Acta Cybernetica **00** (0000) 1–27.

FSM-based Incremental Algorithms for Test Generation and Testing *

Gábor Árpád Németh†, Zoltán Pap†,
Gábor Kovács† and Mahadevan Subramaniam‡

Abstract

In this paper we propose two incremental algorithms to maintain a test set across a series of system revisions for deterministic finite state machines. In each development cycle, the algorithms utilize an existing information – a test set available for a previous version of the system – to produce a new checking sequence by modifying only the subset of sequences that are affected by the given series of changes. The algorithms both support incremental testing – create test cases only for the affected parts of the system – and maintain a checking sequence for the whole system. The first algorithm updates a state cover set in order to reach all states of the machine, while the second one maintains a separating family of sequences used to verify the next state of transitions. The two algorithms can be applied together or independently depending the testing purpose. The analytical and practical analyzes show that our algorithms are very efficient in case of changing system specifications. We also demonstrate our methods through an example.

Keywords: finite state machine, test generation algorithms, test case maintenance, incremental development

1 Introduction

Although finite state machine (FSM)-based test generation methods have been extensively studied in the last few decades [6] [3] [16] [4] [7], very little effort has been focused on dynamic scenarios involving changing system specifications. This is especially surprising considering that most recent system development models propose incremental approaches involving step-by-step refinement of the system under

*This work is connected to the scientific program of the "Development of quality-oriented and cooperative R+D+I strategy and functional model at BME" project. This project is supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

†High Speed Networks Laboratory, Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, Magyar tudósok körútja 2, H-1117, Budapest, HUNGARY E-mail: {gabor.nemeth, pap, kovacsg}@tmit.bme.hu

‡Computer Science Department, University of Nebraska at Omaha, Omaha, NE 68182, USA, E-mail: msubramaniam@mail.unomaha.edu

development [22] [12] [8]. From the testing perspective evolutionary approaches open new possibilities to improve test generation methods. By identifying the effect of changes in each iteration step during the development one may want to reuse as much as possible of the test set of the previous system version, and focus test generation only on specific parts of the system behavior. This may result in significantly faster test generation under most iterative development scenarios.

The incremental approach in the field of testing has first been introduced by El-Fakih et. al. [9]. The method generates test cases only for the modified parts of the system, but it is not capable of maintaining a complete checking sequence across changes. The other important aspect is, that it uses traditional algorithms to create test sequences for a given FSM upon each modification. Therefore its complexity is the function of the size of the input machine, not the extent of change.

In this paper we introduce two bounded incremental algorithms to automatically create test sequences in response to changes applied to the system specification. The two algorithm operate in a similar manner, but have different purpose. The first one maintains a prefix-closed state cover set used to reach all states of the finite state machine, while the second one keeps a separating family of sequences up to date responsible to verify the next state of transitions. The complexity of the proposed algorithms are evaluated based on the bounded incremental model of computation of Ramalingam and Reps [19]. It is shown that the time complexity of our incremental algorithms depends on the extent of changes to the specification rather than the size of the specification itself. It is also shown, that it is never worse than the complexity of the most relevant, traditional algorithm – the HIS-method [20] [25] [18].

The rest of the paper is organized as follows. A short overview of our assumptions and notations are given in Section 2. The relevant FSM test generation algorithms are also briefly discussed. Section 3 gives a quick overview of the incremental approach, alongside its application in testing. In Section 4 we introduce our incremental algorithms to maintain a checking sequence across changes, demonstrate them through an example and provide analytic and practical analyzes of their complexity. Finally, in Section 5 we conclude the paper.

2 Preliminaries

2.1 Finite State Machines

Finite State Machines (FSMs) have been widely used for decades to model systems in various areas, such as sequential circuits [10], communication protocols [13], some types of programs [1] (in lexical analysis, pattern matching etc.) and object-oriented software testing [2]. Several specification languages, such as SDL [14] [24] [21] and ESTELLE [23] [24], are extensions of the FSM formalism.

A Finite State Machine (FSM) M is a quadruple $M = (I, O, S, T)$ where I, O, and S are the finite and nonempty sets of input symbols, output symbols and states, respectively. T is the finite set of transitions between states. Each transition

$t \in T$ is a quadruple $t = (s_j, i, o, s_k)$, where $s_j \in S$ is the start state, $i \in I$ is an input symbol, $o \in O$ is an output symbol and $s_k \in S$ is the next state. The number of states and inputs denoted by $n = |S|$ and $p = |I|$, respectively.

A machine can be represented with a state transition graph, a directed edge-labeled graph whose vertices are labeled with the state symbols of the machine and whose edges correspond to the state transitions. Each edge is labeled with the input and the output, written i/o, associated with the transition.

Consider FSM M with $|S| = n$ states, and implementation $Impl$ with at most n states. A *spanning tree* of FSM M rooted from the initial state s_0 is an acyclic subgraph (a partial FSM) of its state transition graph composed of all the reachable vertices (states) and some of the edges (transitions) of M such that there is exactly one path from s_0 to any other state.

FSM M is *deterministic* if for each (s_j, i) state-input pair there exists at most one transition in T. If there is at least one transition $t \in T$ for all state-input pairs, the machine is said to be *completely specified*. In case of deterministic and completely specified FSMs, each (s_j, i) state-input pair defines exactly one transition, which can be given as $t = (s_j, i, \lambda(s_j, i), \delta(s_j, i))$, where λ: $S \times I \rightarrow O$ denotes the *output function* and δ: $S \times I \rightarrow S$ denotes the *next state function* [16].

FSM M has a *reset capability* if there is an input symbol $r \in I$ that takes the machine from any state back to the $s_0 \in S$ initial state. That is, $\exists r \in I : \forall s_j \in S : \delta(s_j, r) = s_0$. The *reset is reliable* if it is guaranteed to work properly in any implementation machine M^I, i.e., $\delta^I(s_j^I, r) = s_0^I$ for all states $s_j^I \in S^I$, and s_0^I is the initial state of M^I. Note, that a machine with reliable reset capability is strongly connected iff for each state $s_j \in S$ is reachable from the $s_0 \in S$ initial state.

For a given set of symbols Σ, Σ^* is used to denote the set of all finite sequences over Σ. Let $K \subseteq \Sigma^*$ be a set of sequences over Σ. The prefix closure of K – denoted by $\boldsymbol{Pref}(K)$ – includes all the prefixes of all sequences in K. The set K is *prefix-closed* if $\boldsymbol{Pref}(K) = K$.

We extend the next state function δ and output function λ from input symbols to finite input sequences I^* as follows: For a state s_1, an input sequence $x = i_1, ..., i_k$ takes the machine successively to states $s_{j+1} = \delta(s_j, i_j), j = 1, ..., k$ with the final state $\delta(s_1, x) = s_{k+1}$, and produces an output sequence $\lambda(s_1, x) = o_1, ..., o_k$, where $o_j = \lambda(s_j, i_j), j = 1, ..., k$. A machine M is *strongly connected* iff for each pair of states (s_j, s_l), there exists an input sequence which takes M from s_j to s_l.

Two states, s_j and s_l are *distinguishable* iff there exists an $x \in I^*$ input sequence that produces different output: $\lambda(s_j, x) \neq \lambda(s_l, x)$. Such an x input sequence is called a *separating sequence* of the two inequivalent states s_j and s_l. Otherwise – if such sequence does not exist – we say that states s_j and s_l are *equivalent* (undistinguishable), i.e. $s_j \cong s_l$ iff for all input sequences $x \in I^*$, $\lambda(s_j, x) = \lambda(s_l, x)$. A machine is *reduced*, if no two states are equivalent, that is, each pair of states (s_j, s_l) are distinguishable. State minimization is a transformation into an equivalent, reduced state machine with removing equivalent states.

In the paper, we consider strongly connected, completely specified, reduced and deterministic FSMs with reliable reset capability.

2.2 Modification of FSMs

In [17] a consistent approach is proposed to represent changes in deterministic FSMs. In this paper atomic changes are achieved by using edit operators. An *edit operator* turns FSM $M = (I, O, S, T)$ into FSM $M' = (I, O, S', T')$ with the same (labeled alike) input and output sets. For deterministic finite state machines two types of edit operators have been proposed based on widely accepted fault models. An *output change operator* is $\omega_o(s_j, i, o_x, s_k) = (s'_j, i, o_y, s'_k)$, where $\lambda(s_j, i) = o_x \neq o_y = \lambda'(s'_j, i)$. A *next state change operator* is $\omega_n(s_j, i, o_x, s_k) = (s'_j, i, o_x, s'_l)$, where $\delta(s_j, i) = s_k \neq s'_l = \delta'(s'_j, i)$. Multiple changes ω^* can be generated with the concatenation of the two edit operators defined above. This paper focuses on such multiple changes. It has been shown in [17], that with some assumptions the set of deterministic FSMs with a given number of states is closed under the proposed edit operations. Furthermore, for any two deterministic FSMs M_1 and M_2 there is always a sequence of edit operations changing M_1 to M_2, i.e., to a machine isomorphic to M_2.

2.3 FSM-based Test Generation

In FSM-based testing both the implementation and the specification can be modelled as an FSM. The given implementation can be considered as a black box with unknown behavior. *Conformance testing* checks if an $Impl$ implementation conforms to a given M specification: the input/output pairs of *test sequences* are derived from FSM M, then the input test sequences are applied to the machine $Impl$ and the output of $Impl$ is observed. The observed output sequence of $Impl$ is compared to the expected results derived from FSM M. If the output sequences of $Impl$ and M do not match, it implies that the implementation has a fault.

Given a completely specified deterministic FSM M with n states, a *checking sequence* for M is an input sequence x that distinguishes M from all other machines with n states. That is, any implementation machine $Impl$ with at most n states not equivalent to M produces an output different from M on x.

Many algorithms have been introduced to create checking sequence for FSMs with reliable reset capability [16] [3]. Such processes are the W-method [5], the Wp-method [11] and the HIS-method [20] [25] [18]. All of these algorithms have a similar structure containing two main parts. The first – state identification – part checks for each state of the specification whether it exists in the implementation as well. The second – transition testing – part checks all remaining transitions of the implementation by observing whether the output and the next state conforms to the specification. In the following we concentrate on the HIS-method as it is the most general approach of the three.

2.4 The HIS-Method

Let Q denote a *prefix-closed state cover set* $Q = \{q_1, ..., q_n\}$ responsible to reach all states of FSM M. A Q set may be created by constructing a spanning tree of

the state transition graph of the specification machine M from the initial state s_0. Such a spanning tree is presented on Figure 3(a) in Section 4.1. A prefix-closed state cover set Q is the concatenation of the input symbols on all partial paths of the spanning tree, i.e., sequences of input symbols on all consecutive branches from the root of the tree to a state.

Let $Z = \{Z_1, ..., Z_n\}$ denote the *separating family of sequences* [25] of FSM M used to verify the next state of transitions. A separating family of sequences of FSM M is a collection of n sets $Z_i, i = 1, ..., n$ of sequences (one set for each state) satisfying the following two conditions: For every pair of states s_i, s_j: (I) there is an input sequence x that separates them, i.e., $\exists x \in I^*$, $\lambda(s_i, x) \neq \lambda(s_j, x)$; (II) x is a prefix of some sequence in Z_i and some sequence in Z_j. Z_i is called the separating set of state s_i. Such family may be constructed for a reduced FSM the following way: For any pair of states s_i, s_j we generate a sequence z_{ij} that separates them using for example a minimization method [15]. Then define the separating sets as $Z_i = \{z_{ij}\}, j = 1...n$. The HIS-method uses appropriate members of the separating family in both stages of the algorithm to check states of the implementation.

The first, state identification stage of the HIS-method generates test sequences $r.q_i.Z_i$, $i = 1...n$, where r is the reliable reset symbol and "." is the string concatenation operator.

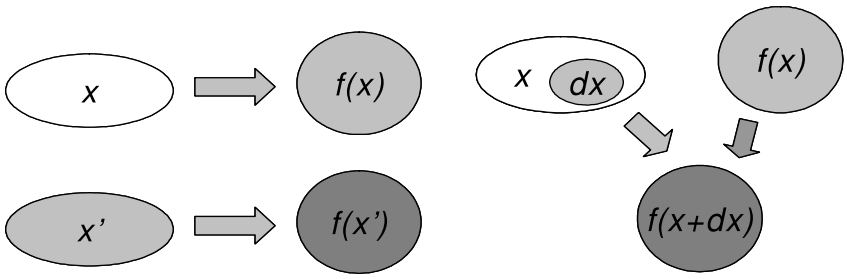
If $Impl$ passes the first stage of the algorithm for all states, then we know that $Impl$ is similar to M, furthermore this portion of the test also verifies all the transitions of the spanning tree. The second, transition testing stage is used to check non-tree transitions. That is, for each transition (s_j, i, o, s_k) not in the spanning tree the following test sequences are generated: $r.q_j.i.Z_k$.

The resulting sequence is a checking sequence, starting at the initial state and consisting of no more than $p \cdot n^2$ test sequences of length less than $2 \cdot n$ interposed with reset [25]. Thus the total complexity of the algorithm is $O(p \cdot n^3)$, where $p = |I|$ and $n = |S|$.

3 The Incremental Approach

3.1 Batch and Incremental Algorithms

The *batch algorithm* for a given problem determine the solution $f(x)$ on some input x. If input x changed into x', the batch algorithm generates the new output $f(x')$ from scratch – see Figure 1(a). The *incremental algorithm* computes the required output $f(x')$ as a batch algorithm. However, it assumes that the same problem has been solved already on a slightly different input x providing output $f(x)$, and that input x has been modified by changes dx since, i.e., $x + dx = x'$. The incremental algorithm uses the previous input x, along with the changes dx, and the previous output $f(x)$ to compute the new output $f(x + dx) = f(x')$ – see Figure 1(b). A batch algorithm can be used as an incremental algorithm, furthermore, in case of a fundamental change the batch algorithm will be the most efficient incremental algorithm.



(a) Batch algorithm (b) Incremental algorithm

Figure 1: Batch and incremental algorithms

3.2 Evaluating the Complexity of an Incremental Algorithm

The complexity of a given algorithm is usually expressed by the maximum number of steps required as a function of the size of the input. Although this worst-case analysis may be adequate in case of batch algorithms, it is not very informative in case of incremental algorithms. Instead of analyzing the complexity as a function of the entire input the [19] paper propose to use an adaptive parameter, which capturing the extent of the changes in the input and output. The parameter Δ represents the size of the *modified* part of the input and the size of the *affected* part of the previous output. Thus Δ represents the minimal amount of work necessary to calculate the new output. The complexity of incremental algorithm can be analyzed in terms of Δ, which is not known a priori, but calculated during the update process. This approach will be used in the current paper to evaluate the complexity of the proposed algorithm and to compare it to existing methods.

3.3 Incremental Test Generation and Incremental Testing

Virtually, all traditional FSM-based conformance test generation algorithms [6] [3] [16] [4] [7] are *batch* algorithms. These approaches are therefore incapable of utilizing any auxiliary information, such as existing tests created for the previous version of the given system. All test sequences have to be created from scratch in each evolution step, no matter how small the change has been. This approach handles resources quite inefficiently, thus incremental approach should be applied.

In case of conformance testing two different incremental approach problem could be considered: *incremental test generation* and *incremental testing*.

Incremental test generation algorithms create test cases for the whole system: both for parts that are affected and for parts that are non-affected by changes. These algorithms utilizes existing test sequences to generate new test sequences for the updated system version. As a result, an incremental test generation algorithm may generate the solution for the given problem significantly faster than a batch algorithm, while it provides the same fault detection capability. In case of software testing, when the new code is compiled and tested only few times the complexity

of test generation is not negligible.

Incremental testing algorithms, however show the parts of the system, which test cases are affected by the given changes, and create new test cases only for these parts. This approach reduce the time required for testing, while all of the parts affected by the given changes are investigated.

In Section 4 we propose two algorithms, which both support incremental test generation (maintain a complete checking sequence) and incremental testing (create test sequences only for the affected parts of the specification).

4 Incremental Generation of a Checking Sequence

In this section we propose two incremental algorithms to maintain a checking sequence through series of changes. We assume that the system specification is given as a reduced, completely specified, deterministic FSM M, that is modified by ω^* sequence of changes. We also assume that a checking sequence for M – the specification machine before changes – is given in form of the HIS-method. Our purpose is to create a new, valid checking sequence for the modified specification FSM M'.

Similarly to the HIS-method, our solution contains two parts: the first one called State Cover Set Maintenance (SCSM, presented in Section 4.1) maintains a prefix-closed state cover set responsible to reach all states of the machine, while the second one, called State Identification Maintenance (SIM, presented in Section 4.2) maintains a Separating Family of Sequences used to verify the next state of transitions.

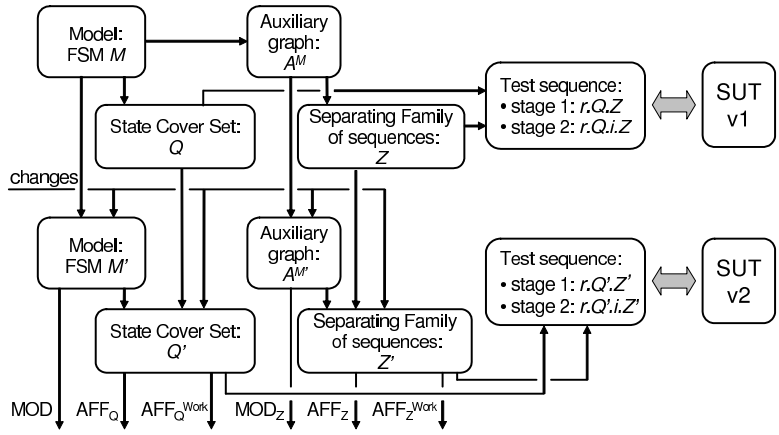


Figure 2: The block diagram of the incremental maintenance of HIS-method test cases

The high level view of our approach is presented in Figure 2. The upper part of the figure represents the traditional generation of the HIS test sequence: the Q state cover set and the Z separating family of sequences are generated, then a

checking sequence is created by the concatenation of these subsequences to apply for the system under test (SUT). Auxiliary graph A^M denotes a graph over state pairs of FSM M, which helps the generation of a Z separating family of sequences. The lower part of the figure shows how the sequence of changes affect the generation of HIS-method's test cases. The changes turn FSM M into FSM M'. Instead of repeating the previous operations on M', the SCSM algorithm (proposed in Section 4.1) utilizes a Q prefix-closed state cover set of the previous version of the system to create a valid Q' state cover set of M'. The SIM algorithm (introduced in Section 4.2) maintains a Z separating family of sequences across changes in order to produce a valid Z' separating family of sequences of M'. The SIM algorithm also utilizes and incrementally maintains the A^M auxiliary graph of the specification in order to generate Z' efficiently. Then these sequences are concatenated into a checking sequence and applied for the modified system (SUT v2).

The two algorithms show which test sequences are affected by the given sequence of changes – denoted by AFF_Q and AFF_Z – so the whole system or only the affected part of the system can be investigated depending on the testing purpose. The modified part of the FSM M' and auxiliary graph $A^{M'}$ could be also observed, denoted by MOD and MOD_Z, respectively. It has to be emphasized that a given change to the specification FSM may affect the SCSM and SIM algorithms differently. Therefore two separate Δ parameters (see Section 3.2) have to be used to capture the extent in which the changes affect the two algorithms, i.e. $\Delta_Q = |MOD \cup AFF_Q|$, $\Delta_Z = |MOD_Z \cup AFF_Z|$.

As our approach involves two completely autonomous incremental algorithms, the SCSM and SIM algorithm may also be applied independently for various purposes. For instance, they could be used to detect undesirable effects of a planned modification during development, such as subsets of states becoming unreachable – denoted by AFF_Q^{Work} – or equivalent – denoted by AFF_Z^{Work}.

4.1 An algorithm to maintain a Prefix-closed State Cover Set

In this section we propose an algorithm, called SCSM, to maintain a prefix-closed state cover set Q of specification machine M through a given ω^* sequence of changes, that turns FSM M into M'.

The problem can be reduced to maintaining a spanning tree ST of the state transition graph of FSM M rooted from the initial state s_0 (see Section 2.4), where ST representing the Q set of M. Our objective is to create a new valid spanning tree ST', which corresponds the Q' set FSM M'.

We use the following notations in the description of the SCSM algorithm. A transition is called an ST-transition iff it is in ST. A subtree of ST rooted from a state s_i is denoted by ST_{s_i}. The MOD set collects the *modified* states: $s_x \in MOD$ iff a transition originating from state s_x of FSM M is modified by a change. State s'_y is called *q-affected* state – collected in set AFF_Q – iff s'_y of FSM M' is affected by a change with respect to the Q set. Note, that if a state is q-affected a new test sequence has to be generated for it. A work set is also used to keep track of

affected states, for which the new test sets have not yet been generated – denoted with AFF_Q^{Work}. An adaptive parameter is also defined to capture the extent of changes in the input and the output: $\Delta_Q = |MOD \cup AFF_Q|$; the complexity of the SCSM algorithm will be expressed as a function of Δ_Q.

Our SCSM algorithm has two different phase. The first one determines the q-affected states of FSM M', while the second one extends the spanning forrest for these states, to create a valid Q set of the modified FSM M'.

4.1.1 Detailed description of SCSM algorithm

Algorithm 1: Maintaining a Prefix-closed State Cover Set

input : M; ω^*; $ST = (V, E)$;
output: $ST' = (V', E')$; MOD; AFF_Q; AFF_Q^{Work}; M';

1 `data`(M; ω^*; $ST = (V, E)$; $ST' = (V', E')$; MOD; AFF_Q; AFF_Q^{Work}; M'; $ST'_{temp} = (V'_{temp}, E'_{temp})$)
2 $M' := M$; $V' := V$; $E' := E$; $MOD := \emptyset$; $AFF_Q := \emptyset$; $AFF_Q^{Work} := \emptyset$;
`/* Phase 1: Apply changes to` M `and identify affected states */`
3 **for** *next* $\omega(s_m, i, o, s_j)$ *in* ω^* **do**
4 $M' :=$Apply $\omega(s_m, i, o, s_j)$ change to M'; $MOD = MOD \cup \{s'_m\}$
5 **if** $(\delta(s_m, i) \neq \delta'(s'_m, i))$`and`$((s_m, i, o, s_j) \in E')$ **then**
6 $AFF_Q := AFF_Q \cup$ `TreeWalk`$(ST', s_j).V$;
7 $AFF_Q^{Work} := AFF_Q^{Work} \cup$ `TreeWalk`$(ST', s_j).V$;
8 $ST' := ST' \setminus ((s_m, i, o, s_j) \cup$ `TreeWalk`$(ST', s_j))$;
`/* Phase 2: Create new test cases for affected states */`
9 **foreach** $s'_j \in AFF_Q^{Work}$ **do**
10 **foreach** $t = (s'_i, i, o, s'_j) \in T$ **do**
11 **if** $s'_i \notin AFF_Q^{Work}$ **then**
12 $ST'_{temp}(V'_{temp}, E'_{temp}) :=$ `SpanningTree`$((AFF_Q^{Work}, M'), s'_j)$; $E' := E' \cup \{(s'_i, s'_j)\} \cup E'_{temp}$; $V' := V' \cup \{s'_j\} \cup V'_{temp}$;
13 $AFF_Q^{Work} := AFF_Q^{Work} \setminus (\{s'_j\} \cup V'_{temp})$;
14 **return** $ST' \in (V', E')$, MOD, AFF_Q, AFF_Q^{Work}, M';

The pseudo-code of the SCSM algorithm is presented in Algorithm 1.

The input of the algorithm is the original machine M, the ω^* sequence of changes and the spanning tree ST of M.

After initialization (line 2), the SCSM algorithm goes through the given ω^* sequence of changes. First it applies an $\omega(s_m, i, o, s_j) \in \omega^*$ change to the state transition graph and marks modified states (line 4). After the change is applied, the algorithms identifies the sets of affected states (lines 5-8). The output change operator only changes an edge label of the state transition graph of the FSM, but does not affects its structure. That is, $\delta(s_m, i) = \delta'(s'_m, i)$, and the change does not affect any states with respect to the Q set, so AFF_Q and AFF_Q^{Work} sets are not extended. Next state changes, however affects states iff the changed transition was

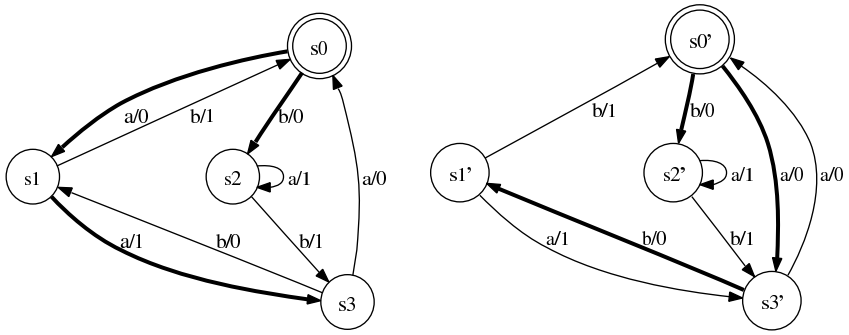
an ST'-transition (line 5). In that case all states of the $ST'_{s'_j}$ subtree with s'_j root are put into the AFF_Q and AFF_Q^{Work} sets and all edges of the $ST'_{s'_j}$ subtree with s'_j root are removed from ST'. Function $TreeWalk$ walks on a subtree of the given ST' spanning tree with a given root using simple breath-first search.

After the affected states have been determined, the Phase 2 of algorithm extend the ST' spanning tree (lines 9-13) for AFF_Q^{Work} elements using breadth-first search implemented in function $SpanningTree$. The algorithm does it in the following way: For each state s'_j in the AFF_Q^{Work} set it checks if there is a transition leading to s'_j from an s'_i state, which state is not in the AFF_Q^{Work} set. If this transition has been found, it is added to ST', s'_j is removed from AFF_Q^{Work} and all states of M' that can be reached from s'_j by function $SpanningTree$ are removed from AFF_Q^{Work}, while the transitions used to reach these states are added to ST'.

As the SCSM algorithm terminates spanning tree ST' represent the updated Q set of the modified FSM M'. MOD denotes the states, where the machine has been modified, AFF_Q represents the states, where the Q-set has been changed and the states, which still remain in the AFF_Q^{Work} set, are unreachable states of FSM M'.

4.1.2 SCSM algorithm example

We demonstrate with an example how our SCSM algorithm maintains a prefix-closed state cover set across a change. We use the following notations in the figures: the bold edges represent spanning tree edges and the double circle denotes the initial state.



(a) FSM M with its spanning tree ST

(b) Modified FSM M' with its updated spanning tree ST'

Figure 3: SCSM algorithm example

Take FSM M on Figure 3(a). Initially let $ST' = ST$, $MOD = \emptyset$, $AFF_Q = \emptyset$ and $AFF_Q^{Work} = \emptyset$. The modification $\omega_n(s_0, a, 0, s_1) = (s'_0, a, 0, s'_3)$ is a next state change: $MOD = \{s'_0\}$. As transition $(s_0, a, 0, s_1)$ is in ST, we need to determine the set of *q-affected* states by walking the $ST'_{s'_1}$ subtree. We get $AFF_Q =$

$\{s'_1, s'_3\}$, $AFF_Q^{Work} = \{s'_1, s'_3\}$. In Phase 2 transitions leading to *q-affected* states – $(s'_0, a, 0, s'_1)$ and $(s'_1, a, 1, s'_3)$ – are removed from ST'. Then we check for each state in AFF_Q^{Work} if there is a transition from a non-AFF_Q^{Work} state. Transition $(s'_0, a, 0, s'_3)$ is identified, which is a link originating from non-AFF_Q^{Work} state s'_0. We add it to ST' and remove s'_3 from AFF_Q^{Work}. We then check transitions originating from s'_3 and find $(s'_3, b, 0, s'_1)$ that leads to an AFF_Q^{Work} state. We add $(s'_3, b, 0, s'_1)$ to ST' and remove s'_1 from AFF_Q^{Work}. Now, $AFF_Q^{Work} = \emptyset$, so the algorithm terminates and returns $MOD = \{s'0\}$, $AFF_Q = \{s'_1, s'_3\}$, $AFF_Q^{Work} = \emptyset$ and ST', see Figure 3(b).

4.2 An algorithm to maintain a Separating Family of Sequences

In this section we propose an algorithm, called SIM, to maintain a separating family of sequences Z of specification machine M through a given sequence of changes ω^*, that turns FSM M into M'.

To maintain a separating family of sequences, the SIM algorithm has to identify all state pairs, which separating sequences affected by changes. Then for all affected state pairs a new separating sequence has to be generated. As the next state verification problem deals with state pairs we define an auxiliary directed graph A^M with $n(n+1)/2$ nodes, where each node corresponds an unordered pair $< s_k, s_l >$ of states of M including identical state pairs $< s_k, s_k >$. A^M has a directed edge labelled with input symbol i from $< s_k, s_l >$ to $< s_m, s_n >$ iff $\delta(s_k, i) = s_m$ and $\delta(s_l, i) = s_n$ in M. The auxiliary directed graph A^M is used to represent and maintain separating sequences of FSM M. Graph A^M is updated by our algorithm at each incremental step; the modified auxiliary graph denoted by $A^{M'}$.

A state pair $< s_x, s_y >$ called as a *separating state pair* iff $\lambda(s_x, i) \neq \lambda(s_y, i)$ for some $i \in I$, where i is the *separating input* – in other words a separating state pair has a separating input, which gives a different output. FSM M is reduced, iff there exist a path in its auxiliary graph A^M from each non-identical state pair $< s_k, s_l >, k \neq l$ to a separating state pair $< s_x, s_y >$. The input labels from $< s_k, s_l >$ to $< s_x, s_y >$ concatenated by the separating input of $< s_x, s_y >$ form a separating sequence of states s_k and s_l.

In order to efficiently maintain the Z set of FSM M we make the following assumptions on the separating sequences of Z: (I) Each separating state pair $< s_x, s_y >$ has a single separating input $i | \lambda(s_x, i) \neq \lambda(s_y, i)$ associated to it. If a given separating state pair has multiple such inputs, then a proper input is chosen randomly. (II) The set of separating sequences of FSM M is prefix-closed. Note, that these requirements do not restrict the generality of our SIM algorithm as it generates a separating family of sequences Z' at each step, such that Z' fulfills the required assumptions.

If the assumptions defined above hold, then the separating sequences of FSM M can be represented with a spanning forest SF over the non-identical state pairs of A^M, such that each tree has a separating state pair as root and all edges of the

given tree are directed toward the root – see Figure 4(a). That is, the problem of generating the separating family of sequences Z' for the modified FSM M' can be reduced to maintaining separating state pairs, their associated separating input and a forest SF' over non-identical state pairs of $A^{M'}$ across changes.

We use the following notations in the description of the algorithm. An edge of A^M is called an SF-edge iff it is in SF. A subtree of SF rooting from state pair $< s_i, s_j >$ is denoted by $SF_{<s_i,s_j>}$. The MOD_Z set collects the *modified* state pairs of the modified auxiliary graph $A^{M'}$: $< s'_i, s'_j > \in MOD_Z$ iff a transition originating from state s_i or s_j of FSM M is modified by a change. Note, that in case of a change modified a transition originated from state s_x, then all state pairs, that involving s'_x are modified in $A^{M'}$. State pair $< s'_x, s'_y >$ are said to be *z-affected* – collected in set AFF_Z – iff $< s'_x, s'_y >$ of auxiliary graph $A^{M'}$ is affected by a change with respect to the Z set, i.e., a new separating sequence of states s'_x and s'_y has to be generated. A work set is also used to keep track of affected state pairs, for which the new test sets have not yet been generated – denoted with AFF_Z^{Work}. We also defined an adaptive parameter to capture the extent of changes in the input and the output: $\Delta_Z = |MOD_Z \cup AFF_Z|$; the complexity of the SIM algorithm will be expressed as a function of Δ_Z.

As the previous SCSM algorithm, the SIM algorithm also consists two phases. The first phase determines z-affected state pairs and destroy invalid test cases, while the second one expand the spanning forrest SF' to create a valid Z' set for FSM M'.

4.2.1 Detailed description of SIM algorithm

The pseudo-code of the SIM algorithm is presented in Algorithm 2.

The input of the algorithm is the auxiliary directed graph A^M of FSM M, the ω^* sequence of changes and the spanning forest SF of A^M.

After initialization (line 2), the SIM algorithm goes through the given ω^* sequence of changes. First it applies an $\omega(s_m, i, o, s_j)$ change to the auxiliary graph (line 4) and marks modified state pairs (line 5). It implies that for each change $\omega(s_m, i, o, s_j)$ in ω^* corresponding edges of $A^{M'}$ are modified, i.e., edges marked with input i originating from all state pairs involving s'_m. After a change is applied, the algorithm identifies the set of affected state pairs (lines 6-25). Function $TreeWalk$ walks on a subtree of a given SF' forest with a given root. Note that by walking such a subtree we always assume that it is explored opposing edge directions from the root toward leaves. The algorithm considers next state and output changes as well. In case of an output change, the algorithm checks the following conditions: (I) If i has been the separating input of separating state pair $< s_m, s_i >$ in A^M, but $\lambda'(s'_m, i) = \lambda'(s'_i, i) = o_y$ and if there is another input i_1 such that $\lambda'(s'_m, i_1) \neq \lambda'(s'_i, i_1)$ (line 10), then $< s'_m, s'_i >$ remains a separating state pair with i_1 associated as separating input. State pairs of the tree of SF' with $< s'_m, s'_i >$ root – including the $< s'_m, s'_i >$ vertex – are added to AFF_Z, but not added to AFF_Z^{Work}. Note that the algorithm preserves the original tree structure with the $< s'_m, s'_i >$ root in SF' in this specific modification scenario. (II) If i has

Algorithm 2: Maintaining a Separating Family of Sequences – 1^{st} phase

input : $A^M = (V, E)$, $V = S \times S$; ω^*;
$SF = (V_{SF}, E_{SF}), V_{SF} \subseteq V, E_{SF} \subseteq E$;
output: $SF' = (V'_{SF}, E'_{SF})$; MOD_Z; AFF_Z; AFF_Z^{Work}; $A^{M'}$

1 data($A^M = (V, E)$; ω^*; $SF = (V_{SF}, E_{SF})$; $SF' = (V'_{SF}, E'_{SF})$; MOD_Z; AFF_Z; AFF_Z^{Work}; $A^{M'}_{AFF} = (V'_{AFF}, E'_{AFF})$; $ST'_{temp} = (V'_{temp}, E'_{temp})$)
2 $A^{M'} := A^M$; $V'_{SF} := V_{SF}$; $E'_{SF} := E_{SF}$; $MOD_Z := \emptyset$;
$AFF_Z^{Work} := AFF_Z := \emptyset$;
3 **for** *next* $\omega(s_m, i, o, s_j)$ *in* ω^* **do**
4 $A^{M'}$:=Apply $\omega(s_m, i, o, s_j)$ change to $A^{M'}$; extend MOD_Z;
5 **foreach** (s'_i, i) **do** $MOD_Z = MOD_Z \cup \{< s'_m, s'_i >\}$;
/* In case of an output change */
6 **if** $\lambda(s_m, i) \neq \lambda'(s'_m, i)$ **then**
7 **foreach** s'_i **do**
8 **if** $\lambda(s_m, i) \neq \lambda(s_i, i)$ **then**
9 **foreach** $i_1 \in I$ **do**
10 **if** $\lambda'(s'_m, i_1) \neq \lambda'(s'_i, i_1)$ **then**
11 $< s'_m, s'_i >$ remains a *sep. state pair* with i_1 *sep. input*
12 $AFF_Z := AFF_Z \cup$ TreeWalk$(SF', < s'_m, s'_i >).V$;
13 **if** $\forall i_1 \in I, \lambda'(s'_m, i_1) = \lambda'(s'_i, i_1)$ **then**
14 *separating state pair* mark removed from $< s'_m, s'_i >$
$AFF_Z := AFF_Z \cup$ TreeWalk$(SF', < s'_m, s'_i >).V$;
$AFF_Z^{Work} =: AFF_Z^{Work} \cup$ TreeWalk$(SF', < s'_m, s'_i >).V$;
15 $SF' := SF' \setminus$ TreeWalk$(SF', < s'_m, s'_i >)$;
16 **if** $(\lambda(s_m, i) = \lambda(s_i, i))and(\lambda'(s'_m, i) \neq \lambda'(s'_i, i))$ **then**
17 $E'_{SF} := E'_{SF} \setminus \{(< s'_m, s'_i >, < s'_x, s'_y >)\}$;
18 $< s'_m, s'_i >$ marked as a *separating state pair* with i *sep. input*
19 $AFF_Z := AFF_Z \cup$ TreeWalk$(SF', < s'_m, s'_i >).V$;
/* In case of a next state change */
20 **if** $\delta(s_m, i) \neq \delta'(s'_m, i)$ **then**
21 **foreach** s'_i **do**
22 **if** $\exists s'_j, s'_k : (< s'_m, s'_i >, i, o, < s'_j, s'_k >) \in E'_{SF}$ **then**
23 $AFF_Z := AFF_Z \cup$ TreeWalk$(SF, < s'_m, s'_i >).V$;
24 $AFF_Z^{Work} := AFF_Z^{Work} \cup$ TreeWalk$(SF, < s'_m, s'_i >).V$;
25 $SF' := SF' \setminus$ TreeWalk$(SF, < s'_m, s'_i >).V$;

been the separating input of separating state pair $< s_m, s_i >$ in A^M, and there is no another separating input from $< s'_m, s'_i >$ (line 13), then the separating state pair mark removed from $< s'_m, s'_i >$. State pairs of the tree of SF' with $< s'_m, s'_i >$ root including $< s'_m, s'_i >$ are added to AFF_Z and AFF_Z^{Work}. The edges and state pairs of the tree of SF' with $< s'_m, s'_i >$ root including $< s'_m, s'_i >$ are removed from SF'. (III) If a state pair $< s'_m, s'_i >$ is a new separating state pair (line 16), then the

Algorithm 2: Maintaining a Separating Family of Sequences – 2^{nd} phase

```
   /* Create subgraph A^{M'}_{AFF} for affected state pairs */
26 V'_{AFF} := 0; E'_{AFF} := 0
27 foreach v'_i ∈ AFF_Z^{Work} do
28     V'_{AFF} := V'_{AFF} ∪ {v'_i}; foreach (v'_i, v'_j) do
29         if v'_j ∈ AFF_Z^{Work} then E'_{AFF} := E'_{AFF} ∪ (v'_i, v'_j)
30         if v'_j ∉ AFF_Z^{Work} then v'_i marked with v'_j
   /* Extend SF' Spanning forest for affected state pairs */
31 foreach v'_i marked with v'_j do
32     ST'_{temp} = (V'_{temp}, E'_{temp}) := SpanningTree((A^{M'}_{AFF}), v'_i);
       E'_{SF} := E'_{SF} ∪ {(v'_i, v'_j)} ∪ E'_{temp}; V'_{SF} := V'_{SF} ∪ V'_{temp};
       AFF_Z^{Work} := AFF_Z^{Work} \ V'_{temp};
33 return SF' ∈ (V'_{SF}, E'_{SF}), MOD_Z, AFF_Z, AFF_Z^{Work}, A^{M'};
```

edge originating from the node corresponding to $< s'_m, s'_i >$ in SF' is deleted and the state pair is marked with separating state pair and i is marked as separating input. In the next step all state pairs of the subtree of SF' with $< s'_m, s'_i >$ root including $< s'_m, s'_i >$ are added to AFF_Z, but not added to AFF_Z^{Work}. Note that the algorithm preserves the original subtree structure with the $< s'_m, s'_i >$ root in SF'. In case of an $\omega_n(s_m, i, o, s_j) = (s'_m, i, o, s'_l), s_j \neq s'_l$ next state change that effect an edge of the spanning forest SF' (line 22), all state pairs of the subtree of SF' with (s'_m, s'_i) root including $< s'_m, s'_i >$ are added to AFF_Z and AFF_Z^{Work}. Furthermore, the edges and state pairs of the tree with $< s'_m, s'_i >$ root – including the $< s'_m, s'_i >$ vertex – are removed from SF'.

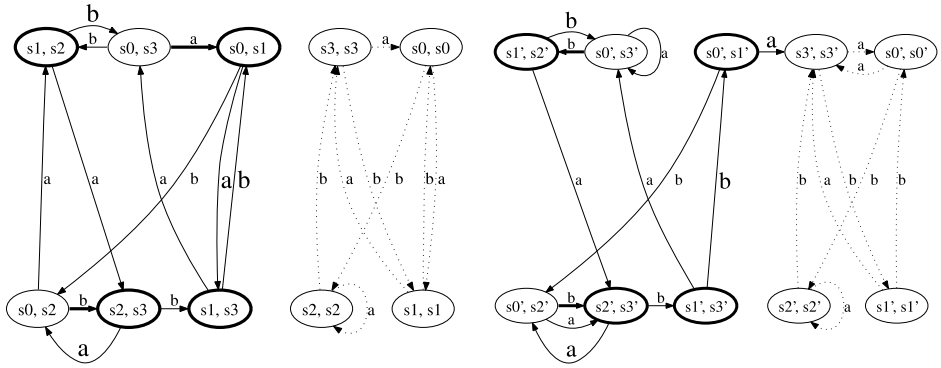
After the affected state pairs have been determined, the Phase 2 of the algorithm creates new test cases for AFF_Z^{Work} state pairs (lines 26-32). The algorithm does it in the following way: It creates an $A^{M'}_{AFF}$ subgraph (lines 26-30) for AFF_Z^{Work} state pairs and marked state pairs, which have an edge originated to a non AFF_Z^{Work} state pair. Finally, the algorithm adds these edges to SF' and extends the SF' spanning forest from these marked AFF_Z^{Work} state pairs (lines 31-32) using breadth-first search implemented in function *SpanningTree*.

As the SIM algorithm terminates spanning forest SF' represent the updated Z set of the modified FSM M'. MOD_Z denotes the state pairs, where the auxiliary graph $A^{M'}$ has been modified, AFF_Z represents the state pairs, where the Z-set have been changed. The proposed SIM algorithm is able to detect if the original assumption on minimized machine do not hold after the given sequence of changes applied: the state pairs, which still remain in AFF_Z^{Work}, are equivalent pairs of states.

4.2.2 SIM algorithm example

The auxiliary graph A^M of M is presented on Figure 4(a). Bold edges represent the forest SF of M, separating state pairs are shown in bold ellipses, separating

inputs are represented by bigger sized edge labels, while the dotted edges between identical state pairs are maintained but have no importance for the algorithm.



(a) The auxiliary graph A^M of FSM M with its spanning forrest

(b) The updated auxiliary graph $A^{M'}$ of FSM M' with its new spanning forrest

Figure 4: SIM algorithm example

Initially $AFF_Z^{Work} = AFF_Z = \emptyset$, $MOD_Z = 0$ and let $SF' = SF$. Edges labeled with input a originating from state pairs $< s'_0, s'_0 >$, $< s'_0, s'_1 >$, $< s'_0, s'_2 >$, $< s'_0, s'_3 >$ are modified to create $A^{M'}$: $MOD_Z = \{< s'_0, s'_0 >, < s'_0, s'_1 >, < s'_0, s'_2 >, < s'_0, s'_3 >\}$. The $< s'_0, s'_1 >$ state pair is a separating state pair and is therefore not affected. The a-labeled edge originating from state pair $< s_0, s_2 >$ is not in SF thus $< s'_0, s'_2 >$ is not affected either. $< s'_0, s'_0 >$ is irrelevant. Therefore only state pair $< s'_0, s'_3 >$ is *z-affected*: $AFF_Z = \{< s'_0, s'_3 >\}, AFF_Z^{Work} = \{< s'_0, s'_3 >\}$. In Phase 2 the a-labeled edge originating from $< s'_0, s'_3 > \in AFF_Z^{Work}$ is removed from SF'. Then edges originating from $< s'_0, s'_3 >$ are checked and an edge $\langle < s'_0, s'_3 >, < s'_1, s'_2 > \rangle$ leading to a non-AFF_Z^{Work} state pair is found. The given edge is added to SF' and $< s'_0, s'_3 >$ is removed from AFF_Z^{Work}. Now, $AFF_Z = \emptyset$, thus the algorithm terminates and returns $MOD_Z = \{< s'_0, s'_0 >, < s'_0, s'_1 >, < s'_0, s'_2 >, < s'_0, s'_3 >\}$, $AFF_Z = \{< s'_0, s'_3 >\}$, $AFF_Z^{Work} = \emptyset$ and SF', see Figure 4(b). All separating sequences are unchanged except the one of states s'_0 and s'_3, which is changed from $a.a$ to $b.b$.

4.3 Complexity calculation

4.3.1 Complexity calculation of the SCSM algorithm

In case of a unit change operator $|MOD| = 1$, while in case of multiple changes $1 \leq |MOD| \leq n$, while $0 \leq |AFF_Q| \leq n$.

In Phase 1 the SCSM algorithm updates the FSM M into M' and identifies affected states. The update of the FSM through ω^* changes requires $O(|MOD|)$ steps. To identify the affected states with breath-first search and to put them into the AFF_Q and AFF_Q^{Work} sets has a complexity of $O(|AFF_Q|)$.

In Phase 2 the SCSM algorithm first searches transitions from non-AFF_Q^{Work} states to AFF_Q^{Work} states. There are exactly $p \cdot |AFF_Q^{Work}|$ transitions originating from the AFF_Q^{Work} states. Therefore there can be at most $p \cdot |AFF_Q^{Work}| \leq p \cdot |AFF_Q|$ backward check steps, which do not provide a path from a non-AFF_Q^{Work} state to an AFF_Q^{Work} state. Thus, in worst case there are $(p+1) \cdot |AFF_Q|$ backward check turns. If a transition is found from a non-AFF_Q^{Work} to an AFF_Q^{Work} state s'_j then all states of AFF_Q^{Work} reachable from s'_j via AFF_Q^{Work} states are removed and ST' are extended. As there are exactly $p \cdot |AFF_Q^{Work}|$ transitions originating from AFF_Q^{Work} states, this requires maximum $p \cdot |AFF_Q^{Work}| \leq p \cdot |AFF_Q|$ steps.

As any of the $p \cdot |AFF_Q|$ transitions are processed at most twice by the algorithm, less than $2 \cdot (p+1) \cdot |AFF_Q| \approx O(p \cdot |AFF_Q|)$ steps are necessary to complete Phase 2. As $\Delta_Q = |MOD \cup AFF_Q|$, the total complexity of SCSM algorithm is $O(p \cdot \Delta_Q)$, where $1 \leq \Delta_Q \leq n$.

4.3.2 Complexity calculation of the SIM algorithm

In case of a unit change operator $|MOD_Z| = n$, while in case of multiple changes $n \leq |MOD_Z| \leq n \cdot (n-1)/2$, while $0 \leq |AFF_Z| \leq n \cdot (n-1)/2$.

In Phase 1 the SIM algorithm updates auxiliary graph A^M into $A^{M'}$ and identifies affected state pairs. The update of the auxiliary graph through ω^* changes requires $O(|MOD_Z|)$ steps. To identify the affected states with breath-first search and to put them into the AFF_Z and AFF_Z^{Work} sets has a complexity of $O(|AFF_Z|)$.

In Phase 2 the SIM algorithm first creates subgraph $A^{M'}_{AFF}$ for AFF_Z^{Work} state pairs and marked state pairs, which have an edge leading to a non-AFF_Z^{Work} state pair in $p \cdot |AFF_Z|$ steps. To extend spanning forest SF' for AFF_Z^{Work} elements requires $O(p \cdot |AFF_Z|)$ complexity. As $\Delta_Z = |MOD_Z \cup AFF_Z|$, the total complexity of the algorithm is $O(p \cdot \Delta_Z)$, where $n \leq \Delta_Z \leq n^2$.

4.3.3 Overall complexity

Our incremental algorithms – as the HIS-method – create a checking sequence by concatenating test sequences from the Q and Z sets. A test sequence of the checking sequence must be regenerated after changes if either its q-sequence or its z-sequence is changed by the proposed algorithms. That is, a test sequence $r \cdot q_i \cdot \ ... \ \cdot z_{ij}$ of M is modified iff $s'_i \in AFF_Q$ or $(s'_i, s'_j) \in AFF_Z$. Thus, the total complexity of incremental test generation is $O(p \cdot \Delta_Q \cdot n^2) + O(n \cdot p \cdot \Delta_Z) \approx O(p \cdot \Delta \cdot n)$, where $p = |I|$, $1 \leq \Delta_Q \leq n$, $n \leq \Delta_Z \leq n^2$ and $n \leq \Delta \leq n^2$ denotes the number of test sequences that have to be modified. That is, in worst case the number of test cases to be generated is equivalent to those generated by a batch algorithm with $O(p \cdot n^3)$ complexity. Note that, the algorithms shows the modified part of the test cases to allow complete testing or testing only the affected part of the system. Also note that the concatenation of the distinct parts of the test sequences only has to be performed as a part of the testing procedure. If no actual testing is needed after changes the algorithms should only maintain the two separate part of the HIS

method: the state cover set with $O(p \cdot \Delta_Q)$ and the separating family of sequences with $O(p \cdot \Delta_Z)$ complexity.

4.4 Experiments

Before the in-depth analysis of experimental results we show some implementation details and the architecture of the simulation environment including how random input FSMs are produced.

4.4.1 Implementation details

We implemented the traditional HIS-method and our algorithms as well in JAVA. In order to make incremental algorithms as efficient as possible, it is essential to create data structures that ensure that all fundamental steps can be realized with the least possible elementary steps. Such elementary steps are condition evaluation, data reading and writing. The other main objective is to assure that the space complexity is as little as possible. The data structure must be efficient both in the representation and in maintenance of the specification graph and its test cases. To assure these purposes we developed the following data structures for our algorithms.

The SCSM algorithm stores the specification graph of the FSM in a matrix, in which rows correspond to the s states and columns correspond to the i inputs, thus the size of the matrix is $n \times p$. Each element of this matrix is an edge of the graph and a transition of the specification FSM M. We store for each transition its s next state and o output label and an ST flag, that denotes whether the transition is an ST-transition or not. For each transition there are also p_{prev} and p_{next} pointers to the previous and to the next transitions which have the same next state. These pointers create bidirectional chains each one associated to a given state. A pointer to the first element of each chains stored in a vertex denoted FNS, i.e the kth element of the FNS vertex will point to the first transition which leads to state s_k. In case of find a transition with a given s_x start state and a given i_y input (for example to modify this edge, see Algorithm 1, line 4), we only need to read the xth row and yth column of the matrix, in case of breadth-first search from a s_x state we can iterate over the row corresponds to s_x, and in case of inverse breadth-first search from a s_x state we can recursively go through the chain originated from the xth element of the FNS array. The fast inverse breadth-first search is essential in both algorithms – the SCSM algorithm uses, when it goes through the AFF_Q^{Work} set in order to search an edge, that is not originated from this set (Algorithm 1, line 10), and the SIM algorithm uses, when it build an ST'_{temp} subtree through the inverse direction of the edges (Algorithm 2, line 32) – and the bidirectional chains defined above ensure to both of them.

The SIM algorithm uses the same data structure as the SCSM one, but instead of the specification graph, it uses that for the auxiliary graph, so the matrix rows denote state pairs instead of states. Each element of this matrix is a 5-tuple ($< s_a, s_b >, sep, SF, p_prev, p_next$), where $< s_a, s_b >$ is the next state pair label, sep and SF are flags that denotes, whether it is a *separating state pair* and if the edge

is in the spanning forest or not. The p and n are previous and next pointers to the bidirectional chains, that collect edges with the same next state pair labels.

4.4.2 Simulation Environment

The incremental algorithms utilize the previous version of test cases generated initially by batch algorithms - as shown previously in Figure 2. To be able to compare batch and incremental algorithms, test cases for the modified machine M' are generated with the batch method as well.

We use random FSMs as input for our test generation algorithms. The generator program creates completely specified, deterministic, strongly connected FSMs with reliable reset capability. The program ensures the strongly connected property the following way: First a tree is created with a vertex for each state and then reset transitions are added from each vertex to the root. The tree is used as a spanning tree of the FSM being generated. The root vertex is denoted with s_0 and all other vertices are being randomly labeled with a state label. Each edge of the tree is then labeled with random inputs and outputs. Finally, for each remaining state-input pair not used in the tree generation a transition is created with random next state and random output label. These steps fulfill the completely specified and deterministic assumptions. The random graph generator does not ensure the generation of reduced FSMs. However, this is not a constraint in our case, because the SIM algorithm (see Section 4.2) is able to detect whether an FSM contains equivalent states.

The complexity calculation of incremental algorithms considers all atomic operations used by the implementation such as data access and condition evaluation. The traditional HIS-method is implemented in the same way therefore the same elementary steps are used for the complexity calculation as in case of the incremental algorithms, so they can be directly compared to each other.

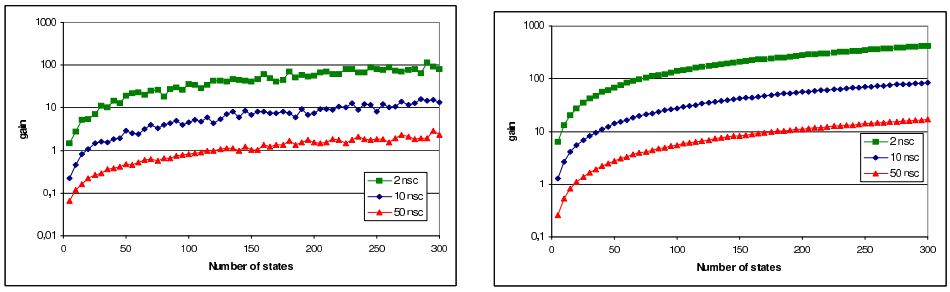
4.5 Simulation results

In the following we analyze how our SCSM and SIM algorithms perform against the traditional HIS-method. We investigate six different scenarios in order to observe our algorithms for different $|S|$, $|I|$, and $|O|$ combinations for different type of changes.

4.5.1 Next state changes

First, we investigate next state changes. In Scenario 1 $|I|$ and $|O|$ are fixed to 5 in order to observe the influence of $|S|$ to the simulation parameters in case of 2, 10 and 50 changes. Each data in the figures is obtained by averaging 10 simulation runs.

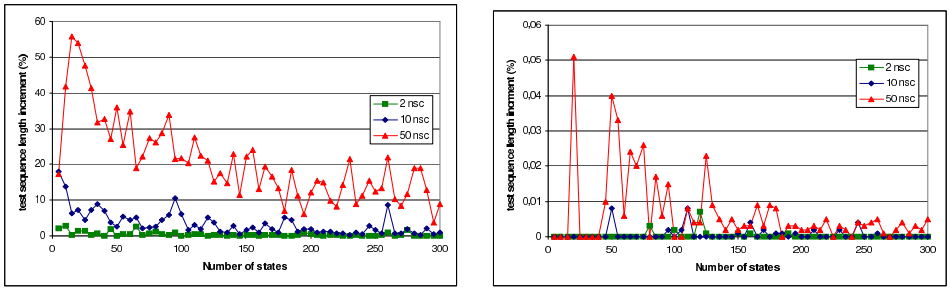
We define the variable *Gain*, which describes the gain of our incremental algorithms compared to the traditional HIS-method and it is calculated as follows:



(a) The gain of the SCSM algorithm (b) The gain of the SIM algorithm

Figure 5: The gain of the algorithms in Scenario 1

$Gain = \frac{Step_{trad_{HIS}}}{Step_{inc_{HIS}}}$, where $Step_{trad_{HIS}}$ is the number of steps in case of the traditional HIS-method, and $Step_{inc_{HIS}}$ is the number of steps in case of our novel, incremental HIS-method. Thus, a higher $Gain$ value corresponds to a better performance compared to the traditional approach. For instance $Gain = 100$ means that our incremental HIS-method requires $1/100th$ steps than the traditional HIS-method. The $Gain$ can be defined for the generation of Q-set and Z-set, independently; the two distinct part of the traditional and incremental HIS-method. The result for the SCSM algorithm is presented in Figure 5(a). The $Gain$ of the SIM algorithm compared to the traditional HIS-method corresponding part is shown in Figure 5(b).

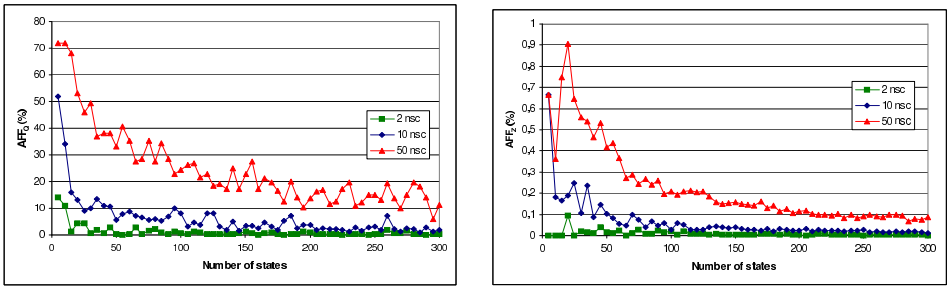


(a) SCSM algorithm (b) SIM algorithm

Figure 6: The increment of the length of the test sequences in Scenario 1

The cost for this high gain is that our algorithms does not guarantee to find test sequences with the shortest possible length. Figures 6(a) and 6(b) shows the difference from the optimal solution in case of the SCSM and SIM algorithm, respectively. This difference in case of the SCSM algorithm is not significant because as Figure 7(a) shows, the left side of the Figure represents a situation near worst case: most part of the machine is affected by changes. As Figure 6(b) indicate, the increment of test sequence length much less in case of the SIM algorithm, than the

SCSM one: in case of SCSM algorithm the difference from the optimal solution is negligible.



(a) Size of AFF_Q vs. number of states (b) Size of AFF_Z vs. number of states

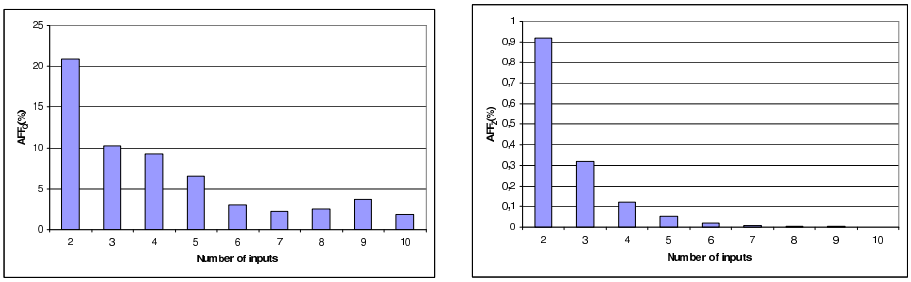
Figure 7: The relative size of affected elements in Scenario 1

The reason is that next state changes affect the separating family of sequences much less, than the state cover set – see Figures 7(a) and 7(b). The ratio of affected elements decreases in function of the number of states. The number of affected elements is much greater in case of the SCSM algorithm than the SIM one. The reason is the following: The SCSM algorithm maintains a spanning tree with one root and $n-1$ transition[1], while the SIM algorithm maintains a spanning forest with several roots and smaller trees. Therefore, a next state change has greater probability in case of the SCSM algorithm to affect a bigger sub-spanning tree, than in case of the SIM one. In case of the SIM algorithm there are several separating state pairs, therefore the test set contains a spanning forest with a relatively large number of trees, each with low depth. A given tree may typically contains only a separating state pair or a separating state pair with few state pairs referring to it. We also observe the deepest trees in our simulations: in case of 10 next state changes the most depth of a tree was only 2, and it occurred only in 26 case in all of our 600 simulations. The number of affected elements shows, where the original test set have been changed. In case of incremental testing, only this part of the machine need to retest after the given sequences of changes applied, thus testing time can be significantly reduced.

We also investigate how the number of input symbols influences the increment of the test sequence length and the ratio of affected elements. In Scenario 2 $|S|$ and $|0|$ is fixed to 100 and 5, respectively, while the number of changes is set to 10. Each data in the figure is obtained by averaging 50 simulation runs.

Our simulations show that both algorithms have less affected elements if the number of input symbols is higher – see Figure 8(a) and Figure 8(b). It is in line with our expectations, because if there are more transitions originating from a state it is less likely that changes affect a transition used in the test set (as an edge of the ST spanning tree or the SF spanning forest).

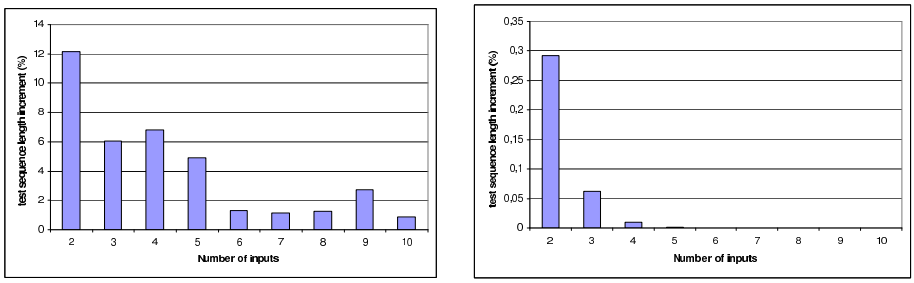
[1]Each state, except the initial state has exactly one spanning edge originated from.



(a) Size of AFF_Q vs. input symbols

(b) Size of AFF_Z vs. input symbols

Figure 8: The relative size of affected elements in Scenario 2



(a) Increment of SCSM algorithm's test sequences length vs. input symbols

(b) Increment of SIM algorithm's test sequences length vs. input symbols

Figure 9: The increment of the length of the test sequences in Scenario 2

As a result the difference from the optimal solution decreases as the number of input symbols increases for both the SCSM – see Figure 9(a) – and for the SIM algorithm – see Figure 9(b).

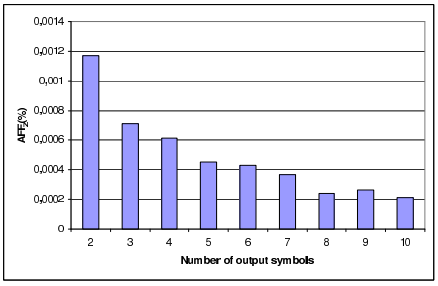


Figure 10: The relative size of AFF_Z in Scenario 3

Another interesting aspect, that how the number of output symbols influences the ratio of affected state pairs and the increment in the test sequences length. Only

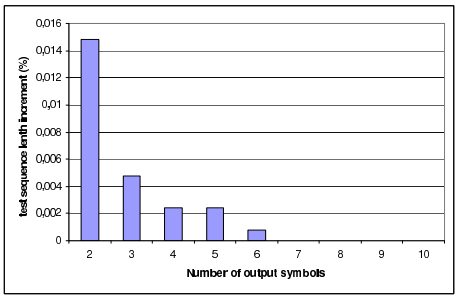


Figure 11: The increment of the SIM algorithm's test sequence length in Scenario 3

the SCSM algorithm is impacted by the number of output symbols, because the SIM one simply does not do anything with the output label of the transitions – it only finds traces among them despite of theirs outputs. Therefore only the relative size of the AFF_Z set in function of output symbols is discussed. In Scenario 3 $|S|$ and $|I|$ are set to 100 and 5, respectively, while the number of changes is set 10. Each data is obtained by averaging 50 simulation runs. The results show that if the number of output symbols increases, the number of affected elements decreases – see Figure 10. It is in line with our expectations, because if there are more possible output, there are more separating state pairs, which causes smaller trees with less edges, thus the probability that a next state change affects one of them is decreases. For the same reason, the increment of test sequence length also decreases as $|O|$ increases – see Figure 11 – although the difference from optimal solution is remains negligible for all observed parameters.

4.5.2 Output changes

We also investigate the case, when FSM M is modified by output changes. Output changes only affect the SIM algorithm, thus we concentrate on that. In Scenario 4, $|I|$ and $|O|$ are fixed to 5 in order to observe the influence of $|S|$ to the simulation parameters in case of 2, 10 and 50 changes. Each data in the figures is obtained by averaging 10 simulation runs.

The *Gain* of the SCSM algorithm is increases as the number of states – see Figure 12. The Gain is higher than the one observed in Scenario 1. The reason for this is that the update of auxiliary graph requires a $|\omega^*| \cdot n$ "fixed cost", which is smaller in case of output changes (in case of next state changes, the SIM algorithm also has to update the bidirectional chains, that collect edges with same next states, see Section 4.4.1).

The increment of the length of test sequences is also observed – see Figure 13. The difference from the optimal solution is negligible even in case of fundamental changes, and it decreases rapidly as the number of states increases.

The relative size of affected state pairs against the number of the states is pre-

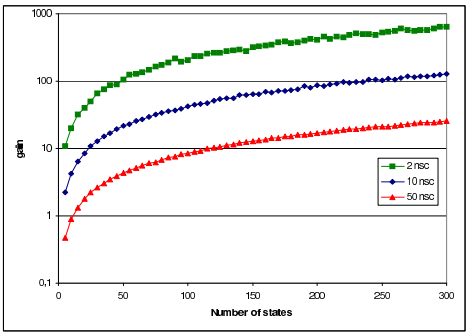


Figure 12: The gain of the SIM algorithm in Scenario 4

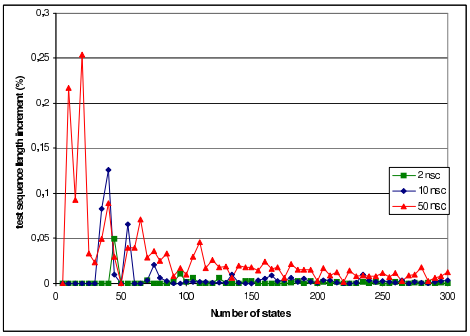


Figure 13: The increment of the length of the test sequences in Scenario 4

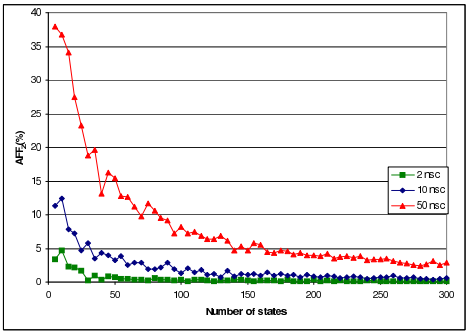
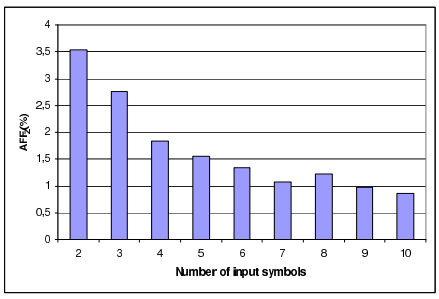


Figure 14: The relative size of AFF_Z vs. number of states in case of five output changes

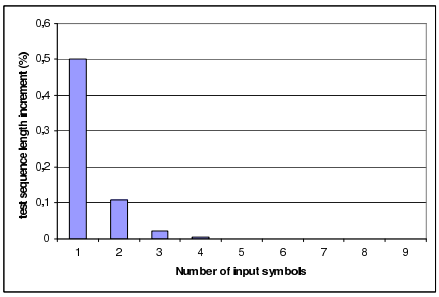
sented in Figure 14. The results indicate that, the ratio of affected state pairs decreases as the number of states increases, but output changes affects the separating family of sequences much more, than next state changes. Nevertheless, at

and above 85 states, the ratio of affected state pairs was below 10 percent even in case of 50 changes, thus incremental testing time can be significantly reduced.

We also investigate how the number of input and output symbols influences the increment of the test sequence length and the ratio of affected elements. In Scenario 5 $|0|$ is fixed to 5 and in Scenario 6 $|I|$ is fixed to 5. $|S|$ and number of output changes is set to 100 and 10, respectively in both cases and each data is obtained by averaging 50 simulation runs.



(a) Size of AFF_Z vs. input symbols

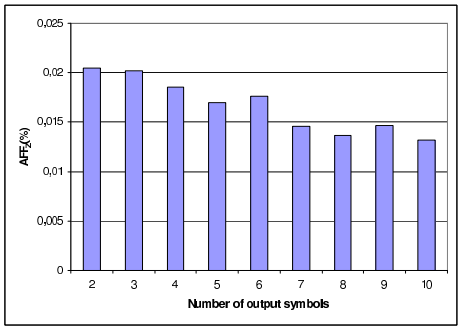


(b) Increment of test sequences length vs. input symbols

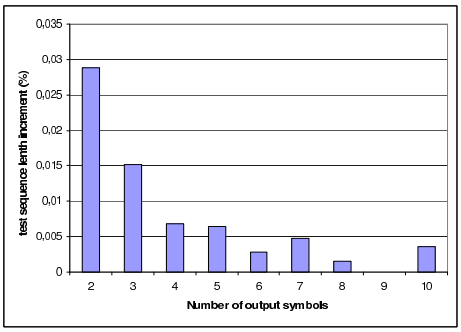
Figure 15: Results of Scenario 5 simulations

First, Scenario 5 will be discussed. Figure 15(a) shows that, the number of affected state pairs length decreases as the number of input symbols increases. This meets with our expectations, because if a state pair has fewer edges originated from it, there is a bigger probability that a change affect an edge, which is used in a test set. There are more affected state pairs than in case of next state changes (presented previously in Figure 8(b)), because the spanning forest contains only few edges, but has many separating state pairs associated with a separating input. Output changes may affect separating inputs, while next state changes don't. The increment of test sequence's length is also observed. Figure 15(b) shows that, the increment of the length of test sequences decreases as the number of input symbols increases, and the difference from the optimal solution is much more pronounced than in case of next state changes (presented previously in Figure 9(b)), although it still remains insignificant.

Finally, Scenario 6 will be discussed. Figure 16(a) shows that, the number of affected state pairs length decreases as the number of input symbols increases, but there are more affected state pairs than in case of next state changes (presented previously in Figure 10). The reason for the decreasing is the same as discussed in next state changes, while the cause of more affected state pairs compared to next state changes is discussed previously in Scenario 5. Figure 16(b) shows that, the increment in the length of test sequences decreases as the number of input symbols increases, and the difference from the optimal solution is much more pronounced than in case of next state changes (presented previously in Figure 11), although it remains insignificant.



(a) Size of AFF_Z vs. output symbols



(b) Increment of test sequences length vs. output symbols

Figure 16: Results of Scenario 6 simulations

5 Conclusion

We have presented bounded incremental algorithms to maintain the test sequences for finite state machine models across updates. The algorithms use the test cases of a relevant traditional algorithm, the HIS-method. Our approach assumes a changing specification and utilizes an existing test set of the previous version to keep a complete test set efficiently up-to-date. For each update of the system a complete test set is generated incrementally with the same fault detection capability as that of the traditional HIS-method. Both analytical calculations and experiments indicate that our approach is very efficient in case of large, changing specifications. Therefore, our algorithms are serviceable in real testing environments – such as software testing – through development phase, because they provide much faster test generation than the traditional approach.

Besides incremental test generation (that maintain a complete checking sequence) our algorithms also support incremental testing to create test cases only for the affected parts of the specification. The incremental testing property of our algorithms reduce the time required for testing, while all of the parts affected by the given changes can be investigated. Our simulations have shown relatively few affected elements in the original test set even in case of fundamental changes, thus testing time can be significantly reduced.

Furthermore, our solution uses two autonomous incremental algorithms, which may be also applied independently for various purposes – such as checking if a subsets of states becoming equivalent or unreachable during the development phase.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Binder, Robert V. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] Bochmann, G. V. and Petrenko, A. Protocol testing: review of methods and relevance for software testing. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124, New York, NY, USA, 1994. ACM Press.

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

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

[6] Dahbura, A.T., Sabnani, K.K., and Uyar, M.U. Formal methods for generating protocol conformance test sequences. *Proceedings of the IEEE*, 78(8):1317 – 1326, August 1990.

[7] Dorofeeva, Rita, El-Fakih, Khaled, Maag, Stephane, Cavalli, Ana R., and Yevtushenko, Nina. Fsm-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*, 52(12):1286 – 1297, 2010.

[8] Dove, R. *Response Ability - The Language, Structure and Culture of the Agile Enterprise.* John Wiley and Sons, 2001.

[9] El-Fakih, Khaled, Yevtushenko, Nina, and von Bochmann, Gregor. FSM-based incremental conformance testing methods. *IEEE Transactions on Software Engineering*, 30(7):425–436, 2004.

[10] Friedman, A. D. and Menon, P. R. *Fault Detection in Digital Circuits.* Prentice-Hall, 1971.

[11] Fujiwara, S., v. Bochmann, G., Khendec, F., Amalou, M., and Ghedamsi, A. Test selection based on finite state model. *IEEE Transactions on Software Engenieering*, 17:591–603, 1991.

[12] Highsmith, J.A. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* Dorset House, 2000.

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

[14] ITU-T. Recommendation Z.100: Specification and description language, 2000.

[15] Kohavi, Z. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1978.

[16] Lee, D. and Yannakakis, M. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[17] Pap, Z., Csopaki, G., and Dibuz, S. On the theory of patching. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods, SEFM*, pages 263–271, 2005.

[18] Petrenko, Alexandre, Yevtushenko, Nina, Lebedev, Alexandre, and Das, Anindya. Nondeterministic state machines in protocol conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 363–378, 1994.

[19] Ramalingam, G. and Reps, Thomas. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1-2):233–277, 1996.

[20] Sabnani, K. and Dahbura, A. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15(4):285–297, 1988.

[21] Sinnott, Richard O. and Hogrefe, Dieter. Finite state machine based: SDL. In *Formal methods for distributed processing*, pages 55–76. Cambridge University Press, New York, NY, USA, 2001.

[22] Smith, P.G. and Reinertsen, D.G. *Developing Products in Half the Time: New Rules, New Tools.* John Wiley and Sons, 1998.

[23] TC97/SC21, ISO. Estelle – a formal description technique based on an extended state transition model. international standard 9074, 1988.

[24] Turner, Kenneth J. *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[25] Yannakakis, Mihalis and Lee, David. Testing finite state machines: fault detection. In *Selected papers of the 23rd annual ACM symposium on Theory of computing*, pages 209–227, 1995.

Received ...