A Bounded Incremental Test Generation Algorithm for Finite State Machines

Zoltán Pap[1], Mahadevan Subramaniam[2], Gábor Kovács[3], Gábor Árpád Németh[3]

[1] Ericsson Telecomm. Hungary, H-1117 Budapest, Irinyi J. u. 4-20, Hungary
zoltan.pap@ericsson.com
[2] Computer Science Department, University of Nebraska at Omaha
Omaha, NE 68182, USA
msubramaniam@mail.unomaha.edu
[3] Department of Telecommunications and Media Informatics – ETIK,
Budapest University of Technology and Economics,
Magyar tudósok körútja 2, H-1117, Budapest, HUNGARY
kovacsg@tmit.bme.hu, rubrika@gmail.com

Abstract. We propose a bounded incremental algorithm to generate test cases for deterministic finite state machine models. Our approach, in contrast to the traditional view, is based on the observation that system specifications are in most cases modified incrementally in practice as requirements evolve. We utilize an existing test set available for a previous version of the system to efficiently generate tests for the current – modified – system. We use a widely accepted framework to evaluate the complexity of the proposed incremental algorithm, and show that it is a function of the size of the change in the specification rather than the size of the specification itself. Thus, the method is very efficient in the case of small changes, and never performs worse than the relevant traditional algorithm – the HIS-method. We also demonstrate our algorithm through an example.

Keywords: conformance testing, finite state machine, test generation algorithms, incremental algorithms

1 Introduction

Large, complex systems continuously evolve to incorporate new features and new requirements. In each evolution step – in addition to changing the specification of the system and producing a corresponding implementation – it may also be necessary to modify the testing infrastructure. Manual modification is an ad hoc and error prone process that should be avoided, and automatic specification-based test generation methods should be applied.

Although testing theory is especially well developed for finite state machine (FSM)-based system specifications, existing algorithms handle changing specifications quite inefficiently. Most research has been focusing on the analysis of rigid, unchanging descriptions. Virtually all proposed methods rely solely on a

given specification machine to generate tests. 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.

In this paper we develop a novel, bounded incremental algorithm to automatically re-generate tests in response to changes to a system specification. In its essence the algorithm maintains two sets incrementally; a prefix-closed state cover set responsible for reaching all states of the finite state machine, and a separating family of sequences applied to verify the next state of transitions. The complexity of the algorithm is evaluated based on the bounded incremental model of computation of Ramalingam and Reps [1]. It is shown that the time complexity of the proposed algorithm depends on the size of the change to the specification rather than the size of the specification itself. Furthermore, it is never worse than the complexity of the most traditional algorithm – the HIS-method [2] [3] [4].

This research builds on our earlier work in [5] where we have developed a framework to analyze the effects of changes on tests based on the notion of consistency between tests and protocol descriptions. In the current paper, we have extended our focus to the test generation problem, which is a major step both in terms of complexity and practical importance.

The rest of the paper is organized as follows. A brief overview of our assumptions and notations is given in Section 2. In Section 3, we describe some relevant FSM test generation algorithms and the HIS-Method in particular. Section 4 describes the model of incremental computation. In Section 5 we introduce the incremental algorithm for maintaining a checking sequence across changes, provide a thorough analysis of its complexity and demonstrate it through an example. Sections 6 and 7 describe related work and conclusions, respectively.

2 Finite State Machines

Finite state machines have been widely used for decades to model systems in various areas. These include sequential circuits [6], some types of programs [7] (in lexical analysis, pattern matching etc.), and communication protocols [8]. Several specification languages, such as SDL [9] and ESTELLE [10], are extensions of the FSM formalism.

A finite state machine M is a quadruple $M = (I, O, S, T)$ where I is the finite set of input symbols, O is the finite set of output symbols, S is the finite set of states, and $T \subseteq S \times I \times O \times S$ is the finite set of (state) transitions. Each transition $t \in T$ is a 4-tuple $t = (s_j, i, o, s_k)$ consisting of start state $s_j \in S$, input symbol $i \in I$, output symbol $o \in O$ and next state $s_k \in S$.

An FSM can be represented by a state transition graph, a directed edge-labeled graph whose vertices are labeled as the states of the machine and whose edges correspond to the state transitions. Each edge is labeled with the input and output associated with the transition.

FSM M is said to be deterministic if for each start state – input pair (s, i) there is at most one transition in T. In the case of deterministic FSMs both the output and the next state of a transition may be given as a function of the start state and the input of the transition. These functions are referred to as the next state function δ: $S \times I \rightarrow S$ and the output function λ: $S \times I \rightarrow O$. Thus a transition of a deterministic machine may be given as $t = (s_j, i, \lambda(s_j, i), \delta(s_j, i))$.

For a given set of symbols A, A^* is used to denote the set of all finite sequences (words) over A. Let $K \subseteq A^*$ be a set of sequences over A. The prefix closure of K, written $\boldsymbol{Pref}(K)$, 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 $o_j = \lambda(s_j, i_j), j = 1, ..., k$. The input/output sequence $i_1 o_1 i_2 o_2 ... i_k o_k$ is then called a trace of M.

FSM M is said to be strongly connected if, for each pair of states (s_j, s_l), there exists an input sequence which takes M from s_j to s_l. If there is at least one transition $t \in T$ for all start state – input pairs, the FSM is said to be completely specified (or completely defined); otherwise, M is a said to be partially specified or simply a partial FSM.

We say that machine M has a reset capability if there is an initial state $s_0 \in S$ and an input symbol $r \in I$ that takes the machine from any state back to s_0. 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; otherwise it is unreliable.

Finite state machines may contain redundant states. State minimization is a transformation into an equivalent state machine to remove redundant states. Two states are equivalent written $s_j \cong s_l$ iff for all input sequences $x \in I^*$, $\lambda(s_j, x) = \lambda(s_l, x)$. Two states, s_j and s_l are distinguishable (inequivalent), iff $\exists x \in I^*$, $\lambda(s_j, x) \neq \lambda(s_l, x)$. Such an input sequence x is called a separating sequence of the two inequivalent states. A FSM M is reduced (minimized), if no two states are equivalent, that is, each pair of states (s_j, s_l) are distinguishable.

For the rest of the paper, we focus on strongly connected, completely specified and reduced deterministic machines with reliable reset capability. We will denote the number of states and inputs by $n = |S|$ and $p = |I|$, respectively.[4]

2.1 Representing Changes to FSMs

Although the FSM modeling technique has been used extensively in various fields, impact of changes on FSM models and their effects on test sets have only been studied recently following the observation that system specifications are in most cases modified incrementally in practice as requirements evolve. (See some of our earlier papers [11] [12] [5]).

[4] Therefore, $|T| = p * n$.

A consistent approach for representing changes to FSM systems has been proposed in [12]. Atomic changes to a finite state machine M are represented by the means of edit operators $\omega^M : T \rightarrow T$.[5] An edit operator turns FSM $M = (I, O, S, T)$ into FSM $M' = (I, O, S', T')$ with the same input and output sets. We use the term "same states" written $s_j = s'_j$ for states that are labeled alike in different machines. Obviously, these states are not necessarily equivalent, written $s_j \cong s'_j$.

For deterministic finite state machines two types of edit operators have been proposed based on widely accepted fault models. A next state change operator is $\omega_n(s_j, i, o_x, s_k) = (s'_j, i, o_x, s'_l)$, where $\delta(s_j, i) = s_k \neq s'_l = \delta'(s'_j, i)$. An output change operator is $\omega_o(s_j, i, o_x, s_k) = (s'_j, i, o_y, s'_k)$, where $\lambda(s_j, i) = o_x \neq o_y = \lambda'(s'_j, i)$. It has been shown in [12], that with some assumptions the set of deterministic finite state machines with a given number of states is closed under the edit operations defined above. 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.

3 FSM Test Generation and the HIS-Method

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

Several algorithms have been proposed to generate checking sequences for machines with reliable reset capability [13] [14], including the W-method [15], the Wp-method [16] and the HIS-method [2] [3] [4]. They all share the same fundamental structure consisting of two stages: Tests derived for the first – state identification – stage check that each state presented in the specification also exists in the implementation. Tests for the second – transition testing – stage check all remaining transitions of the implementation for correct output and ending state as defined by the specification. The methods, however, use different approaches to identify a state during the first stage, and to check the ending state of the transitions in the second stage. In the following we concentrate on the HIS-method as it is the most general approach of the three.

The HIS-method derives a *family of harmonized identifiers* [4], also referred to as a *separating family* of sequences [3]. 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. The HIS-method uses appropriate members of the separating family in both stages of the algorithm to check states of the implementation.

[5] $\omega(t)$ is used instead of $\omega^M(t)$ if M can be omitted without causing confusion.

3.1 The HIS-Method

Consider FSM M with $|S| = n$ states, and implementation $Impl$ with at most n states. Let $Z = \{Z_1, ..., Z_n\}$ be a separating family of sequences of FSM M. 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 [17]. Then define the separating sets as $Z_i = \{z_{ij}\}, j = 1...n$.

The state identification stage of the HIS-method requires a prefix-closed state cover set $Q = \{q_1, ..., q_n\}$ of FSM M, and generates test sequences $r \cdot q_i \cdot Z_i$, $i = 1...n$ based on it, where r is the reliable reset symbol and "$\cdot$" is the string concatenation operator. A Q set may be created by constructing a spanning tree[6] of the state transition graph of the specification machine M from the initial state s_0. Such a spanning tree is presented on Figure 1(a) in Section 5.1. 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.

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: $r \cdot q_j \cdot i \cdot Z_k$.

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

4 Incremental Computation Model

A *batch* algorithm for a given problem is an algorithm capable of computing the solution of the problem $f(x')$ – the output – on some input x'. Virtually all traditional FSM-based conformance test generation algorithms [13] [14] are such *batch* algorithms. Their input is the specification of a system in form of an FSM model and the output is a checking sequence that is (under some assumptions) capable of determining if an implementation conforms to the specification FSM.

An incremental algorithm intends to solve a given problem by computing an output $f(x')$ just as a batch algorithm. Incremental computation, however, assumes that the same problem has been solved previously on a slightly different input x providing output $f(x)$, and that the input has undergone some changes since, resulting in the current input $x + dx = x'$. An incremental algorithm takes the input x and the output $f(x)$ of the previous computation, along with

[6] A spanning tree of FSM M rooted from the initial state 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 the initial state s_0 to any other state.

the change in the input dx. From that it computes the new output $f(x + dx)$, where $x + dx$ denotes the modified input. A batch algorithm can be used as an incremental algorithm, furthermore, in case of a fundamental change (take $x = null$ input for example) the batch algorithm will be the most efficient incremental algorithm.

4.1 Evaluating the Complexity of an Incremental Algorithm

The complexity of an algorithm is commonly evaluated using asymptotic worst-case analysis; by expressing the maximum cost of the computation as a function of the size of the input. While this approach is adequate for most batch algorithms, worst-case analysis is often not very informative for incremental algorithms. Thus, alternative ways have been proposed in the literature to express the complexity of incremental algorithms. The most widely accepted approach has been proposed in [1]. Instead of analyzing the complexity of incremental algorithms in terms of the size of the entire current input, the authors suggest the use of an adaptive parameter capturing the extent of the changes in the input and output. The parameter Δ or "$CHANGED$" 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 is analyzed in terms of Δ, which is not known a priori, but calculated during the update process. This approach will be used in this paper to evaluate the complexity of the presented algorithm and to compare it to existing batch and incremental methods.

5 Incremental Test Generation Method

This section presents a novel incremental test generation algorithm. The algorithm – in contrast to traditional (batch) test generation methods – is capable of maintaining a checking sequence across changes in the specification, thus avoiding the need of regenerating a checking sequence from scratch at each stage of an incremental development.

We focus on the following problem: Consider a system specification given as a reduced, completely specified and deterministic FSM M. There exists a complete checking sequence for M capable of detecting any fault in an implementation $Impl$, which has the same input I and output O alphabet as M and has no more states than M. The specification is modified to M' by a unit change, i.e., by applying a single – output or a next state – change operator. The problem is to create a complete checking sequence for the new specification M' if such exists.

We concentrate on systems with reliable reset capability, and we assume the HIS-Method as a reference point in creating an incremental algorithm and evaluating its performance. The HIS-method is essentially the superposition of two completely independent algorithms. One is used to build a set of input

sequences responsible for reaching all states of the finite state machine (a prefix-closed state cover set). The other is applied to create a set of input sequences to verify the next state of the transition (a separating family of sequences).

Our incremental test generation method likewise involves two completely autonomous incremental algorithms. Note that these algorithms may also be applied independently for various purposes. They could be used to detect undesirable effects of a planned modification during development, such as subsets of states becoming equivalent or unreachable.

It has to be emphasized that a given change to the specification FSM may affect the two algorithms differently. Therefore two separate Δ parameters (see Section 4.1) have to be used to capture the extent in which the changes affect the two algorithms.

5.1 Incremental Algorithm for Maintaining a Prefix-closed State Cover Set

Given a specification FSM M, a prefix-closed state cover set Q of M and a change $\omega(s_m, i, o, s_j)$ to FSM M turning it to M' our purpose is to create a new valid prefix-closed state cover set Q' for M'.

The problem can be reduced to maintaining a spanning tree of the state transition graph of the specification machine rooted from the initial state s_0 (see Section 3.1). Assuming the spanning tree ST of FSM M representing the Q set – i.e., input sequences on all partial paths of ST are in Q – we intend to produce a new valid spanning tree ST' of FSM M'.

Let us call a transition an ST-transition iff it is in ST. A subtree of ST rooted from a state $s_i \neq s_0$ is a proper subtree of the spanning tree ST and will be referred to as ST_{s_i}.

Given the change $\omega(s_m, i, o, s_j)$ we will refer to state s'_m of FSM M' as *modified* state, since a transition originating from state s_m of FSM M is modified by the change. In this paper we focus on unit changes; at each incremental step there is a single *modified* state, i.e., the cardinality of the set of *modified* states $MODIFIED$ abbreviated as MOD is one: $|MOD| = 1$.

A state s'_i of FSM M' is affected by the change with respect to the Q set iff for input sequence $q_i \in Q$ corresponding to state s_i: $\delta(s_0, q_i) \neq \delta'(s'_0, q_i)$. Such a state is said to be a *q-affected* state.[7] The algorithm identifies the set of *q-affected* states $AFFECTED_Q$ abbreviated as AFF_Q, where $0 \leq |AFF_Q| \leq n$. If AFF_Q is not an empty set – $|AFF_Q| > 0$ – then ST must be adapted to M'.

We define the set $CHANGED_Q \subseteq S'$ to be $MOD \cup AFF_Q$ and denote $|CHANGED_Q|$ as Δ_Q. The set $CHANGED_Q$ will be used as a measure of the

[7] Other definitions of *q-affected* state could be used depending on the assumed testing algorithm. A more relaxed definition could be for example the following: A state s'_i of FSM M' is affected by the change with respect to the Q set iff there exists no path from s'_0 to s'_i in ST' after the change. This definition should be assumed in case the same set (for example a distinguishing sequence or W-set) is used to check each ending state.

size of the change to the specification, and the complexity of the incremental algorithm will be expressed as a function of parameter Δ_Q, where $1 \leq \Delta_Q \leq n$.

The input of the algorithm is the original machine M, the change operator and the spanning tree ST of M. It provides M', the new spanning tree ST' of M' and the set of unreachable states as output. The algorithm consists of two phases and handles output and next state changes separately in the first phase. The first phase marks all q-affected states of FSM M' then collects them in the set AFF_Q. If $|AFF_Q| = 0$ then ST is a valid spanning tree of M' and the algorithm terminates, otherwise the second phase completes the spanning tree for all *q-affected* states.

Phase 1 – Output Change Take output change $\omega_o(s_m, i, o_x, s_j) = (s'_m, i, o_y, s'_j), o_x \neq o_y$. Create FSM M' by applying the change operator. Initialize AFF_Q as an empty set, and the spanning tree ST' of M' as $ST' := ST$.

As an output change operator is applied to FSM M, it only changes an edge label of the state transition graph of FSM M, but does not affect 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. AFF_Q is not extended.

Phase 1 – Next State Change Take next state change $\omega_n(s_m, i, o_x, s_j) = (s'_m, i, o_x, s'_k), s_j \neq s'_k$. Create FSM M' by applying the change operator. Initialize AFF_Q as an empty set, and the spanning tree ST' of M' as $ST' := ST$.

- $(s_m, i, o, s_j) \notin ST$: If the transition upon input i at state s_m is not an ST-transition then any change to it can not affect the spanning tree of FSM M. AFF_Q is not extended.
- $(s_m, i, o, s_j) \in ST$: If the transition upon input i at state s_m is an ST-transition then the change affects the spanning tree. The *q-affected* states are identified walking the $ST'_{s'_j}$ subtree. All states of $ST'_{s'_j}$ (including s'_j) are marked as *q-affected* states. The AFF_Q set can be determined using a simple breadth-first search of the $ST'_{s'_j}$ subtree with a worst case complexity of $|AFF_Q|$.

Phase 2: Determining a spanning tree of M' Phase 2 of the algorithm takes the set AFF_Q from Phase 1 and completes the spanning tree for each member of AFF_Q to create a spanning tree ST' of M'.

If $|AFF_Q| = 0$ (there are no *q-affected* states) then ST is a spanning tree of M'. Return ST' and the algorithm terminates.

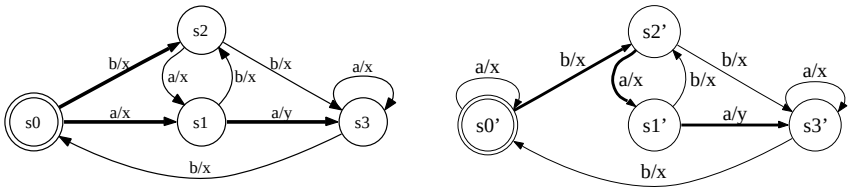
If $|AFF_Q| > 0$ then we apply the following method: All transitions of ST' leading to *q-affected* states are removed from ST' along with the modified transition (s'_m, i, o, s'_k). Then we extend ST' as follows.

For all s'_x in AFF_Q we start checking the transitions leading to s'_x in M' until either a transition originating from an unaffected state is found or there are no more inbound transitions left. If a transition (s'_i, i, o, s'_x) such that $s'_i \notin AFF'_Q$ is found then: (I) $ST' := ST' \cup (s'_i, i, o, s'_x)$, (II) $AFF_Q := AFF_Q \setminus \{s'_x\}$, (III) if there is transition (s'_x, i, o, s'_y) where $s'_y \in AFF_Q$ then repeat Steps I-III on s'_y.

The algorithm stops after all s'_x in AFF_Q has been checked, then return ST' and AFF_Q; the algorithm terminates. At the end of the last turn ST' will be a

spanning tree of M', and any s'_z remaining in AFF_Q is unreachable from s'_0 in M'.

Q-set example. Take FSM M on Figure 1(a) where bold edges represent the spanning tree and the double circle denotes the initial state.



(a) FSM M with its spanning tree ST

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

Fig. 1. Example for maintaining the preamble

Initially let $ST' = ST$ and $AFF_Q = \emptyset$. The modification $\omega_n(s_0, a, x, s_1) = (s'_0, a, x, s'_0)$ is a next state change. As transition (s_0, a, x, s_1) is in ST, we need to determine the set of *q-affected* states by walking the $ST'_{s'_1}$ subtree. We get $AFF_Q = \{s'_1, s'_3\}$. In Phase 2 transitions leading to *q-affected* states – (s'_0, a, x, s'_1) and (s'_1, a, y, s'_3) – are removed from ST'. Then one of the states – say s'_1 – is selected from AFF_Q. Transition (s'_2, a, x, s'_1) is identified, which is a link originating from a not affected state s'_2. We add it to ST' and remove s'_1 from AFF_Q. We then check transitions originating from s'_1 and find (s'_1, a, y, s'_3) that leads to a *q-affected* state. We add (s'_1, a, y, s'_3) to ST' and remove s'_3 from AFF_Q. Now, $AFF_Q = \emptyset$, so the algorithm terminates and returns ST', see Figure 1(b).

Theorem 1. *The incremental algorithm for maintaining a spanning tree described above has a time complexity of $O(p * \Delta_Q)$, where $1 \leq \Delta_Q \leq n$.*

Proof. Phase 1 of the algorithm has worst case complexity of $O(|AFF_Q|)$.

Phase 2 of the algorithm first searches a path from the unaffected states of M' to the *q-affected* states. There are exactly $p * |AFF_Q|$ transitions originating from the *q-affected* states. Therefore there can be at most $p * |AFF_Q|$ steps that do not provide a path from unaffected states of M' to the *q-affected* states summarized over all backward check turns of Phase 2. Thus there are no more than $(p + 1) * |AFF_Q|$ backward check turns.

If a link is found from an unaffected state to an affected state s'_x then the algorithm adds all states of AFF_Q reachable from s'_x via affected states. Again, there can be at most $p * |AFF_Q|$ such steps summarized over all forward check turns of Phase 2.

As any of the $p * |AFF_Q|$ transitions are processed at most twice by the algorithm, less than $2 * (p + 1) * |AFF_Q| \approx O(p * |AFF_Q|)$ steps are necessary

to complete Phase 2. The total complexity of the algorithm is $O(p * |AFF_Q|) \leq O(p * \Delta_Q)$ □

The new set Q' of M' contains $|AFF_Q|$ modified sequences: Input sequences of ST' leading from s'_0 to s'_i for all s'_i in AFF_Q.

5.2 Incremental Algorithm for Maintaining a Separating Family of Sequences

We are again given the specification FSM M, and the change $\omega(s_m, i, o, s_j)$ turning M to M'. We also have a separating family of sequences of FSM M (a separating set for each state): $Z = \{Z_1, ..., Z_n\} | Z_i = \{z_{ij}\}, j = 1...n$, where z_{ij} is a separating sequence of states s_i, s_j of FSM M. Our objective is to create a new separating family of sequences Z' for M'. Note that we consider a somewhat structured separating family of sequences as discussed later. This, however, does not restrict the generality of the approach as each incremental step generates a separating family according the assumed structure.

Informally speaking, to maintain a separating family of sequences we have to identify all separating sequences affected by the change. Then for all such state pairs a new separating sequence has to be generated. Notice that this is a problem over state pairs rather than states. Therefore we introduce an auxiliary directed graph A^M with $n(n+1)/2$ nodes, one for each unordered pair (s_j, s_k) of states of M including identical state pairs (s_j, s_j). There is a directed edge from (s_j, s_k) to (s_l, s_m) labeled with input symbol i iff $\delta(s_j, i) = s_l$ and $\delta(s_k, i) = s_m$ in M. The auxiliary directed graph A^M is used to represent and maintain separating sequences of FSM M. The graph is updated by our algorithm at each incremental step.

We define a *separating state pair* as an unordered pair of states (s_x, s_y) such that $\lambda(s_x, i) \neq \lambda(s_y, i)$ for some $i \in I$. A machine M is minimal iff there is a path from each non-identical state pair $(s_j, s_k), j \neq k$ to a *separating state pair* in its auxiliary directed graph A^M. The input labels along the route concatenated by the input distinguishing the *separating state pair* form a separating sequence of states s_j and s_k.

We make the following assumptions on the separating sequences of FSM M: (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 pair has multiple such inputs, then the input to be associated is chosen randomly. (II)The set of separating sequences of FSM M is prefix-closed.

Then separating sequences of FSM M form an acyclic subgraph of the auxiliary directed graph A^M, such that there is exactly one path from each state pair $(s_x, s_y), x \neq y$ to a *separating state pair*. That is, separating sequences form a forest 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 2(a) below for example. Let us refer to this forest (a subgraph of A^M) as SF. We call an edge of A^M an SF-edge iff it is in SF. A subtree of SF having state pair (s_i, s_j) as root is a proper subtree of the forest SF and

will be referred to as SF_{s_i,s_j}. Note that by walking such a tree (or its subtree) we always assume that it is explored opposing edge directions from the root (or an inner node) toward leaves.

Thus the problem of deriving the separating family of sequences for 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.

Given the change $\omega(s_m, i, o, s_j)$ turning M to M' all state pairs that include state s_m are modified to construct the auxiliary directed graph $A^{M'}$ of FSM M'. Accordingly all unordered state pairs of $A^{M'}$ involving s'_m are referred to as *z-modified* state pairs. As a result of the unit change assumption the cardinality of the set of *z-modified* state pairs $MODIFIED_Z$ abbreviated as MOD_Z is n: $|MOD_Z| = n$.[8]

The algorithm derives the set of state pairs affected by the change. Such state pairs are said to be *z-affected*. The set of *z-affected* state pairs is referred to as $AFFECTED_Z$ abbreviated as AFF_Z, where $0 \leq |AFF_Z| \leq n(n-1)/2$. We define the set $CHANGED_Z \subseteq S' \times S'$ as $MOD_Z \cup AFF_Z$. The complexity of the incremental algorithm will be expressed as a function of parameter $|CHANGED_Z|$ referred to as Δ_Z, where $n \leq \Delta_Z \leq n(n-1)/2$.

The input of the algorithm is the auxiliary directed graph A^M of FSM M, the change operator and the forest SF of A^M representing separating sequences of M. The output is $A^{M'}$, the new forest SF' of $A^{M'}$ and a set containing pairs of equivalent states.

The algorithm consists of two phases and handles output and next state changes separately in the first phase.

Phase 1 – Output Change Take output change $\omega_o(s_m, i, o_x, s_k) = (s'_m, i, o_y, s'_k), o_x \neq o_y$. Initialize AFF_Z as an empty set, $A^{M'} := A^M$ and $SF' := SF$.

For state pairs $\forall s'_i \in S' : (s'_m, s'_i)$ apply the change to $A^{M'}$ and:

- If state pair (s'_m, s'_i) is a new *separating state pair* then mark it and associate i as separating input.
- If i has been the separating input of *separating state pair* (s_m, s_i) in A^M but $\lambda'(s'_m, i) = \lambda'(s'_i, i) = o_y$ then all state pairs of the tree with (s'_m, s'_i) root – including (s'_m, s'_i) – are added to AFF_Z (marked as *z-affected*). These states can be identified by walking the given tree from the root.
  - If there is another input $i_1 | \lambda'(s'_m, i_1) \neq \lambda'(s'_i, i_1)$ then (s'_m, s'_i) remains a *separating state pair* with i_1 associated as separating input. State pair (s'_m, s'_i) is removed from AFF_Z.
  - If $\forall i \in I : \lambda'(s'_m, i) = \lambda'(s'_i, i)$ then (s'_m, s'_i) is no longer a *separating state pair*, thus the *separating state pair* marking is removed from (s'_m, s'_i).
- Do nothing otherwise.[9]

Phase 1 – Next State Change Take next state change $\omega_n(s_m, i, o_x, s_k) = (s'_m, i, o_x, s'_l), s_k \neq s'_l$. Initialize AFF_Z as an empty set, $A^{M'} := A^M$ and $SF' := SF$.

For state pairs $\forall s'_i \in S : (s'_m, s'_i)$ apply the change to $A^{M'}$ and:

[8] Pairs s'_i, s'_i of identical states are also modified here.

[9] One could assume different definitions for affected state pairs as a design choice.

– If the edge of A^M marked by input i at state pair (s_m, s_i) is an SF-edge then the modification affects the given tree of the spanning forest. All state pairs of the $SF'_{s'_m,s'_i}$ subtree are *z-affected* states (including (s'_m, s'_i)). Thus the $SF'_{s'_m,s'_i}$ subtree is explored using a simple breadth-first search, all state pairs are added to AFF_Z (marked as *z-affected*).
– Do nothing otherwise.

Phase 2 Phase 2 of the algorithm takes the set AFF_Z from Phase 1 and updates the forest SF' for each member of AFF_Z.

If $|AFF_Z| = 0$ then SF is a valid forest over $A^{M'}$ representing a separating sequence for each non-identical state pair of M'. Return SF' and the algorithm terminates.

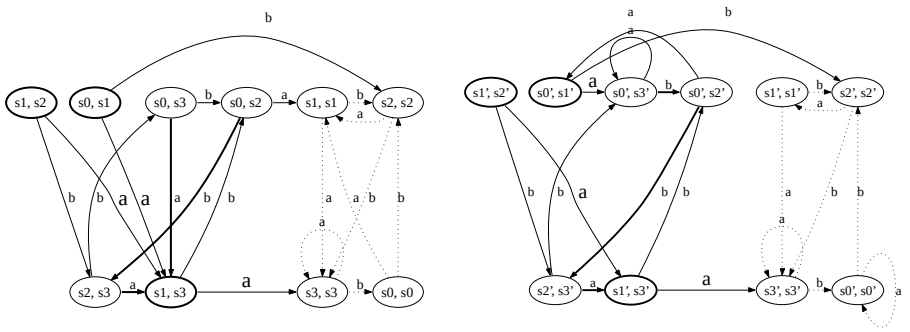
If $|AFF_Z| > 0$ then the following method is applied: All edges of SF' originating from *z-affected* state pairs are removed from SF'. Then we extend SF' as follows. We examine all edges of $A^{M'}$ originating from a *z-affected* state pair and construct a subgraph of $A^{M'}$ denoted as $A^{M'}_{AFF}$ the following way: (I) For each *z-affected* state pair there is a corresponding node in $A^{M'}_{AFF}$. (II) For each edge between *z-affected* state pairs there is an edge in $A^{M'}_{AFF}$. (III) If there is an edge originating from a *z-affected* state pair leading to a not affected state pair then we mark the given *z-affected* state pair at the head of the edge.

Next we explore $A^{M'}_{AFF}$ opposing edge directions from marked state pairs using breadth-first search to create a spanning forest over $A^{M'}_{AFF}$ with marked state as root nodes. All state pairs covered by the spanning forest are removed from AFF_Z. Finally SF' is expanded simply appending the spanning forest of $A^{M'}_{AFF}$. Each tree of the forest of $A^{M'}_{AFF}$ is linked to SF' by an edge leading to a not affected state pair from its marked root node. Return SF' and AFF_Z; the algorithm terminates.

At the end of the algorithm AFF_Z contains any pairs of equivalent states for which no separating sequence exists. Each partial path of SF' represents a separating sequence of M': Given a path from node (s'_i, s'_j) to *separating state pair* (s'_x, s'_y) the input labels along the route concatenated by the separating input of s'_x, s'_y form a separating sequence z'_{ij} of states s'_i and s'_j. The separating family of sequences of FSM M' is given as $Z' = \{Z'_1, ..., Z'_n\} | Z'_i = \{z'_{ij}\}, j = 1...n$.

Z set example. The auxiliary graph A^M of M is presented on Figure 2(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.

Initially $AFF_Z = \emptyset$ 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 $A^{M'}$. (s'_0, s'_1) 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*: $AFF_Z = \{(s'_0, s'_3)\}$. In Phase 2 the a-labeled edge originating from (s'_0, s'_3) is removed from SF'. Then edges originating from (s'_0, s'_3) are checked



(a) The auxiliary graph A^M of FSM M (b) The updated auxiliary graph $A^{M'}$ of FSM M'

Fig. 2. Auxiliary graphs

and an edge $\langle (s'_0, s'_3), (s'_0, s'_2) \rangle$ leading to a non-affected state pair is found. The given edge is added to SF' and (s'_0, s'_3) is removed from AFF_Z. Now, $AFF_Z = \emptyset$, thus the algorithm terminates and returns SF', see Figure 2(b). All separating sequences are unchanged except the one of states s'_0 s'_3, which is changed from $a \cdot a$ to $b \cdot b \cdot a \cdot a$.

Theorem 2. *The incremental algorithm for maintaining a separating family of sequences described above has a time complexity of* $O(p * \Delta_Z)$*, where* $n \leq \Delta_Z \leq n^2$.

Proof. Regardless of change operator type Phase 1 of the algorithm involves $|MOD_Z|$ modification steps and $O(|AFF_Z|)$ steps to identify affected state pairs. Phase 2 of the algorithm first creates a subgraph in $p * |AFF_Z|$ steps and then creates a spanning forest over it with $O(p * |AFF_Z|)$ complexity. Thus the total complexity of the algorithm is $O(p * \Delta_Z)$. □

5.3 Total Complexity of the Incremental Testing

Our algorithm – just as the original HIS-method – derives actual test sequences in two stages by concatenating sequences from the sets Q and Z_i. Each test sequence – a part of the checking sequence – is either a sequence $r \cdot q_i \cdot z_{ij}$ (stage 1) or a sequence $r \cdot q_i \cdot i \cdot z_{xy}$ (stage 2). A test sequence must be regenerated after a change if either the q-sequence, or the z-sequence of the given test sequence is modified by our incremental algorithms above. That is, a test sequence $r \cdot q_i \cdot \ \ldots \ \cdot z_{ij}$ of M is modified iff $s'_i \in AFF_Q$ or $(s'_i, s'_j) \in AFF_Z$. Such sequences are identified using links between the sets Q', $Z'_i, i = 1...n$ and test sequences. The number of modified test sequences is less or equal than $p * n^2$, i.e., in worst case the number of test cases to be generated is equivalent to those generated by a batch algorithm. The resulting test set is a complete test set of M' that is no different

than one generated using the *batch* HIS-method. It consists of the same set of test sequences generated using valid Q' and Z' sets of M'.

Note that the concatenation operation is – from the complexity point of view – a quite expensive step. Concatenation, however, only has to be performed as a part of the testing procedure itself. If no actual testing is needed after a change to the specification then we should only – very efficiently – maintain the sets Q and Z according to each modification and do the concatenation just as necessary.

6 Related Work

Nearly all test generation approaches in the FSM test generation literature propose batch algorithms, i.e., they focus on building test sets for a system from scratch without utilizing any information from tests created for previous versions of the given system. One of the few exceptions – the most relevant research – has been the work of El-Fakih at al. [18]. Similarly to our approach, the authors assume a specification in form of a complete deterministic FSM M, which is modified to a new specification M'. The problem in both cases is to generate a checking sequence to test if an implementation $Impl$ conforms to M'. Our approach, however, involves some fundamental improvements on El-Fakih's method. The most important are:

1. El-Fakih's algorithm does not intend to create a complete test set for the modified specification M', instead it may be used to "generate tests that would only test the parts of the new implementation that correspond to the modified parts of the specification" [18]. This necessitates the following quite restrictive assumption: "the parts of the system implementation that correspond to the unmodified parts of the specification have not been changed" [18]. Thus, it is presumed that no accidental or intentional (malicious) changes are introduced to supposedly unmodified parts of the implementation. Such faults could remain undetected as the test set generated by the algorithm is not complete. Note that the assumption above is unavoidable as not even the union of the existing test set of M and the incremental test set generated by the algorithm provide a complete test set for M'.
   Our algorithm, on the other hand, maintains a complete test set across the changes to the specification. The algorithm modifies the existing test set of M to create a complete test set for M' – if such exists – capable of detecting any fault in $Impl$.
2. El-Fakih's algorithm is not a bounded incremental algorithm in the sense that it uses traditional batch algorithms to create a state cover set and a separating family of sequences for a given FSM upon each modification. Therefore its complexity is the function of the size of the input FSM, not the extent of the change.
   Our method in turn is a bounded incremental algorithm, its complexity is dependent on the extent of the change. It identifies how the modification affects the existing test set of the original specification machine M. New tests

are only generated for the affected part and the calculation is independent of the unaffected part. The complexity of our algorithm is no worse than the complexity of the corresponding batch algorithm.

7 Conclusion

We have presented a bounded incremental algorithm to generate test cases for deterministic finite state machine models. Our approach assumes a changing specification and utilizes an existing test set of the previous version to efficiently maintain a complete test set across the changes to the specification. For each update of the system a complete test set is generated with the same fault detection capability as that of a traditional *batch* algorithm. The complexity of the algorithm is evaluated based on the bounded incremental model of computation of Ramalingam and Reps [1]. The time complexity of the proposed algorithm is shown to be bounded; it is a function of the size of the change to the specification rather than the size of the specification itself. It is never worse than the complexity of the relevant traditional algorithm – the HIS-method. Furthermore, the two autonomous incremental algorithms building up the incremental test generation method may also be applied independently for various purposes during development.

In the future we plan to further experiment with the presented algorithm to gain sufficient performance data for practical analysis. Our current focus has been on time complexity but the approach leaves space for fine-tuning and optimizations in several aspects that will have to be studied. The research reported here is regarded as a first step in developing efficient incremental testing algorithms. We plan to investigate if this approach can be extended to different problems and models.

References

1. Ramalingam, G., Reps, T.: On the computational complexity of dynamic graph problems. Theoretical Computer Science **158**(1-2) (1996) 233–277
2. Sabnani, K., Dahbura, A.: A protocol test generation procedure. Computer Networks and ISDN Systems **15**(4) (1988) 285–297
3. Yannakakis, M., Lee, D.: Testing finite state machines: fault detection. In: Selected papers of the 23rd annual ACM symposium on Theory of computing. (1995) 209–227
4. Petrenko, A., Yevtushenko, N., Lebedev, A., Das, A.: Nondeterministic state machines in protocol conformance testing. In: Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI. (1994) 363–378
5. Subramaniam, M., Pap, Z.: Analyzing the impact of protocol changes on tests. In: Proceedings of the IFIP International Conference on Testing Communicating Systems, TestCom. (2006) 197–212
6. Friedman, A.D., Menon, P.R.: Fault Detection in Digital Circuits. Prentice-Hall (1971)

7. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
8. Holzmann, G.J.: Design and Validation of Protocols. Prentice-Hall (1990)
9. ITU-T: Recommendation Z.100: Specification and description language (2000)
10. TC97/SC21, I.: Estelle – a formal description technique based on an extended state transition model. international standard 9074 (1988)
11. Subramaniam, M., Chundi, P.: An approach to preserve protocol consistency and executability across updates. In: Proceedings of Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM. (2004) 341–356
12. Pap, Z., Csopaki, G., Dibuz, S.: On the theory of patching. In: Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods, SEFM. (2005) 263–271
13. Lee, D., Yiannakakis, M.: Principles and methods of testing finite state machines – a survey. Proceedings of the IEEE **84**(8) (1996) 1090–1123
14. Bochmann, G.V., 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, New York, NY, USA, ACM Press (1994) 109–124
15. Chow, T.: Testing software design modelled by finite-state machines. IEEE Transactions on Software Engineering **4**(3) (1978) 178–187
16. Fujiwara, S., v. Bochmann, G., Khendec, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state model. IEEE Transactions on Software Engenieering **17** (1991) 591–603
17. Kohavi, Z.: Switching and Finite Automata Theory. McGraw-Hill, New York (1978)
18. El-Fakih, K., Yevtushenko, N., von Bochmann, G.: FSM-based incremental conformance testing methods. IEEE Transactions on Software Engineering **30**(7) (2004) 425–436