BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
DEPARTMENT OF TELECOMMUNICATIONS AND MEDIA INFORMATICS
DOCTORAL SCHOOL OF INFORMATICS

INCREMENTAL TEST GENERATION ALGORITHMS

Gábor Árpád Németh

MSc. in Electrical Engineering

Summary of the Ph.D. Dissertation

Supervised by:

Dr. Zoltán Pap

Dr. Gyula Csopaki

Budapest, Hungary

2015

1 Introduction

Testing plays a very important role in software and hardware development. The complexity of software and hardware products is continuously increasing, whereas the time frame available for the development of a new product is becoming shorter. The fast-paced development increases the probability of faults. While quality assurance is essential, the resources available for testing may be limited compared to the large amount of work to be done. The testing process of an end product often needs to be carried out without knowing the exact internal architecture of the product we want to test.

A promising solution for the problems above is to formulate the requirements of the product in a formal language and use this description as a starting point for creating test cases. Formal descriptions can help understand how the system works in the early design phase and in turn augment or redesign it as required. The Finite State Machines (FSMs) are the most common formal description for telecommunication software and protocols.

Although test generation methods based on FSMs have been studied extensively in the last decades [15, 19, 6, 8], very little effort has been invested in dynamic scenarios involving changing system specifications. Almost all FSM-based test generation algorithms suppose rigid, unchanging specifications. This is especially surprising considering that most recent system development methodologies propose *incremental* approaches involving step-by-step refinement of the system under development [26, 16, 9]. Such a methodology is for example the widespread and popular Agile in software development [21].

From the testing perspective, iterative approaches open new possibilities for improving test generation methods. By identifying the effect of changes in each iteration step during development, one may want to reuse as much of the test set of the previous version of the system as possible, and focus test generation only on specific parts of the system. This may result in significantly faster test generation in iterative development scenarios. This approach supports both testing of the entire system and testing specific parts of the system, which are affected by the series of changes applied to the specification. The partition of the tested system into affected and unaffected parts also matches recent, non model-based software testing practices (e.g. functional tests focus on the new functionality of the software and the role of regression tests is to assure backward compatibility). As a result, iterative approaches in FSM model-based testing can be easily adapted into current software testing methodologies.

2 Research Objectives

The purpose of the dissertation is to propose novel incremental algorithms that generate test cases much more efficiently for evolving system specifications than previous methods. The focus is on FSM modelling technique; the most common formal description for telecommunication software and protocols.

My approach assumes changing specifications and utilizes an additional information – an existing test set of the previous version of specification – to assure a complete test set of the specification after the development steps have been applied. The research focuses on the extent of changes to the previous test set and the creation of appropriate update functions for different type of development steps to provide a new test set for the updated specification efficiently.

I divided my results into two thesis groups.

The first thesis group focuses on the maintenance of a *checking sequence*, which guarantees to find output and transfer faults of a given FSM. It deals with a structured test suite, where the different parts of the test suite can be used for different testing purposes (i.e. for checking assumptions about specification FSMs). In this thesis group I propose two algorithms which keep the test sequences of the HIS-method [28, 24] update across changes.

The second thesis group focuses on the maintenance of a test sequence, which requires much less assumptions about specification and implementation machines and development steps as the previous test suite has. In this thesis group I introduce two cooperating algorithms which together keep the *Transition Tour* [22] test sequence across changes up to date. This test sequence traverses every transition of the FSM at least once and – although it provides no guarantee to find transfer faults – it is much shorter than the checking sequence of the HIS-method.

3 Methodology

In my thesis I have used the results of graph theory, FSM theory and complexity theory.

The proposed incremental algorithms of Theses 1 and 2 are based on the traditional algorithms of FSM theory. In Thesis 2 different tools and methods related to graph theory is used.

I have always provided complexity calculations for the presented algorithms. In Thesis 1 the complexity calculation is based on the approach of Ramalingam and Reps [25].

I have implemented the algorithms in Java and in C++ using LEMON [2] library. I demonstrated the algorithms through examples and extensive simulations were also performed to investigate their effectiveness and compare to the most relevant, traditional methods of FSM theory.

4 New Results

4.1 Bounded Incremental Algorithms for HIS-method Test Case Maintenance Across Updates

Given a completely specified, deterministic, reduced FSM M with n states and p input symbols, 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 a checking sequence for FSMs with reliable reset capability [19, 5]. Such solutions are the W-method [7], the Wp-method [14] and the HIS-method [28, 24]. All of these algorithms have a similar structure containing two main stages. The first – state identification – stage checks for each state of the specification whether it exists in the implementation as well. The second – transition testing – stage 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.

The HIS-method contains two fundamental data structures:

- A *state cover set* $Q = \{q_1, \ldots, q_n\}$ responsible for reaching all states $s_1 ... s_n$ of the FSM.
- A *separating family of sequences* (which is sometimes also referred to as the *family of Harmonized State Identifiers (HSI)*) $Z = \{Z_1, \ldots, Z_n\}$, that verifies the next state of each transition (where $Z_i = \{z_{ij}\}, j = 1 \ldots n$ are the separating sets for state s_i used to distinguish s_i for all states $s_j, j \neq i$).

Test cases for stages 1 and 2 are derived by the concatenation of the appropriate members of the data structures described above.

Nonetheless, the methods discussed above deal only with unchanging specifications and are therefore unable to use any test set generated for a previous system version. The incremental approach in the field of testing has been first introduced by El-Fakih et al. [12]. Their method generates test cases only for the modified parts of the system, but it is not capable of maintaining a complete test set across changes. Moreover, the method can not be considered as a true bounded incremental algorithm, as its complexity depends on the size of the input FSM rather than on the extent of changes.

The main purpose of the research was to find a solution to efficiently generate a *checking sequence* for changing specifications. As far as I am concerned no true bounded incremental test generation algorithm that is capable to maintain the complete test set across changes has been proposed before our work in [C1].

Thesis 1. *[C1][C6] I have proposed two incremental algorithms to keep a checking sequence updated across changes if the system specification is given as a deterministic, completely specified, strongly connected FSM with reliable reset capability. The algorithms utilize an existing test set of the previous version of the specification to efficiently generate the result for the updated specification.*

Similarly to the HIS-method, the proposed solution consists of two parts: the first one referred to as State Cover Set Maintenance (SCSM, presented in Section 4.1.1) maintains a prefix-closed state cover set responsible for reaching all states of the machine, while the second one, called State Identification Maintenance (SIM, presented in Section 4.1.2) maintains a separating family of sequences used to verify the next state of transitions.

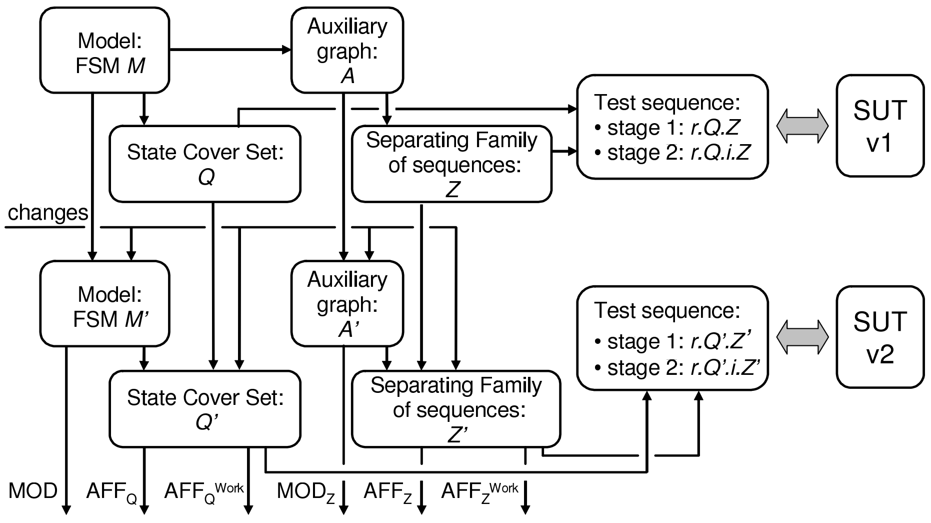
Model: FSM M
Auxiliary graph: A
Test sequence: • stage 1: r.Q.Z • stage 2: r.Q.i.Z
SUT v1
State Cover Set: Q
Separating Family of sequences: Z
changes
Model: FSM M'
Auxiliary graph: A'
Test sequence: • stage 1: r.Q'.Z' • stage 2: r.Q'.i.Z'
SUT v2
State Cover Set: Q'
Separating Family of sequences: Z'
MOD
AFF_Q
AFF_Q^Work
MOD_Z
AFF_Z
AFF_Z^Work

Figure 1: Block diagram of the incremental maintenance of HIS-method test cases

The high-level view of the approach is presented in Figure 1. The upper part of the figure represents the traditional generation of the HIS test sequence: state cover set Q and separating family of sequences Z are generated from scratch, then a checking sequence is created by the concatenation of the subsequences of these sets. This checking sequence is then applied to the system under test (SUT). The auxiliary graph A denotes a graph over state pairs of FSM M, which helps generate separating family of sequences Z. The lower part of the figure shows how the sequence of changes affects the generation of the test cases of the HIS-method. The sequence of edit operators turns FSM M into FSM M'. Instead of repeating the previous operations on M', the SCSM algorithm (presented in Section 4.1.1) utilizes a prefix-closed state cover set Q of the previous system version to create a valid state cover set Q' of M'. The SIM algorithm (introduced in Section 4.1.2) maintains a separating family of sequences Z for this development cycle in order to produce a valid separating family of sequences Z' of M'. The SIM algorithm also utilizes and incrementally maintains the auxiliary graph A of the specification in order to generate Z' efficiently. Then the subsequences of sets Q' and Z' are concatenated into a checking sequence and can be applied to the modified system (SUT v2).

The two algorithms show the elements whose test cases are affected by the given sequence of changes, denoted by AFF_Q and AFF_Z. Thus, the entire 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' can also be observed, denoted by MOD and MOD_Z, respectively.

As algorithms maintain a complete checking sequence, the specification FSM and an auxiliary graph of this specification FSM through a series of changes, the results can be used as an input for the next iteration, i.e. the algorithms support the step-by-step development process of the specification.

Based on [23] the following changes are considered to modify the FSM:

- changing the next state of a transition,
- changing the output label of a transition.

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

Thesis 1.1. *[C1][C6] I have created an algorithm to maintain a State Cover Set from a given s_0 initial state. The proposed algorithm updates the given State Cover Set only over states that are affected by the given sequence of changes. The algorithm is able to detect if the original assumption on strongly connected capability does not hold after the sequence of changes has been applied.*

A specification FSM M, a prefix-closed state cover set Q of M and a sequence of changes Ω that turns the FSM M into M' are given. The purpose is to create a new valid prefix-closed state cover set Q' for M'.

As the set Q of M can be represented by a spanning tree ST rooted at the initial state s_0, the problem can be reduced to the maintenance of this spanning tree across changes. Thus, the objective is to create a new valid spanning tree ST', which corresponds to the Q' set of FSM M'.

The following notations are used in the description of the SCSM (State Cover Set Maintenance) algorithm. A transition is called an ST-transition iff it is in ST. A subtree of ST rooted at state s_i is denoted by ST_{s_i}. The MOD set contains 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 set Q. Note that if a state is q-affected, a new input sequence has to be generated for it. Let AFF_Q^{Work} denote the work set that is used to keep track of affected states, for which the new input sequences have not yet been generated. 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 algorithm is expressed as a function of Δ_Q in Thesis 1.2.

The algorithm (see Algorithm 1) consists of two phases. The first one determines the set of q-affected states of FSM M' and deletes spanning edges that lead to these states. Then, the second phase extends the spanning tree for q-affacted states to create a valid set Q' of the modified FSM M'.

When all elements of AFF_Q^{Work} have been checked, the SCSM algorithm terminates:

- Spanning tree ST' represents the updated prefix-closed state cover set Q' of the modified FSM M'.

- The elements of set MOD denote the states where the machine has been modified.

- Set AFF_Q represents states of M', for which the sequences of state cover set have been changed.

- Set AFF_Q^{Work} represents states of M', which have become unreachable after the series of changes Ω have been applied to M.

Algorithm 1: Maintaining a Prefix-closed State Cover Set

```
   input  : $M = (I, O, S, T)$; $\Omega = \omega_1, .., \omega_K$; $ST = (S_{ST}, T_{ST}), T_{ST} \subset T$; $S_{ST} \subseteq S$;
   output: $ST'$; $MOD$; $AFF_Q$; $AFF_Q^{Work}$; $M'$;
0  data($k = 1..K$; $M' = (I, O, S', T')$; $ST' = (S'_{ST}, T'_{ST})$; $MOD \subseteq S$; $AFF_Q \subseteq S$;
   $AFF_Q^{Work} \subseteq S$; $ST_{temp} = (S_{temp}, T_{STtemp})$);
   /* Function $TreeWalk$ walks on a subtree of the given $ST'$ spanning
      tree with a given root $s'_b$ using simple breadth-first search.
      It results in a structure, that contains states and
      transitions that can be reached from a given state $s'_b$, and that
      will be referred to as $.S$ and $.T$, respectively.  Function
      $SpanningTree$ creates a spanning tree for a given subset of
      states in set $AFF_Q^{Work}$ which are reachable from a given root
      state $s'_j$ using breadth-first search in $M'$.  */
1  $M' := M$; $S'_{ST} := S_{ST}$; $T'_{ST} := T_{ST}$; $MOD := \emptyset$; $AFF_Q := \emptyset$; $AFF_Q^{Work} := \emptyset$;
   /* Phase 1:  Apply changes to $M$ and identify affected states */
2  foreach $k$ where
   $\omega_k(s_a, i, o_x, s_b) = (s'_a, i, o_y, s'_c) \in \Omega, s_a, s_b \in S, s'_a, s'_c \in S', i \in I, o_x, o_y \in O$  do
3      $T' := T' \setminus (s'_a, i, o_x, s'_b) \cup (s'_a, i, o_y, s'_c)$; $MOD := MOD \cup \{s'_a\}$;
4      if $(s_b \neq s'_c) \wedge ((s'_a, i, o_x, s_b) \in T'_{ST})$  then
5          $AFF_Q := AFF_Q \cup$ TreeWalk($ST', s'_b).S$;
6          $AFF_Q^{Work} := AFF_Q^{Work} \cup$ TreeWalk($ST', s'_b).S$;
7          $S'_{ST} := S'_{ST} \setminus$ TreeWalk($ST', s'_b).S$; $T'_{ST} := T'_{ST} \setminus$ TreeWalk($ST', s'_b).T$;
   /* Phase 2:  Create new test cases for affected states */
8  foreach $s'_j \in AFF_Q^{Work}$ do
9      foreach $t = (s'_i, i, o, s'_j) \in T$ do
10         if $s'_i \notin AFF_Q^{Work}$ then
11             $ST'_{temp}(S'_{STtemp}, T'_{STtemp})$ := SpanningTree($M', s'_j, AFF_Q^{Work}$);
               $T'_{ST} := T'_{ST} \cup \{(s'_i, s'_j)\} \cup T'_{STtemp}$; $S'_{ST} := S'_{ST} \cup \{s'_j\} \cup S'_{STtemp}$;
12             $AFF_Q^{Work} := AFF_Q^{Work} \setminus (\{s'_j\} \cup S'_{STtemp})$;
13 return $ST' \in (V', E')$, $MOD$, $AFF_Q$, $AFF_Q^{Work}$, $M'$;
```

Thesis 1.2. *[C1] I have proven that the time complexity of the incremental algorithm for maintaining a State Cover Set described above depends only on the extent of changes, and can be described as $O(p \cdot \Delta_Q)$, where $1 \leq \Delta_Q \leq n$.*

Proof. First of all, we note that for a unit change $|MOD| = 1$, and for multiple changes $1 \leq |MOD| \leq n$. The number of affected states – where the new test set has to be changed – is $0 \leq |AFF_Q| \leq n$.

In Phase 1, the SCSM algorithm updates the FSM M in order to get M', identifies the affected states and deletes spanning edges leading to these states. Updating the

FSM with changes Ω requires $O(|MOD|)$ steps. Identifying the affected states with breadth-first search, putting them into the AFF_Q and AFF_Q^{Work} sets and deleting spanning edges leading to these states has a total time complexity of $O(|AFF_Q|)$.

In Phase 2, the SCSM algorithm searches for 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. Thus there are at most $p \cdot |AFF_Q^{Work}| \leq p \cdot |AFF_Q|$ steps which do not find a path from a non-AFF_Q^{Work} state to an AFF_Q^{Work} state. Accordingly, $(p+1) \cdot |AFF_Q|$ check steps are required in the worst case. 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' is 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. Because any of the $p \cdot |AFF_Q|$ transitions are processed at most twice by the SCSM algorithm, no more than $2 \cdot p \cdot |AFF_Q| \approx O(p \cdot |AFF_Q|)$ steps are needed in Phase 2.

As $\Delta_Q = |MOD \cup AFF_Q|$, the total time complexity of the SCSM algorithm is $O(p \cdot \Delta_Q)$, where $1 \leq \Delta_Q \leq n$.

□

Note that the space complexity of the incremental incremental algorithm for maintaining a State Cover Set is $O(p \cdot n)$, which is the same as that of the corresponding part of the traditional HIS-method.

4.1.2 Incremental Algorithm for Maintaining a Separating Family of Sequences

Thesis 1.3. *[C1][C6] I have created an algorithm to maintain a separating family of sequences over changes. I represented the separating family of sequences as a spanning forest over an auxiliary directed graph, where each node of this auxiliary graph corresponds to a pair of states of the state transition graph. The proposed algorithm updates the spanning forest only over state pairs that are affected by the given sequence of changes. The algorithm is able to detect if the original assumption on the minimized machine does not hold after the sequence of changes has been applied.*

The SIM (State Identification Maintenance) algorithm has to identify all those pairs of states of M' that can no longer be distinguished with the corresponding separating sequence in set Z of machine M, and generate a new one for these state pairs. Let us define an auxiliary directed graph A with $n(n+1)/2$ nodes, where each node corresponds to an unordered pair $< s_k, s_l >$ of states of M including identical state pairs $< s_k, s_k >$. A 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 (where δ: $S \times I \to S$ denotes the next state function). The auxiliary directed graph A is used to represent and maintain separating sequences of FSM M. Graph A is updated by the SIM algorithm in each incremental step. Let the modified auxiliary graph be denoted by A'.

The following definitions and notations are used in the description of the SIM algorithm. A state pair $< s_x, s_y >$ is called a *separating state pair* iff it has a *separating input* i, which gives a different output when applied to the two states, i.e. $\lambda(s_x, i) \neq \lambda(s_y, i)$ for some $i \in I$ (where λ: $S \times I \to O$ denotes the output function). FSM M is reduced iff there exists a path in its auxiliary graph A 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 along the path 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 maintain set Z of FSM M efficiently, the following assumptions are made on the separating sequences of Z: (I) Each separating state pair $< s_x, s_y >$ has a single separating input $i \mid \lambda(s_x, i) \neq \lambda(s_y, i)$ assigned to it. If a given separating state pair has multiple such inputs, then one input symbol 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 the SIM algorithm as it generates a separating family of sequences Z' at each step, so that Z' fulfills the required assumptions.

If the assumptions 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 and with separating inputs, so that each tree of SF has a separating state pair as root and all edges of the given tree are directed towards the root. 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' across changes.

An edge of A is called an SF-edge iff it is in SF. A subtree of SF rooted at state pair $< s_i, s_j >$ is denoted by $SF_{<s_i,s_j>}$. The set MOD_Z collects the *modified* state pairs of the modified auxiliary graph A': $< s'_i, s'_j > \in MOD_Z$ iff a transition originating from state s_i or s_j of FSM M is modified by a change. Note that if a change modified a transition originating from state s_x, then all state pairs that involve s'_x are modified in A'. State pair $< s'_x, s'_y >$ is said to be *z-affected* and collected in set AFF_Z iff $< s'_x, s'_y >$ of the auxiliary graph A' is affected by a change with respect to the set Z, thus 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; this is denoted with AFF_Z^{Work}. An adaptive parameter is also defined 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 is expressed as a function of Δ_Z in Thesis 1.4.

As the previously discussed SCSM algorithm, the SIM algorithm (see Algorithm 2) also consists of two phases. The first phase determines z-affected state pairs and deletes their invalid test cases, while the second one expands the spanning forest SF' of A' to create a new, valid Z' set for the modified FSM M'.

Algorithm 2: Maintaining a Separating Family of Sequences

```
input  : $M = (I, O, S, T)$, $A = (V, E)$, $V = S \times S$; $E \subseteq V \times V$;
         $\Omega = \{\omega_1, .., \omega_K\}$; $SF = (V_{SF}, E_{SF}), V_{SF} \subset V, E_{SF} \subseteq E$;
output: : $SF' = (V'_{SF}, E'_{SF})$; $MOD_Z$; $AFF_Z$; $AFF_Z^{Work}$; $A'$;
0  data( $k = 1..K$; $A' = (V', E')$; $SF' = (V'_{SF}, E'_{SF})$; $MOD_Z \subseteq V$; $AFF_Z \subseteq V$;
   $AFF_Z^{Work} \subseteq V$; $A'_{AFF} = (V'_{AFF}, E'_{AFF})$; $ST'_{temp} = (V'_{temp}, E'_{temp})$)
1  $A' := A$; $V'_{SF} := V_{SF}$; $E'_{SF} := E_{SF}$; $MOD_Z := \emptyset$; $AFF_Z := \emptyset$; $AFF_Z^{Work} := \emptyset$;
2  foreach $k$ where
   $\omega_k(s_a, i, o_x, s_b) = (s'_a, i, o_y, s'_c) \in \Omega, s_a, s_b \in S, s'_c \in S', i \in I, o_x, o_y \in O$ do
3      foreach $e = (v_m, i, o_x, v_n), v_m = (s_m, s_n), v_n \in V, s_m = s'_a \vee s_n = s'_a$ do
4          $v_n := (s'_c, s'_l)$; $e := (v_m, i, o_y, v_n)$; $MOD_Z := MOD_Z \cup \{(v_m)\}$
       /* Phase 1: Identify affected state pairs */
       /* In case of an output change */
5      if $o_x \neq o_y$ then
6          foreach $s'_j \in S'$ do
7              if ($i$ has been sep. input of $< s_a, s_j >$)and($\lambda'(s'_a, i) = \lambda'(s'_j, i)$) then
8                  foreach $i_t \in I$ do
9                      if $\lambda'(s'_a, i_t) \neq \lambda'(s'_j, i_t)$ then
10                         $< s'_a, s'_j >$ remains a sep. state pair with $i_t$ sep. input
11                         $AFF_Z := AFF_Z \cup$ TreeWalk($SF', < s'_a, s'_j >).V$;
12                     else if $\forall i_t \in I, \lambda'(s'_a, i_t) = \lambda'(s'_j, i_t)$ then
13                         separating state pair mark removed from $< s'_a, s'_j >$
                           $AFF_Z := AFF_Z \cup$ TreeWalk($SF', < s'_a, s'_j >).V$;
                           $AFF_Z^{Work} =: AFF_Z^{Work} \cup$ TreeWalk($SF', < s'_a, s'_j >).V$;
14                         $E'_{SF} := E'_{SF} \setminus$ TreeWalk($SF', < s'_a, s'_j >).E$;
15                         $V'_{SF} := V'_{SF} \setminus$ TreeWalk($SF', < s'_a, s'_j >).V$;
16             else if $\neg(< s_a, s_j >$ sep. s. p. )and($\lambda'(s'_a, i) \neq \lambda'(s'_j, i)$) then
17                 $E'_{SF} := E'_{SF} \setminus \{(< s'_a, s'_j >, < \delta(s'_a), \delta(s'_j) >)\}$;
18                 $< s'_a, s'_j >$ marked as a separating state pair with $i$ sep. input
19                 $AFF_Z := AFF_Z \cup$ TreeWalk($SF', < s'_a, s'_j >).V$;
```

Algorithm 2: Maintaining a Separating Family of Sequences

```
    /* In case of a next state change */
20  else if $s_b \ncong s'_c$ then
21      foreach $s'_j \in S'$ do
22          if $(< s'_a, s'_j >, i, o, < s'_c, \delta(s'_j) >) \in E'_{SF}$ then
23              $AFF_Z := AFF_Z \cup$ TreeWalk$(SF, < s'_a, s'_j >).V$;
24              $AFF_Z^{Work} := AFF_Z^{Work} \cup$ TreeWalk$(SF, < s'_a, s'_j >).V$;
25              $E'_{SF} := E'_{SF} \setminus$ TreeWalk$(SF, < s'_a, s'_j >).E$;
26              $V'_{SF} := V'_{SF} \setminus$ TreeWalk$(SF, < s'_a, s'_j >).V$;
    /* Phase 2:  Create new test cases for affected state pairs */
    /* Create subgraph $A'_{AFF}$ for affected state pairs */
26  $V'_{AFF} := \emptyset$; $E'_{AFF} := \emptyset$;
27  foreach $v'_i \in AFF_Z^{Work}$ do
28      $V'_{AFF} := V'_{AFF} \cup \{v'_i\}$;
29      foreach $e = (v'_i, i, o, v'_j)$ do
30          if $v'_j \in AFF_Z^{Work}$ then  $E'_{AFF} := E'_{AFF} \cup (v'_i, v'_j)$
31          else $v'_i$ marked with $v'_j$
    /* Extend $SF'$ Spanning forest for affected state pairs */
32  foreach $v'_i$ marked with $v'_j$ do
33      $ST'_{temp}(V'_{temp}, E'_{temp})$ := SpanningTree$((A'_{AFF}), v'_i)$;
        $E'_{SF} := E'_{SF} \cup \{(v'_i, v'_j)\} \cup E'_{temp}$; $V'_{SF} := V'_{SF} \cup V'_{temp}$;
        $AFF_Z^{Work} := AFF_Z^{Work} \setminus V'_{temp}$;
34  return $SF' \in (V'_{SF}, E'_{SF})$, $MOD_Z$, $AFF_Z$, $AFF_Z^{Work}$, $A'$;
    /* Function $TreeWalk$ walks on a subtree of a given $SF'$ forest
       with a given root.  Function $SpanningTree$ creates a spanning
       tree for a given subset of state pairs in set $AFF_Z^{Work}$ which
       are reachable from a given root node in auxiliary graph $A'$.  */
```

When all elements of AFF_Z^{Work} have been checked, the SIM algorithm terminates:

- Spanning forest SF' represents the updated separating family of sequences Z' of the modified FSM M'.
- Set MOD_Z denotes the state pairs where the auxiliary graph A' has been modified.
- Set AFF_Z represents the state pairs of M' for which the separating sequences have been changed.
- Set AFF_Z^{Work} represents equivalent state pairs, thus the presented SIM algorithm is able to detect whether the machine is still minimized after the given sequence of changes Ω has been applied.

Thesis 1.4. *[C1] I have proven that the time complexity of the incremental algorithm for maintaining a separating family of sequences described above depends on the extent of changes, and can be described as $O(p \cdot \Delta_Z)$, where $n \leq \Delta_Z \leq n^2$.*

Proof. First of all, we note that for a unit change $|MOD_Z| = n$, and for multiple changes $n \leq |MOD_Z| \leq n \cdot (n+1)/2$. The cardinality of set AFF_Z is $0 \leq |AFF_Z| \leq n \cdot (n-1)/2$.

In Phase 1, the SIM algorithm updates auxiliary graph A into A' and identifies affected state pairs. Updating the auxiliary graph through changes Ω requires $O(|MOD_Z|)$ steps. To identify the affected states, put them into the AFF_Z and AFF_Z^{Work} sets and delete invalid spanning forest edges has a total time complexity of $O(|AFF_Z|)$.

In Phase 2, the SIM algorithm creates subgraph A'_{AFF} for AFF_Z^{Work} state pairs and marked state pairs, which have an edge leading to a non-AFF_Z^{Work} state pair with $O(p \cdot |AFF_Z|)$ complexity. Extending spanning forest SF' for AFF_Z^{Work} elements also requires $O(p \cdot |AFF_Z|)$ time complexity.

As $\Delta_Z = |MOD_Z \cup AFF_Z|$, the total time complexity of the SIM algorithm is $O(p \cdot \Delta_Z)$, where $n \leq \Delta_Z \leq n^2$.

□

Note that the space complexity of the incremental incremental algorithm for maintaining a Separating Family of Sequences is $O(p \cdot n^2)$, which is the same as that of the corresponding part of the traditional HIS-method.

4.2 Incremental Algorithms for Transition Tour Maintenance Across Updates

The Chinese Postman Problem (CPP) is to find a minimum length walk that covers each edge of the given graph at least once and return to the start node. The resulting sequence is called the postman tour of the given graph. The original problem was addressed in [18] and the different variations derived from the original CPP and their solutions are summarized in [11]. A special case of CPP is the Directed Chinese Postman Problem (DCPP), where each edge of the graph is directed. For FSM model based testing the Transition Tour (TT) method is proposed [22]. The TT-method produces a test sequence that traverses every transition of the deterministic, strongly connected specification machine at least once. The TT-method is equivalent to the DCPP with unit costs for each edge.

The traditional TT-method assumes rigid, unchanging specifications, thus test generation is very inefficient in case of iterative development scenarios. My goal was to find a solution to efficiently generate a Transition Tour test sequence for changing specifications.

Thesis 2. *[J2][C7] I have proposed two cooperating algorithms to maintain the Transition Tour test sequence across changes if the system specification is given as a deterministic, strongly connected FSM. The algorithms utilize existing information generated for the previous version of the specification to efficiently create the test sequence for the updated specification.*

Similarly to previous approaches solving the DCPP [10, 3, 20] my solution – which keeps the Transition Tour (TT) [22] test sequence up to date across changes – consists of two parts: Keep Graph Eulerian (KGE, presented in Section 4.2.1), which maintains a graph to keep it Eulerian after changes, and Keep Euler Tour (KET, presented in Section 4.2.2), which maintains the Euler tour in this updated Eulerian graph.

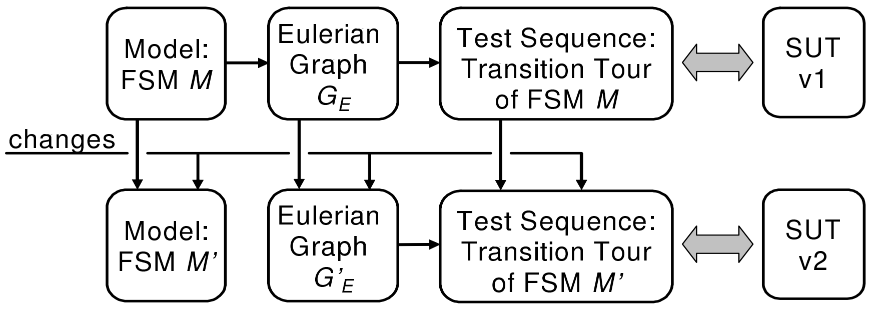

Figure 2: Block diagram of the incremental maintenance of Transition Tour

The high-level view of the approach is presented in Figure 2. The upper part of the figure shows the traditional generation of the TT test sequence: FSM M is transformed into an Eulerian graph G_E [1] and an Euler tour in G_E is generated. The Euler tour of graph G_E corresponds to the TT test sequence of FSM M, which can be applied to the system under test (SUT). The lower part of the figure shows how the sequence of changes affects the generation of TT. The changes turn FSM M into FSM M'. Instead of repeating the previous operations on M', the KGE algorithm (proposed in Section 4.2.1) maintains graph G_E across the given sequence of changes to keep it Eulerian; the resulting graph is denoted by G'_E. The KET algorithm (introduced in Section 4.2.2) maintains the Euler tour of G'_E, which corresponds to the TT sequence of the modified FSM M'. Then this TT test sequence can be applied

[1]FSM M can be transformed into a directed graph G, where edges and nodes of G correspond to the transitions and states of M, respectively. Then some edges of this graph G are duplicated such that the resulting graph will be Eulerian. This augmented graph is denoted by G_E.

to the modified system (SUT v2). Note that the results of the algorithms can be used as an input for the next iteration.

Based on the fault model of [4], the following changes are considered to modify the FSM:

- adding a transition,
- removing a transition,
- adding a state with one incoming and one outgoing transition,
- removing a state,
- changing the output label of a transition,
- changing the next state of a transition.

The output label change and the next state change of a transition are frequently applied steps in changing system specifications because they preserve the deterministic property of the FSM. Adding a state with one incoming and one outgoing transition is also a frequently applied step as it prevents the added state from becoming isolated. However, some changes described above may not preserve the strongly connected condition of the FSM. This is not a problem in our case, because the algorithm proposed in Section 4.2.2 is able to show if the strongly connected condition does not hold after a series of changes is applied to the graph of specification FSM M.

In the following, the graph representation of FSMs is used to describe the algorithms, thus the FSM fault operators discussed above are used in the graph representation form.

4.2.1 An algorithm to keep the graph Eulerian

Thesis 2.1. *[J2][C7] I have created an algorithm which keeps every node of the given Eulerian graph balanced across changes. The proposed algorithm guarantees that the subgraph on the augmenting edges remains acyclic, thus it avoids the overly increase of the given graph across the given sequence of changes. The algorithm is also able to detect if the original assumption on strongly connectedness does not hold after the sequence of changes has been applied.*

An edge of the original graph G or G' is called a *black edge.* An edge is called a *red edge* if it is a duplication of a black edge of the original graph to be transformed in an augmented Eulerian graph G_E or G'_E.

The KGE (Keep Graph Eulerian) algorithm consists of two main logical units, which solve two different subproblems:

1. A procedure that balances a given pair of nodes which has become unbalanced.
2. A main task that determines how the different types of changes make the nodes of graph G_E unbalanced, and uses the procedure above to balance them.

Balancing the unbalanced nodes

In a nutshell, the KGE algorithm creates new *green edges* between nodes, which have become unbalanced after the changes. The KGE algorithm substitutes these green edges into a sequence of red edges by duplicating some of the black edges that are present in the original graph. First, the path between two unbalanced nodes is found by breadth-first search, the edges of this path are added to G'_E as red edges, called *on-path edges.* After that, the algorithm checks whether the new on-path edges produce cycles with other red edges. If they do, then the algorithm deletes these cycles, i.e. eliminates the corresponding red edges. The reason why cycles can be eliminated is that each red edge has a corresponding black edge. Thus each edge of the original graph will be covered by the Euler tour of G'_E and the length of this tour can be reduced with the elimination of cycles formed by red edges. To determine and delete the cycles consisting of red edges, the KGE algorithm identifies the SCCs of the graph made up of red edges using Tarjan's algorithm [27].

Handling changes

This task determines, how the different types of changes turn the nodes of graph G_E unbalanced, and uses the previous procedure to balance them. If the changed edge has red copies, it also handles these red copies.

The KGE algorithm terminates, when all nodes of the graph become balanced. The resulting graph G'_E will become Eulerian and the subgraph on the red edges is again acyclic. The algorithm also shows if the strongly connected assumption does not hold after the given sequence of changes Ω have been applied.

4.2.2 An algorithm to maintain the Euler tour

Thesis 2.2. *[J2][C7] I have created an algorithm, that incrementally maintains the Euler tour described in the next-node representation form. The algorithm utilizes the Euler tour of the previous system version, and modifies only parts, that are affected by the given sequence of changes. The proposed algorithm is more efficient than traditional solutions in the case of small changes. The algorithm is also able to detect if the original assumption on strongly connection does not hold after the sequence of changes has been applied.*

The input of the algorithm consists of (1) an Euler tour in graph G_E before the changes in next-node representation [10], (2) the list of changes and (3) an Eulerian graph G'_E (with black and red edges) generated by the KGE algorithm proposed in Section 4.2.1. As the Euler tour is given in next-node representation, its maintenance can be reduced to the maintenance of a spanning tree TS (the edges of which are directed towards the root). Without loss of generality it is assumed that the spanning tree TS comprises exclusively of black edges.

The algorithm consists of three main logical units:

1. Procedures that describe frequently used steps that can be applied to update the Euler tour through edge changes. Such steps are:
    - Adding a new non-spanning edge to a node
    - Adding a spanning edge to a node
    - Removing a non-spanning edge from a node
    - Removing a spanning edge from a node
    - The handling of green edge substitution done by the first algorithm
2. A main task that determines how the different types of changes affect the next-node representation of the Euler tour and uses Procedures 1 to maintain it.
3. A task that constructs a subgraph and then creates new spanning edges for the following nodes: (i) nodes for which the directed path in the spanning tree are removed or changed during the update process, (ii) nodes which are added to the graph.

5 Applications of the results

The proposed incremental test generation algorithms are very efficient in case of large, evolving system specifications. Various assumptions about specification machines and differences in test cases (length of the entire test sequence vs. fault coverage) provide good scalability to the test engineer. The algorithms are suitable for the Agile working process because they support the step-by-step refinement of the system under development.

5.1 Possible applications of Thesis group 1

The proposed algorithms can speed up both test generation and test execution in most iterative development scenarios. The presented solution can be used in any model-based testing approach, such as testing of communication protocols and software. For example the given solution is applicable to test web portals, where the pages, the links, the parameters of HTTP requests and the text of the pages correspond to the states, transitions, input symbols and output symbols of FSMs, respectively. In the Dissertation the maintenance of a checking sequence has been investigated through the step-by-step refinement of the registration process of Session Initiation Protocol (SIP) [17].

The fact, that the algorithms keep the test cases of the unaffected part of specification unchanged, is also a huge benefit for the test engineer. For instance, a leading software in the field of industrial model-based testing tools, Conformiq Designer [1] stores the already generated test sets of previous system versions and reuses them if possible[2]. Conformiq Designer does not seem to generate the new test cases faster[3] in the case of changing specifications, but it does not force the designer to learn a completely different structure for the test cases of the new system version if a high proportion of old test cases from the previous system version are reusable.

5.2 Possible applications of Thesis group 2

Typically, model-based testing case studies deal only with state and transition coverage [13]. Thus the TT-method, which provides a coverage of all the states and transitions of FSMs, is adequate for most circumstances.

[2]This new feature appears first in Conformiq Qtronic 1.3 in 4th April of 2008, nearly a year after our [C1] paper.

[3]The internal working of Conformiq Designer is unknown, because it is not an open source tool. However, it seems to be that it checks with brute force if the old test cases is applicable for the new version of the system.

For software testing in which case the new code is compiled and tested only a few times, test generation of the traditional TT-method requires much more time than the execution of the test sequence. The proposed algorithms can efficiently speed up test generation of evolving specifications, thus my algorithms can save a significant amount of resources in the automated testing process.

Note that the proposed incremental algorithms can be also applicable to non-FSM based systems for protocol and software testing.

In the Dissertation the maintenance of the TT test sequence has been investigated through the step-by-step development of the registration process of SIP [17].

6 Acknowledgements

First of all, I would like to thank my supervisor, Zoltán Pap for guidance and support. I would like to express my gratitude to Gyula Csopaki for his continuous support and Gábor Kovács for his helpful comments.

I would like to thank the High Speed Networks Laboratory (HSNLab) for the technical and financial background. I am very grateful to Róbert Szabó and Erzsébet Győri for their administrative help.

I would also like to thank my roommates, Péter Babarczi and Levente Erős for their valuable advices and for the working time we spent together. I would also like to express my gratitude to Gusztáv Adamis and Levente Hunyadi for their help and Tamás Bőhm for the discussions related to SIP registration.

I am also very grateful to my friends, who cheered me up when I was feeling down.

Last, but not least I would like to thank my wife, Zsófi for her continuous and unconditional support.

7 Appendix – List of notations

A	auxiliary, state pair graph
AFF_Q	set of q-affected states
AFF_Q^{Work}	work set of set AFF_Q
AFF_Z	set of z-affected state pairs
AFF_Z^{Work}	work set of set AFF_Z
G	directed graph
G_E	augmented, Eulerian graph of G
i_x	an input symbol of an FSM
I	set of input symbols
$Impl$	implementation FSM
M	specification FSM
MOD	set of modified states
MOD_Z	set of modified state pairs
n	number of states of an FSM or the number of nodes in a graph
o_x	an output symbol of an FSM
O	set of output symbols
p	number of input symbols, i.e. $p = \|I\|$.
Q	State Cover Set of an FSM
r	reset symbol
S	set of states
s_k	a state of an FSM
$< s_i, s_j >$	a state pair of an FSM
SF	spanning forest of an auxiliary, state pair graph (edges are directed toward the root)
ST	spanning tree of an FSM (transitions are directed away from the root)

t	a transition of an FSM
T	set of transitions
TS	spanning tree of a graph (edges are directed toward the root)
x	input sequence
Z	separating family of sequences
Z_i	separating set of state s_i
λ	output function: λ: $S \times I \rightarrow O$
δ	next state function: δ: $S \times I \rightarrow S$
Δ	size of the modified size of the input and affected part of the output
Δ_Q	$\lvert MOD \cup AFF_Q \rvert$
Δ_Z	$\lvert MOD_Z \cup AFF_Z \rvert$
ω_k	atomic change
Ω	sequence of changes

References

[1] Conformiq Designer. Model-based testing tool. `http://www.conformiq.com/` Accessed: 2015-05-25.

[2] LEMON. A C++ library for efficient modeling and optimization in networks. `http://lemon.cs.elte.hu` Accessed: 2015-05-25.

[3] E. J. Beltrami and L. D. Bodin. Networks and vehicle routing for municipal waste collection. *Networks*, 4(1):65–94, 1974.

[4] Gregor von Bochmann, Anindya Das, Rachida Dssouli, Martin Dubuc, Abderrazak Ghedamsi, and Gang Luo. Fault Models in Testing. In *Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV*, pages 17–30, Amsterdam, The Netherlands, 1992. North-Holland Publishing Co.

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

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

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

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

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

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

[11] H. A. Eiselt, M. Gendreau, and G. Laporte. Arc Routing Problems, Part I: The Chinese Postman Problem. *Operations Research*, 43(2):231–242, 1995.

[12] Khaled El-Fakih, Nina Yevtushenko, and Gregor von Bochmann. FSM-Based Incremental Conformance Testing Methods. *IEEE Transactions on Software Engineering*, 30(7):425–436, 2004.

[13] ETSI. Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Application of MBT in ETSI case studies. Technical Report DTR/MTS 001411. `http://docbox.etsi.org/MTS/MTS/07-Drafts/00141_MBT_CaseStudies/` Accessed: 2015-05-25.

[14] S. Fujiwara, G. v. Bochmann, F. Khendec, M. Amalou, and A. Ghedamsi. Test selection based on finite state model. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[15] A. Gill. *Introduction to the theory of finite-state machines.* McGraw-Hill electronic sciences series. McGraw-Hill, 1962.

[16] James A. Highsmith. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[17] IETF. RFC 3261: SIP: Session Initiation Protocol, 2002. `https://tools.ietf.org/html/rfc3261` Accessed: 2015-05-25.

[18] M. K. Kwan. Graphic programming using odd or even points. *Chinese Math*, (1):273–277, 1962.

[19] David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines – A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[20] Y. Lin and Y. C. Zhao. A new algorithm for the directed chinese postman problem. *Computers and Operations Research*, 15(6):577–584, 1988.

[21] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[22] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition-tours. In *Proceedings of the 11th IEEE Fault-Tolerant Computing Conference (FTCS 1981)*, pages 238–243. IEEE Computer Society Press, 1981.

[23] Zoltán Pap, Gyula Csopaki, and Sarolta Dibuz. On the Theory of Patching. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods, SEFM*, pages 263–271, 2005.

[24] Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, and Anindya Das. Nondeterministic State Machines in Protocol Conformance Testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 363–378, 1994.

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

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

[27] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

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

Publications

Journal papers

[J1] G. Kovács, **G. Á. Németh**, Z. Pap, and M. Subramaniam: Deriving compact test suites for telecommunication software using distance metrics. In *Journal of Communications Software and Systems*, Volume: 5 Number: 2. 2009. pp. 57–61.

[J2] **G. Á. Németh** and Z. Pap: Incremental Maintenance of Transition Tour. In *Fundamenta Informaticae*, Volume: 129 Number: 3. 2014. pp. 279–300.

Book chapters

[B1] **G. Á. Németh**: Protocol Operation. In *Advanced Communication Protocol Technologies: Solutions, Methods and Applications*. Hershey; New York: IGI Global, Information Science Reference, 2011. pp. 20–37.

[B2] G. Kovács, **G. Á. Németh** and Z. Pap: Convergence of fix and mobile networks. In *Advanced Communication Protocol Technologies: Solutions, Methods and Applications*. Hershey; New York: IGI Global, Information Science Reference, 2011. pp. 226–246.

Lecture Notes in Computer Science conference papers

[C1] Z. Pap, M. Subramaniam, G. Kovács and **G. Á. Németh**: A Bounded Incremental Test Generation Algorithm for Finite State Machines. In *Proc., Testcom/Fates 2007*, Tallin, Estonia, June 2007. pp. 244–259.

[C2] G. Kovács, **G. Á. Németh**, Mahadevan Subramaniam and Zoltán Pap: Optimal String Edit Distance Based Test Suite Reduction for SDL Specifications. In *Proc., SDL 2009*, Bochum, Germany, September 2009. pp. 82–97.

[C3] G. Adamis, A. Wu-Hen-Chang, **G. Á. Németh**, L. Erős, G. Kovács: Data Flow Testing in TTCN-3 with a Relational Database Schema. In *Proc., SDL 2013*, Montreal, Canada, June 2013. pp. 1–18.

Other conference papers

[C4] G. Kovács, **G. Á. Németh**, Z. Pap, and M. Subramaniam: Deriving compact test suites for telecommunication software using distance metrics. In *Proc., SoftCOM 2008*, Split-Dubrovnik, Croatia, September 2008. pp. 394–398.

[C5] G. Németh and **G. Á. Németh**: A Generic Model for Advanced Networks Handling Imprecise Information. In *Proc., SoftCOM 2009*, Hvar, Croatia, September 2009. pp. 116–120.

[C6] **G. Á. Németh**, Z. Pap and G. Kovács: The Incremental Maintenance of a Checking Sequence. In *Proc., AACS'13*, Budapest, Hungary, June 2013. pp. 58–69.

Hungarian conference paper

[C7] **G. Á. Németh** and Z. Pap: Inkrementális tesztgenerálás. In *Tavaszi Szél 2012*, Győr, Hungary, 2012. pp. 442–448.

Independent Citations

[C1-1] A. Simão and A. Petrenko: Fault coverage-driven incremental test generation. In *Computer Journal* Volume 53, Issue 9, November 2010, pp. 1508–1522

[C1-2] A. Simão and A. Petrenko: Incremental Test Generation Guided by Fault Coverage: Technical Report. 2009. *http://www.crim.ca/Publications/2009/documents/plein_texte/ASD_SimA_al_0903-01.pdf*. Accessed: 2015-05-25. ISBN-13 : 978-2-89522-116-6. ISBN-10 : 2-89522-116-2.

[C1-3] E. Uzuncaova, S. Khurshid, and D. Batory: Incremental Test Generation for Software Product Lines. In *IEEE Transactions on Software Engineering*, Volume 36, Number 3, May/June 2010, pp. 309–322

[C1-4] E. Uzuncaova: Efficient Specification-based Testing Using Incremental Techniques. Dissertation. 2008. *http://www.library.utexas.edu/etd/d/2008/uzuncaovae51392/uzuncaovae51392.pdf*. Accessed: 2015-05-25.

[C1-5] H. Luo: On Distributed Multi-Point Concurrent Test System and Its Implementation. In *Social Informatics and Telecommunications Engineering* 4: 125–129 (2009)

[C1-6] L. Erős: Performance Testing and Performance Improvement Methods for Communicating Systems. Dissertation. 2013. *http://www.omikk.bme.hu/collections/phd/Villamosmernoki_es_Informatikai_Kar/2013/Eros_Levente/ertekezes.pdf*. Accessed: 2015-05-25.

[C2-1] Y. Ledru, A. Petrenko, S. Boroday and N. Mandran: Prioritizing test cases with string distances In *Automated Software Engineering*, Volume 19, Issue 1, March 2012, pp. 65–95