test sequence is thus at most $(n-1)+1+l+(n-1)$ which is in $O(2n+l)$. The number of test sequences is bounded by $n \cdot p^{l+1} \cdot (n-1)$ because there is $n$ access sequences that are extended with all sequences of length up to $l+1$ and at most $n-1$ separating sequences are then appended. Together, the size of test suite is in $O((2n+l)(n^2 p^{l+1}))$ which is in $O(n^3 p^{l+1})$ if $n$ is strictly greater than $l$. This is important bound as the standard testing methods including the H- and SPY- methods have the same (worst case) space complexity $O(n^3 p^{l+1})$.

The worst case time complexity can be calculated based on Algorithms 2–7. The most time is spent in the function DISTINGUISH. It is called approximately $(n + np \cdot p^l \cdot 2)$ times; for each of $n$ access sequences (line 3 of Algorithm 3) 1 call plus for each of at most $np$ unverified transitions and each of their $p^l$ extensions there are 2 calls. Note that the exact number of extensions is $\frac{p^{l+1}-1}{p-1}$ as the sum of a geometric progression. Inside the function, the given class $[u]$ is distinguished from particular classes in $V$. For the first $n$ calls of DISTINGUISH $|V|$ is $n$ and for the other calls the size of $V$ is at most $n+l$. All common extensions of the given classes are checked by GETBESTPREFIXOFSEPSEQ (line 2 of Algorithm 4). There are very different numbers of extensions for different classes during the design, however, it is possible to bound them by the (worst case) size of test suite, that is, $n^3 p^{l+1}$. The separating sequence $w$ can be obtained proportionally to its length so that at most in $O(n)$. The last bit are two calls of APPENDSEPARATINGSEQUENCE (Algorithm 2). This function chooses one sequence of the given class and appends $w$ to it. Let assume that every class has the same number of convergent sequences in the end, that is, there are $n$ classes so that each has $n^2 p^{l+1}$ sequences. However, as each sequence of the class is checked for an extension that is a prefix of the given $w$, APPENDSEPARATINGSEQUENCE runs in $O(n^3 p^{l+1})$. Putting the above figures together, the SPYH-method spends $O((n^2 + 2np^{l+1}(n+l))(n^3 p^{l+1} + n + n^3 p^{l+1}))$, or $O((n^5 + n^4 l)p^{2l+2} + (n^5 + n^3 + n^2 l)p^{l+1} + n^3)$, time with function DISTINGUISH. The other parts such as generation of state cover, sorting unverified transitions or merging convergent classes, do not change the estimated complexity so that the worst case time complexity of the SPYH-method is $O((n^5 + n^4 l)p^{2l+2})$. Nevertheless, the experiments in Section IV will show that the complexity is close to quadratic in most cases.

## C. Running Example

The construction of an $m$-complete test suite by the SPYH-method is explained on an example in this subsection. One of the simplest machines is a turnstile with the control system that can be modelled by the completely-specified Mealy machine shown in Fig. 1. It has 2 states, Locked and Unlocked (abbreviated to L and U), 2 inputs (‘c’ and ‘p’), 3 outputs (‘N’, ‘L’ and ‘F’) and the turnstile is initially in the state Locked to which the machine can be reset any time. The letters that denote the inputs and outputs stand for actions and observations depicted in Fig. 1. Each transition is labelled with an input symbol and the corresponding output symbol, for example, if one inserts a coin in the turnstile (input ‘c’), no response is observed (output ‘N’) but the machine transfers into the state Unlocked.

Assume that the control chip of the turnstile allows to represent up to 3 states. Therefore, 1 extra state is considered in the construction of an $m$-complete test suite in order to confirm that the turnstile is implemented correctly according

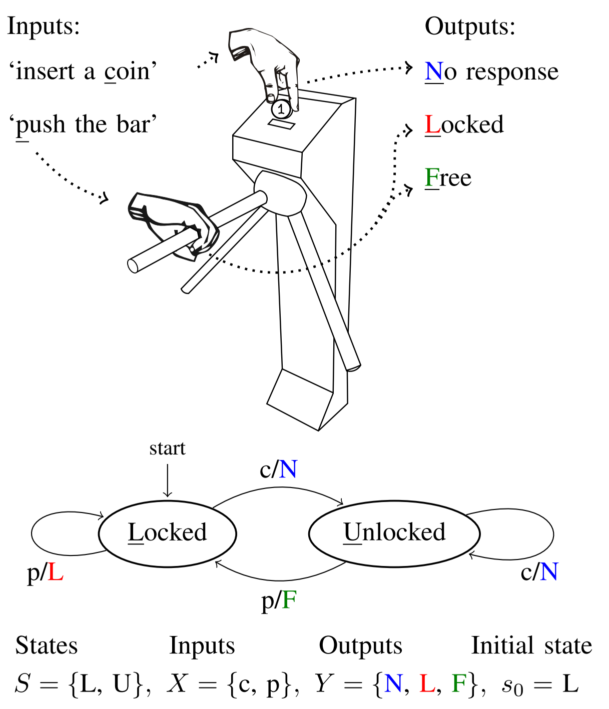

Fig. 1. A turnstile with its specification

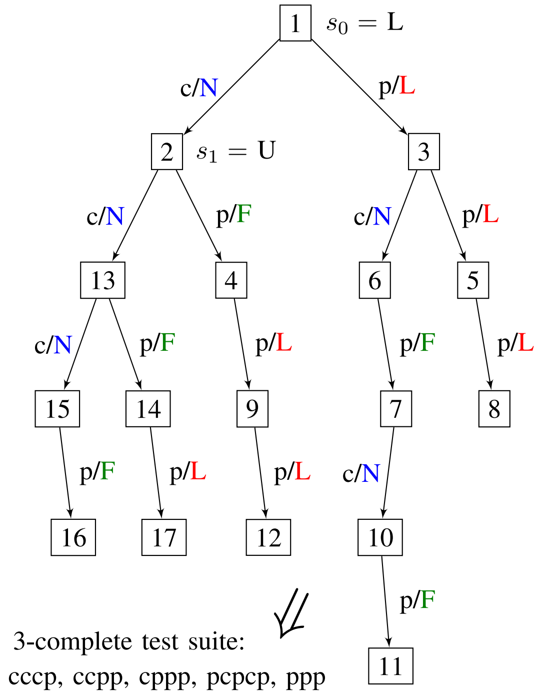

Fig. 2. Testing tree constructed by the SPYH-method for the turnstile defined in Fig. 1 and 1 extra state

to the specification; $m = 3$ as the specification has 2 states and the number $l$ of extra states is 1.

The test suite $T$ is initialized with a prefix-closed state cover $\bar{S}$ (line 1 of Algorithm 3) that corresponds to the sequences $\varepsilon$ and 'c'. The SPYH-method stores $T$ in the testing tree. Fig. 2 shows the final testing tree with nodes numbered according to the order in which the test sequences are added to the test suite $T$. The testing tree with just $\bar{S}$ contains only the nodes 1 and 2 that represent both states of the turnstile. The method needs to distinguish the access sequences of $\bar{S}$ in order to create a divergence-preserving state cover in $T$. Both access sequences have no common extension, therefore, GETBESTPREFIXOFSEPSEQ returns (2, $\varepsilon$) and the shortest separating sequence 'p' is appended to both $\varepsilon$ and 'c' (nodes 3 and 4 of the testing tree).

There are 3 transitions that are not verified. They are sorted according to line 4 of Algorithm 3. The transition (L, p) will be verified first, then the transition (U, p) and the last one will be (U, c). This is the first change compared to the H- and SPY- methods because (U, c) would be verified before (U, p) in their case as the input 'c' is lexicographically lower than 'p'. The SPYH-method looks at the length of the access sequences of both the start and target states. Therefore, the transition (U, p) leading to L has the value of 1 and (U, c) corresponds to 2 as it leads back to the state U with the access sequence 'c'.

The first unverified transition is (L, p). It is already captured in $T$ so that the function DISTINGUISHFROMSET is called with the parameters $u$ = p, $v = \varepsilon$, $V = \{\varepsilon$, c$\}$ and *depth* $= 1$. In order to distinguish $[u]$ from $V$, the separating sequence 'p' extends 'p'; the node 5 is created in the testing tree by the function APPENDSEPARATINGSEQUENCE called on line 5 of Algorithm 4. Note that the separating sequence 'p' already extends the access sequence $\varepsilon$ of the state L so that the call of APPENDSEPARATINGSEQUENCE on line 6 of Algorithm 4 does not influence $T$. The variable $v$ in DISTINGUISHFROMSET corresponds to the access sequence of L so that the boolean variable *notReferenced* gets false and DISTINGUISH is not called for $[v]$. As 1 extra state is considered and so *depth* $= 1$, the successors of $\delta$(L, p) need to be checked. The sequence 'p' is thus added to $V$ (line 5 of Algorithm 5) and for each input the corresponding successors of both $[u]$ and $[v]$ are first created if they are not already captured in $T$ and DISTINGUISHFROMSET is called on them. The first input 'c' does not already extend 'p' so that 'pc' is added to $T$ and node 6 is created in the testing tree. $[\varepsilon]$ is already extended with 'c'. DISTINGUISHFROMSET called on [pc] and [c] calls only DISTINGUISH$([u], V)$ because 'c' is the access sequence so that *notReferenced* is again false and *depth* is 0 as it was decreased by 1 in the recursive call. The sequence 'pc' leads to the state U, therefore, DISTINGUISH appends the separating sequence 'p' to 'pc' (node 7) in order to distinguish it from $\varepsilon$ and 'p' that are in $V$ and lead to the state U. No other extensions are needed to verify 'pc'. The second input 'p' already extends both 'p' and $\varepsilon$ so that DISTINGUISHFROMSET is called on [pp] and [p]. It is again sufficient to append 'p' (node 8) and 'pp' is verified. All successors are thus checked and the transition (L, p) is verified. The nodes of the testing tree are merged in the convergent graph such that nodes 1, 3, 5 and 8 represent the state L, nodes 2 and 6 represent the state U, and [cp] contains nodes 4 and 7 that represent $\delta$(U, p).

The next unverified transition is (U, p). The transition is captured in $T$ but the reached state L is not verified so that the separating sequence 'p' is appended (node 9). In order to check the successor of $\delta$(U, p) on 'c', 'c' needs to extend [cp]. APPENDSEPARATINGSEQUENCE chooses 'pcp' from [cp] because it is maximal in $T$ and so 2 input symbols are saved in the total number of symbols of the test suite compared to the case of extending 'cp' that is not maximal in $T$ (which is the case of the H-method). The sequence 'pcpc' is then extended with the separating sequence 'p' (node 11). After 'p' is also appended to 'cpp', (U, p) is verified and all 12 nodes of the testing tree are included in the corresponding convergent classes [$\varepsilon$] and [c], that is, [c] contains nodes 2, 6 and 10, and the other nodes are grouped in [$\varepsilon$].

The last unverified transition (U, c) is verified by the nodes 13–17 of the testing tree in Fig. 2. The constructed 3-complete test suite $T$ consists of maximal sequences in the testing tree, that is, 'cccp', 'ccpp', 'cppp', 'pcpcp' and 'ppp'. It contains 5 test sequence and 20 input symbols in total. The testing tree produced by the H-method would have 'cp' appended to 'cp' (node 4) instead of 'pcp' (node 7) which would result in 6 tests of total length of 22. The SPY-method would append this 'cp' to 'cccp' and so the constructed test suite would also have 5 test sequences and 20 input symbols in total.

## IV. Experiments

The new testing method proposed in the previous section was compared with a number of well-known methods on randomly-generated machines. The results of experiments are described in this section. Besides the SPY- and H- methods, the standard testing methods include the W-method [9], [10], the Wp-method [11] and the HSI-method [12]. The implementation of each method used for experimental evaluation is described in [4] and available in FSMlib v3.1[1] developed by the authors.

The FSMlib contains a generator of random DFSM models. The DFSM generator first assigns the target state to each transition randomly and then changes some of the transitions such that each state is reachable from the initial state. The outputs are also assigned randomly but such that each output symbol is captured at least once in the machine. If the generated machine is not minimal, it is thrown away and another machine is generated. This is repeated until the given number of minimal completely-specified machines with the given numbers of states, inputs and outputs is obtained. The experiments consist of 1700 Mealy machines and 1700 Moore machines with 5 inputs and 5 outputs. There are 17 groups of 100 machines with different number of states for both machine types. The number of states of these 17 'state groups' are: multiples of 10 ranging from 10 to 100 (10 groups) and 150, 200, 300, 400, 600, 800 and 1000. Each of the 6 testing methods constructs 3 $m$-complete test suites for each of 3400 machines depending on the given number $l$ of extra states that is 0, 1 or 2. All machines and the results are available in the repository FSMmodels v1.0[2].

The exploration efficiency is a new objective developed by the authors. It is calculated as the number of edges in the testing tree of $T$ divided by the total length of tests in $T$. As it is based on the testing tree, it permits one to evaluate how much of the implementation will be explored by tests, even in the implementation with much more states than the specification. Moreover, it captures how many prefixes of tests are overlapping with other tests, for example, the fixed access sequences are covered by several tests. The exploration efficiency is thus higher (and better) if a testing method constructs longer sequences that do not overlap much. In the case of the test suite $T$ constructed for the turnstile (Fig. 1) by the SPYH-method in the previous section, the final testing tree (Fig. 2) has 16 edges and the total number of inputs in $T$, or the total length of tests, is 20. Therefore, the exploration efficiency of the SPYH-method is 80 % in this case.

Fig. 3 shows the results for Mealy machines and 0 extra states. It compares the testing methods on 4 objectives: the total number of inputs in the constructed test suite $T$, the number of tests in $T$, the exploration efficiency and the time spent by the construction of $T$. Each of 4 graphs show the first and third quartiles calculated for each state group of 100 machines, and boxplots with minimum and maximum values as whiskers for the machines with 1000 states.

The SPYH-method beats the standard testing methods in the three objectives that directly relate to the testing of the implementation. The method constructs the least tests and their total length (total number of inputs) is also minimal. Hence, a tester spends less time with testing the implementation. This is at the cost of longer time of construction of tests. However, less than 2 seconds to create an $n$-complete test suite for a machine with 1000 states is acceptable. Note that the size of test suites in terms of the number of tests or the total number of inputs grows linearly with the number of states and the construction time seems to grow quadratically. Therefore, the worst-case time and space complexity derived in Section III-B are far away for the randomly-generated machines used for experiments. Fig. 3 captures just one setting out of 6 possible (2 machine types and 3 different numbers of extra states). The results of other settings capture the same trends. The growth of values remains linear in the total number of inputs and the number of tests. The relative order of the testing methods also remains but the values change. The values are multiplied by 5 when the number of extra states increases by 1; this corresponds to $p^l$ in the derived time and space complexities.

## V. Conclusion

This paper proposed a new testing method, the SPYH-method, that combines the two most advanced testing methods for completely-specified DFSM with reset. It was experimentally shown that the SPYH-method constructs much smaller test suites compared to the two most advanced methods.

[1]https://github.com/Soucha/FSMlib/releases/tag/v3.1

[2]https://github.com/Soucha/FSMmodels/releases/tag/v1.0

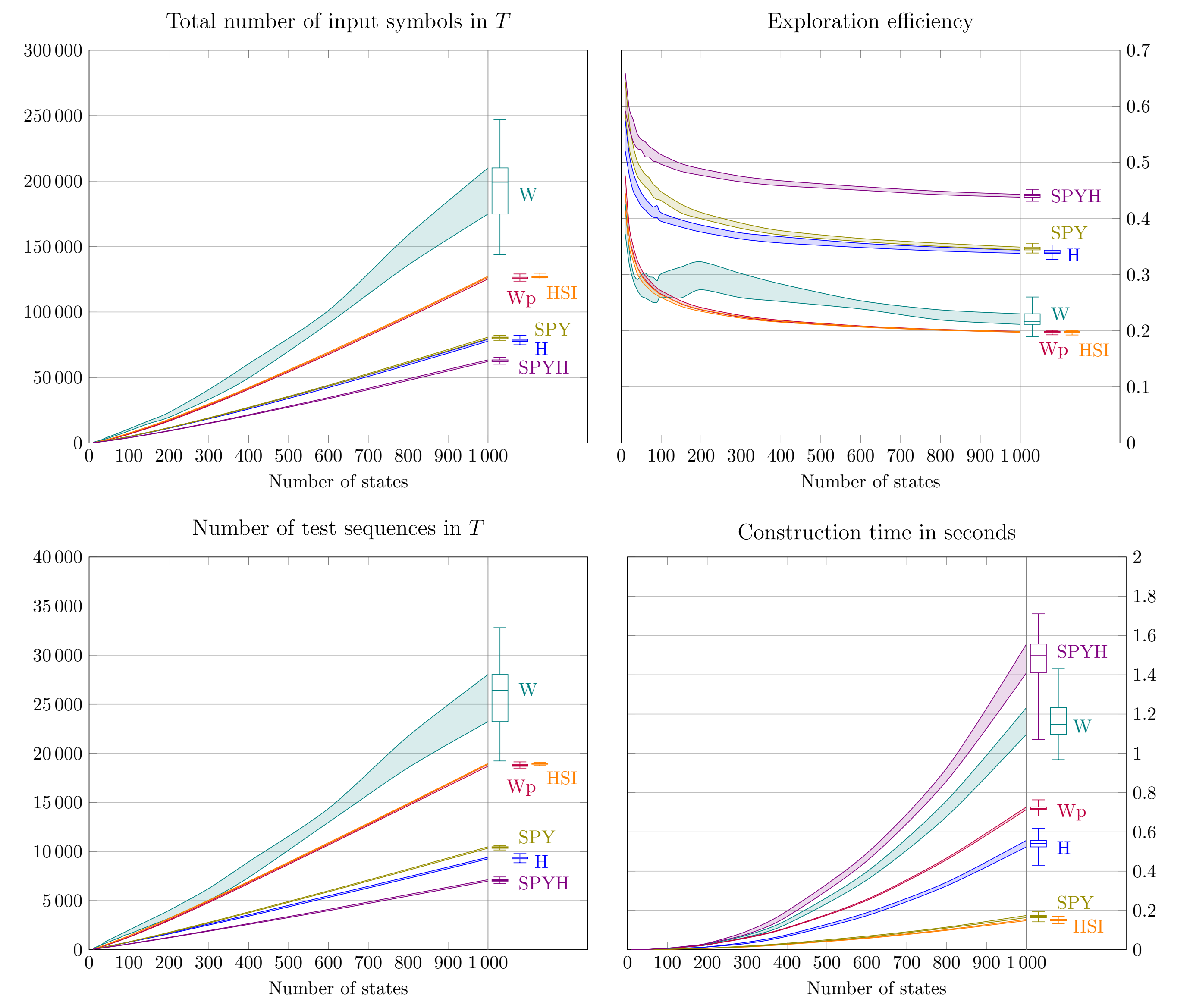

Fig. 3. Construction of test suites $T$ for Mealy machines with 5 inputs and no extra state: first and third quartiles calculated for 100 machines per each state group; boxplots with whiskers from minimum to maximum for machines with 1000 states are shown on the right of each graph

## References

[1] R. Dorofeeva, K. El-Fakih, and N. Yevtushenko, "An improved conformance testing method," in *Formal Techniques for Networked and Distributed Systems-FORTE 2005*. Springer, 2005, pp. 204–218.

[2] A. Simão, A. Petrenko, and N. Yevtushenko, "On reducing test length for FSMs with extra states," *Software Testing, Verification and Reliability*, vol. 22, no. 6, pp. 435–454, 2012.

[3] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[4] M. Soucha, "Checking experiment design methods," Master thesis, 2015.

[5] A. Simão and A. Petrenko, "Checking sequence generation using state distinguishing subsequences," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. IEEE, 2009, pp. 48–56.

[6] J. Oncina and P. Garcia, "Inferring regular languages in polynomial updated time," in *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*. World Scientific, 1992, pp. 49–61.

[7] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm," in *Grammatical Inference; 4th International Colloquium, ICGI-98*, ser. LNCS/LNAI, vol. 1433. Springer, 1998, pp. 1–12.

[8] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen, "On the correspondence between conformance testing and regular inference," in *Fundamental Approaches to Software Engineering*. Springer, 2005, pp. 175–189.

[9] M. Vasilevskii, "Failure diagnosis of automata," *Cybernetics and Systems Analysis*, vol. 9, no. 4, pp. 653–665, 1973.

[10] T. S. Chow, "Testing software design modeled by finite-state machines," *Software Engineering, IEEE Transactions on*, no. 3, pp. 178–187, 1978.

[11] S. Fujiwara, F. Khendek, M. Amalou, A. Ghedamsi *et al.*, "Test selection based on finite state models," *Software Engineering, IEEE Transactions on*, vol. 17, no. 6, pp. 591–603, 1991.

[12] A. Petrenko, "Checking experiments with protocol machines," in *Proceedings of the IFIP TC6/WG6. 1 Fourth International Workshop on Protocol Test Systems IV*. North-Holland Publishing Co., 1991, pp. 83–94.